Linear Dependent Type Theory for Quantum Programming Languages

Extended Abstract

Peng Fu
Dalhousie University
Canada

Kohei Kishida
University of Illinois at
Urbana-Champaign
U.S.A.

Peter Selinger
Dalhousie University
Canada

Abstract

Modern quantum programming languages integrate quantum resources and classical control. They must, on the one hand, be linearly typed to reflect the no-cloning property of quantum resources. On the other hand, high-level and practical languages should also support quantum circuits as first-class citizens, as well as families of circuits that are indexed by some classical parameters. Quantum programming languages thus need linear dependent type theory. This paper defines a general semantic structure for such a type theory via certain fibrations of monoidal categories. The categorical model of the quantum circuit description language Proto-Quipper-M in [28] constitutes an example of such a fibration, which means that the language can readily be integrated with dependent types. We then devise both a general linear dependent type system and a dependently typed extension of Proto-Quipper-M, and provide them with operational semantics as well as a prototype implementation.

CCS Concepts: • **Software and its engineering** → **Semantics**.

Keywords: Quantum programming languages, linear dependent types, categorical model, fibration

ACM Reference Format:
Peng Fu, Kohei Kishida, and Peter Selinger. 2020. Linear Dependent Type Theory for Quantum Programming Languages: Extended Abstract. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '20), July 8–11, 2020, Saarbrücken, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3373718.3394765

ACM ISBN 978-1-4503-7104-9/20/07…$15.00
https://doi.org/10.1145/3373718.3394765

1 Introduction

1.1 Quantum programming and linear types

Quipper [12, 13] is a functional programming language for describing and generating quantum circuits. In addition to providing a low-level paradigm for generating circuits, where circuits are constructed by applying one gate at a time, it also provides a high-level paradigm, where circuits are first-class citizens to which meta-operations can be applied. This was found to provide an appropriate level of abstraction for formalizing many quantum algorithms in a way that is close to how they are informally described in the literature [33].

Quipper is implemented as an embedded language within Haskell and lacks formal foundations. This is why a family of languages known as Proto-Quipper [28, 29] has been developed as a formalization of suitable fragments of Quipper, susceptible to formal methods.

One aspect of the formal foundations missing in Quipper is linear types. The "no-cloning" property of quantum mechanics means that quantum resources are linear in the sense that one cannot duplicate a quantum state [22]. This is why linear types [10, 34] have been adopted to provide a type theory for quantum computing [31]. Since Haskell currently lacks linear types, Quipper leaves the programmer with the responsibility to keep track of the use of non-duplicable quantum resources. Proto-Quipper solves this problem by giving a syntactically sound linear type system, so that any duplication of a quantum state will be detected as a typing error at compile time.

Since Quipper is a circuit description language, it shares certain aspects of hardware description languages, such as a distinction between *circuit generation time* and *circuit runtime*. Values that are known at circuit generation time are called *parameters* and values that are known at circuit runtime are called *states*. The parameter/state distinction ties in with linearity, because although states can be linear (such as qubits) or nonlinear (such as classical bits used in circuits), parameters are always classical and therefore never linear. One of Quipper's design choices [33] is that parameters and states share the same name space; in fact, they can occur together in a single data structure, such as a pair of an integer and a qubit, or a list of qubits (in which case the length of the list is a parameter and the actual qubits in the list are states).

Among the languages of the Proto-Quipper family, Proto-Quipper-M [28] has several desirable features. Its linear type system accommodates nontrivial interactions of parameters and states. It supports higher-order functions and quantum data types such as lists of qubits. And, most significantly for our purposes, it has a denotational semantics in terms of a categorical model, called $\bar{\bar{\mathbf{M}}}$, that takes advantage of a linear-nonlinear adjunction in the sense of Benton [4].

1.2 Quantum programming and dependent types

The existing versions of Proto-Quipper do not support dependent types [18].

Dependent types are good at expressing program invariants and constraints at the level of types [23]. In the context of quantum circuit programming, dependent types give the programmer access to *dependent quantum data types*. For example, when one considers a linear function such as the quantum Fourier transform, it is really a family of circuits, rather than a single circuit, indexed by the length of the vector of qubits that the function takes as input. Another important application of dependent types in quantum circuit programming is the use of existential dependent quantum data types to hide garbage qubits [7, 21], while guaranteeing that they will be uncomputed. This facilitates the programming of large scale reversible circuits. It is therefore desirable to equip Proto-Quipper with dependent types.

The problem of how to formally add dependent types to a quantum programming language is not restricted to Proto-Quipper. The quantum circuit description language QWire [24] also uses a linear type system and has a denotational semantics (based on density matrices). It was recently embedded in the proof assistant Coq [27], making it possible to formally verify some simple quantum programs. QWire does not support quantum data types, much less dependent ones. Paykin et al. [24, Section 6.2] briefly discuss dependent types in the context of QWire but neither a semantic model nor a detailed analysis is given. The problem of defining a denotational semantics that can accommodate linear and dependent types for quantum circuit programming has remained open until now.

1.3 Linear dependent types

Designing a linear dependent type system is a challenge. One of the major hurdles is to understand what it means for a type to depend on a linear resource, and to provide a semantics for that kind of dependency. Indeed, suppose that a function f has a linear dependent type $(x : A) \multimap B[x]$ and that we are given a linear resource $a : A$. How should we understand $f\,a : B[a]$ (note that this seems to use a twice, once in a term and once in a type)?

Several approaches to linear dependent types have been proposed. Cervesato and Pfenning [6] extended the intuitionistic logical framework LF with linear types of the form $A \multimap B$. Their system separates a context into a linear part and an intuitionistic part: On the one hand, to form a lambda term $\hat{\lambda} x.M : A \multimap B$, the context $x \mathrel{\hat{:}} A$ must be a linear one. On the other hand, to form a lambda term $\lambda x.M : (x : A) \to B[x]$ of a dependent type, the context $x : A$ must be an intuitionistic one. When applying the linear lambda term $\hat{\lambda} x.M$ to another term, one combines two linear contexts as usual. On the other hand, when applying the intuitionistic lambda term $\lambda x.M$ to another term N, one has to make sure that N does not contain any linear variables. Note that in this system, linear function types and dependent function types are completely separate; effectively this means that parameters and states live in different name spaces. A similar approach is taken by Krishnaswami et al. [17], who proposed a type system extending Benton's logic LNL [4], Vakar [32], who extended Barber's dual intuitionistic linear logic [2] with dependent types, and Gaboardi et al. [9], who combined lightweight indexed types with bounded linear types [11].

From a unifying perspective, McBride [19] proposed the use of indices 0, 1, ω to decorate the context. The indices 0, 1, and ω mean that the variable they decorate is, respectively, never used, used exactly once, and used more than once. He then introduced linear dependent types of the form $(x :_k A) \multimap B[x]$, which correspond to the irrelevant quantification $\forall(x : A).B[x]$ à la Miquel [20] when $k = 0$, to Cervesato and Pfenning's dependent type when $k = \omega$, and to the linear dependent type when $k = 1$. The typing judgments in McBride's system are of the form $\Gamma \vdash M :_k A$, where the index k on the right of turnstile separates the set of typing judgments into two kinds: $\Gamma \vdash M :_0 A$ for terms that occur in types and $\Gamma \vdash M :_1 A$ for terms that will be evaluated at runtime. McBride's understanding of $f\,a :_1 B[a]$ is that the term a that occurs in the type $B[a]$ has been "contemplated" (which happens at type checking time) but should not be considered to have been "consumed" for the purpose of linearity (which can only happen at runtime).

In this paper, we propose to interpret the dependence of B on x in $(x : A) \multimap B[x]$ as saying that B depends on the *shape* of x. Informally, if N is a term of a quantum data type A, for example a tree whose leaves are qubits, then the shape of N, denoted $\mathrm{Sh}(N)$, is the same tree, but without the leaves (or more precisely, where the data in the leaves has been replaced by data of unit type). We write $\mathrm{Sh}(A)$ for the type of shapes of A, so that $\mathrm{Sh}(N) : \mathrm{Sh}(A)$. With these conventions, the dependence of B on x in $(x : A) \multimap B[x]$ is interpreted as $x : \mathrm{Sh}(A) \vdash B[x] : *$. It is understood that, for any resource N (linear or otherwise), its shape $\mathrm{Sh}(N)$ is duplicable. Accordingly, our typing rule for application has the following form (where Γ_i has the form

$x_1 :_{k_1} A_1, \ldots, x_n :_{k_n} A_n$, following McBride).

$$\frac{\Gamma_1 \vdash M : (x : A) \multimap B[x] \quad \Gamma_2 \vdash N : A}{\Gamma_1 + \Gamma_2 \vdash MN : B[\mathrm{Sh}(N)]}$$

So instead of separating typing contexts into linear and nonlinear parts like Cervesato and Pfenning, or of having two kinds of typing judgments like McBride and like Atkey [1], we identify a subset of typing judgment of the form $\mathrm{Sh}(\Gamma) \vdash \mathrm{Sh}(N) : \mathrm{Sh}(A)$. This subfragment of typing accounts for the (duplicable) intuitionistic fragment that does not change the quantum states. In this way, the notion of shape does not only play a conceptual role in our understanding of the dependence of B on A; it is indeed part of the formalism of our treatment of linear dependent types. Since the intuitionistic fragment $\mathrm{Sh}(\Gamma) \vdash \mathrm{Sh}(N) : \mathrm{Sh}(A)$ is only used for typing and kinding purposes, in practice programmers do not directly program with terms such as $\mathrm{Sh}(N)$.

A related concept under the name *shapely types* was considered by Jay and Cockett [14]. Their notion of *shapely types* is defined as certain pullbacks of the list functor. Although our notion of *shape-unit functor* is defined differently, it appears both notions of shapes coincide for data types such as lists. However, our notion of shape-unit functor or the shape operation can also be used for mapping a state-modifying program into a "pure" program that does not modify state. This is beyond Jay and Cockett's treatment of shapely types.

1.4 Contributions

In this paper, we provide a new framework of linear dependent type theory suitable for quantum circuit programming languages. The framework comes with both operational and denotational semantics. Indeed, the categorical structure for our denotational semantics subsumes Rios and Selinger's category $\overline{\overline{\mathbf{M}}}$, showing that Proto-Quipper-M can be extended with dependent types. Furthermore, our categorical structure, in terms of a fibration of a symmetric monoidal closed category over a locally cartesian closed category, can be viewed as a generalization of Seely's [30] semantics of nonlinear dependent type theory.

- In Section 2, we define a notion of state-parameter fibration. We then show how the state-parameter fibration naturally gives rise to concepts such as dependent monoidal product, dependent function space, and a *shape-unit functor* that interprets the shapes of types and terms.
- In Section 3, we provide typing rules and an operational semantics for a linear dependent type system that features the *shape operation*. We show that the shape operation corresponds to the shape-unit functor on the state-parameter fibration. We interpret the type system using the state-parameter fibration and prove the soundness of the interpretation.
- In Section 4, we show how to extend the linear dependent type system of Section 3 to support programming quantum circuits, giving rise to a dependently typed version of Proto-Quipper. We discuss some practical aspects of the implementation.

2 Categorical structure

Since Seely [30], it has been well understood how (nonlinear) dependent type theory can be interpreted in locally cartesian closed categories (LCCCs). To interpret a linear dependent type theory, one might naively look for an analogous notion of "locally monoidal closed category." However, no such structure is readily available. Instead, to build our model, we must take the notions of parameter and state seriously. We believe that the correct way to do this is to consider a fibration of a symmetric monoidal closed category (representing states) over an LCCC (representing parameters).

2.1 Pullbacks and fibrations

In modelling dependent types, pullbacks play the essential role of interpreting substitution of terms into types and contexts. To carry this idea over to the linear setting, the perspective of fibrations proves useful.

Definition 2.1. Given a functor $F : \mathbf{E} \to \mathbf{B}$, a morphism $f : B \to A$ of $\mathbf{E}$ is said to be *cartesian* if

- For every $g : C \to A$ in $\mathbf{E}$ and $b : F(C) \to F(B)$ in $\mathbf{B}$ such that $F(g) = F(f) \circ b$, there is a unique $u : C \to B$ such that $F(u) = b$ and $f \circ u = g$.

$$\begin{tikzcd}
C \arrow[d, mapsto] \arrow[rr, bend left, "g"] \arrow[r, dashed, "u"] & B \arrow[r, "f"] \arrow[d, mapsto] & A \arrow[d, mapsto] \\
F(C) \arrow[r, "b"'] & F(B) \arrow[r, "F(f)"'] & F(A)
\end{tikzcd}$$

Given an arrow $a : X \to F(A)$ of $\mathbf{B}$, we say that $f : B \to A$ is a *cartesian lifting* of a if $a = F(f)$ and f is cartesian. We say that F is a *Grothendieck fibration* if every arrow $a : X \to F(A)$ has a cartesian lifting at A.

As the picture above suggests, cartesian arrows are akin to pullbacks. In fact, we have the following:

Fact 2.2. Suppose $F : \mathbf{E} \to \mathbf{B}$ has a right adjoint G. Then an arrow f of $\mathbf{E}$ is cartesian iff the following square is a pullback, where η is the unit of the adjunction.

$$\begin{tikzcd}
B \arrow[r, "f"] \arrow[d, "\eta_B"'] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, "\eta_A"] \\
GF(B) \arrow[r, "GF(f)"] & GF(A)
\end{tikzcd}$$

Indeed, the following subclass of fibrations can be characterized in terms of adjoints and pullbacks.

Definition 2.3. Let us call a Grothendieck fibration $\sharp : \mathbf{E} \to \mathbf{B}$ a *pullback fibration* if

(1) $\mathbf{B}$ has pullbacks, and
(2) $\mathbf{E}$ has a terminal object 1 and $\sharp$ preserves it.

Fact 2.4. Given any functor $\sharp : \mathbf{E} \to \mathbf{B}$, it is a pullback fibration iff

(3) $\sharp$ has a full and faithful right adjoint q, making $\mathbf{B}$ a reflective subcategory of $\mathbf{E}$, and moreover the counit of the adjunction is strictly the identity.
(4) $\mathbf{B}$ has a terminal object 1.
(5) Any arrows of $\mathbf{E}$ of the form $qf : qY \to qX$ and $h : A \to qX$ have a pullback B, and $\sharp$ preserves it to a pullback of $\sharp qf = f$ and $\sharp h$.

$$\begin{tikzcd} B \arrow[r, "\pi_2"] \arrow[d, "\pi_1"'] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, "h"] \\ qY \arrow[r, "qf"] & qX \end{tikzcd}$$

In particular, if $X = \sharp A$ and $h = \eta_A$ for the unit of adjunction $\eta : \mathrm{id}_{\mathbf{E}} \to q \circ \sharp$, then qf and h have a pullback B as above with $Y = \sharp B$, $\pi_1 = \eta_B$, and $f = \sharp \pi_2$ (hence π_2 is the cartesian lifting of f).

It is useful to note how a pullback fibration $\sharp$ defines its right adjoint q in (3): since $\sharp 1$ is terminal in $\mathbf{B}$, we set $q\sharp 1 = 1$ and define qX and qf by cartesian liftings:

$$\begin{tikzcd} qY \arrow[r, "qf"] \arrow[d, mapsto] & qX \arrow[r, "q!_X"] \arrow[d, mapsto] & q\sharp 1 \arrow[d, mapsto] \\ Y \arrow[r, "f"] & X \arrow[r, "!_X"] & \sharp 1. \end{tikzcd}$$

Notation 2.5. By abuse of notation, we may write X for qX (since q is a full embedding), and $\underline{A}$ for both $\sharp A$ and $q\sharp A$.

We use the total category $\mathbf{E}$ of such a fibration $\sharp$ as a category of quantum structures. It may not have all pullbacks, but the ones given by $\sharp$ are enough for interpreting dependent types.

2.2 State-parameter fibration

Our semantics takes a pullback fibration $\sharp$ of a symmetric monoidal closed category $\mathbf{E}$ of quantum (and other) state spaces over a locally cartesian closed subcategory $\mathbf{B}$ of parameter spaces. A state space A of $\mathbf{E}$ then sits over a parameter space $\sharp A$ via $\eta_A : A \to q\sharp A$. With this we express a parameter-indexed family of quantum states—formally by assuming:

(6) $\sharp$ is a monoidal closed functor.

This means that $\sharp$ preserves the monoidal closed structure, i.e., it maps $\otimes$, $\multimap$, I in $\mathbf{E}$ to $\times$, $\Rightarrow$, 1 in $\mathbf{B}$.

In this setup, cartesian arrows are understood as simply reindexing parameters, without any effect on quantum states. This understanding implies that

(7) If arrows f and g of $\mathbf{E}$ are cartesian, so is $f \otimes g$.

Our setup also gives rise to several useful structures. One is a "sublocal" monoidal structure:

Definition 2.6. Let $\sharp : \mathbf{E} \to \mathbf{B}$ be a monoidal closed pullback fibration, and $\mathbf{B}$ be an LCCC, so that therefore each slice category $\mathbf{B}/X$ is cartesian closed, with $\times_X$ and $\Rightarrow_X$. Let A, B be objects in $\mathbf{E}/qX$. We then define the *fibered monoidal product* $A \otimes_{qX} B$ and the *fibered function space* $A \multimap_{qX} B$ as the following cartesian liftings.

$$\begin{tikzcd} A \otimes_{qX} B \arrow[r, tail] \arrow[d, mapsto] & A \otimes B \arrow[d, mapsto] & A \multimap_{qX} B \arrow[r, tail] \arrow[d, mapsto] & A \multimap B \arrow[d, mapsto] \\ \underline{A} \times_X \underline{B} \arrow[r, tail] & \underline{A} \times \underline{B} & \underline{A} \Rightarrow_X \underline{B} \arrow[r, tail] & \underline{A} \Rightarrow \underline{B} \end{tikzcd}$$

$A \otimes_{qX} B$ and $A \multimap_{qX} B$ are also objects of $\mathbf{E}/qX$. We will assume that

(8) $- \otimes_{qX} A \dashv A \multimap_{qX} -$ for every object X of $\mathbf{B}$,

so that each $\mathbf{E}/qX$ is monoidal closed. We call $\mathbf{E}$ "sublocally" monoidal closed: while $\mathbf{E}/A$ is not monoidal closed for every object A, it is if A is in the subcategory $\mathbf{B}$.

Another structure is a functor that gives parameter-indexed families of monoidal units:

Definition 2.7. Given a monoidal closed pullback fibration $\sharp : \mathbf{E} \to \mathbf{B}$, we define a functor $p : \mathbf{B} \to \mathbf{E}$ by setting $p1 = I$ and taking cartesian liftings of the entire $\mathbf{B}$, or equivalently, pulling back $!_{p1} = \eta_I : I \to \underline{I}$.

$$\begin{tikzcd} pY \arrow[r, "pf"] \arrow[d, mapsto] & pX \arrow[r, "p!_X"] \arrow[d, mapsto] & p1 \arrow[d, mapsto] & pY \arrow[r, "pf"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & pX \arrow[r, "p!_X"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & p1 \arrow[d, "!_{p1}=\eta_I"] \\ Y \arrow[r, "f"] & X \arrow[r, "!_X"] & 1 & qY \arrow[r, "qf"] & qX \arrow[r, "q!_X"] & q1 \end{tikzcd}$$

It follows that $\sharp \circ p = \mathrm{id}_{\mathbf{B}}$. We see pX as a family of the monoidal unit I indexed by parameters in X; it is indeed the monoidal unit of $\mathbf{E}/qX$. (One may also note that $pX = I \times qX$.) We may therefore refer to each pX as an "X-parameterized unit." We will assume that

(9) p has a right adjoint $\flat$,

which is enough to make $p \dashv \flat$ a *linear-nonlinear adjunction* as defined by Benton [4]. We then write ! for the comonad $p \circ \flat$ and force : ! $\to \mathrm{id}_{\mathbf{E}}$ for the counit.

Putting everything together, we arrive at the following definition.

Definition 2.8. We say that a pullback fibration $\sharp : \mathbf{E} \to \mathbf{B}$ over an LCCC $\mathbf{B}$ (see (1)–(5)) is a *linear-state-nonlinear-parameter fibration*, or *state-parameter fibration* for short, if (6)–(9) hold. We call the associated $q, p : \mathbf{B} \to \mathbf{E}$ the *parameterized-terminal* and *parameterized-unit* functors, respectively, of $\sharp$.

Fact 2.9. Given any state-parameter fibration $\sharp$, its parameterized-unit functor p is strong monoidal, i.e., $p(X \times Y) \cong$

$pX \otimes pY$ and $p1 \cong I$, and $p \dashv \flat$ is a linear-nonlinear adjunction.

Proof. $p1 \cong I$ by definition. To show $p(X \times Y) \cong pX \otimes pY$, note that $p!_X : pX \to p1 \cong I$ and $p!_Y$ are cartesian; hence $p!_X \otimes p!_Y : pX \otimes pY \to p1 \otimes p1 \cong p1$ is also cartesian by (7), while $\underline{pX \otimes pY} \cong X \times Y$. Thus $pX \otimes pY = p(X \times Y)$ by Definition 2.7. Then, by [16, Theorem 1.5], strong monoidal p makes $\flat$ monoidal and $p \dashv \flat$ a linear-nonlinear adjunction. □

2.3 Example: $\bar{\bar{\mathbf{M}}}$ over Set

Rios and Selinger's [28] categorical model of Proto-Quipper-M naturally gives rise to a state-parameter fibration. This concrete example may help the reader see how our idea of states and parameters works conceptually.

The definition of the model first fixes a symmetric monoidal category $\mathbf{M}$ of "generalized circuits" (which could be, for example, a category of quantum circuits), and fully embeds it into some symmetric monoidal closed category $\bar{\mathbf{M}}$ with products, e.g., by taking the Yoneda embedding. Then

Definition 2.10. Let $\bar{\bar{\mathbf{M}}}$ be the free coproduct completion of $\bar{\mathbf{M}}$. More concretely, it is given as follows:

- An object A is a pair $(\underline{A}, (A_x)_{x \in \underline{A}})$ of a set $\underline{A}$ and an indexed family of objects A_x of the category $\bar{\mathbf{M}}$.
- An arrow $f : A \to B$ is a pair $(\underline{f}, (f_x)_{x \in \underline{A}})$ of a function $\underline{f} : \underline{A} \to \underline{B}$ and an indexed family of arrows $f_x : A_x \to B_{\underline{f}(x)}$ of $\bar{\mathbf{M}}$.

Objects and arrows of $\bar{\bar{\mathbf{M}}}$ are families of objects and morphisms of $\bar{\mathbf{M}}$, indexed by parameters in sets. The obvious functor $\sharp : \bar{\bar{\mathbf{M}}} \to \mathbf{Set} :: A \mapsto \underline{A}$ is a Grothendieck fibration: the cartesian lifting of a function $f : X \to \underline{A}$ at A has domain $(X, (A_{f(x)})_{x \in X})$ and consists of f paired with a family $(\mathrm{id}_{A_{f(x)}})_{x \in X}$ of identity arrows—thus, cartesian arrows simply reindex parameters and have no effect on states. Since $\sharp$ preserves the terminal $(1, 1)$ of $\bar{\bar{\mathbf{M}}}$ to 1 (and since $\mathbf{Set}$ is an LCCC), it is a pullback fibration. While $\sharp$ has a full and faithful right adjoint $q : \mathbf{Set} \to \bar{\bar{\mathbf{M}}} :: X \mapsto (X, (1)_{x \in X})$, there is also $p : \mathbf{Set} \to \bar{\bar{\mathbf{M}}} :: X \mapsto (X, (I)_{x \in X})$, which can be obtained as in Definition 2.7. Since p has a right adjoint $\flat : \bar{\bar{\mathbf{M}}} \to \mathbf{Set} :: A \mapsto \sum_{x \in \underline{A}} \bar{\mathbf{M}}(I, A_x)$, property (9) holds.

The category $\bar{\bar{\mathbf{M}}}$ is symmetric monoidal closed, with I, $\otimes$, $\multimap$ defined by indexed families of I, $\otimes$, $\multimap$ of $\bar{\mathbf{M}}$ with parameter sets that make $\sharp$ monoidal closed: $I = (1, I)$, $A \otimes B = (\underline{A} \times \underline{B}, (A_x \otimes B_y)_{x \in \underline{A}, y \in \underline{B}})$, and $A \multimap B = (\underline{A} \Rightarrow \underline{B}, (\prod_{x \in \underline{A}} (A_x \multimap B_{f(x)}))_{f : \underline{A} \to \underline{B}})$. This definition and the characterization of cartesian arrows above imply (7). Moreover, $A \otimes_{qX} B$ and $A \multimap_{qX} B$ are defined to be subfamilies of $A \otimes B$ and $A \multimap B$ (with smaller parameter sets $\underline{A} \times_X \underline{B}$ and $\underline{A} \Rightarrow_X \underline{B}$), which implies (8), that $\bar{\bar{\mathbf{M}}}/qX$ is monoidal closed. To sum up,

Fact 2.11. The functor $\sharp : \bar{\bar{\mathbf{M}}} \to \mathbf{Set}$ is a state-parameter fibration.

The remainder of this section and Section 3 explore how state-parameter fibrations can model linear dependent types. Fact 2.11 then means that Proto-Quipper-M can be extended with dependent types, as we will see in Section 4.

2.4 Structures for dependent types

The sublocally monoidal closed structure of a state-parameter fibration $\sharp : \mathbf{E} \to \mathbf{B}$ enables us to interpret dependent types with several constructions.

First let us note that any arrow $\pi : A \to qX$, i.e., any object of $\mathbf{E}/qX$, factors as $\underline{\pi} \circ \eta_A$. We take this to signify that A, intuitively an $\underline{A}$-indexed family of state spaces, is dependent, via $\underline{\pi}$, on parameters in X. In the same vein, any arrow of $\mathbf{E}/qX$ is parameterized by X. Based on this idea, we use $\mathbf{E}/qX$ as the category of (objects and arrows interpreting) types and terms dependent on parameters in X. So, schematically, a context $\Gamma \vdash$, a dependent type $\Gamma \vdash A : *$, and a term $\Gamma \vdash M : A$ are interpreted respectively by Γ, π, and M in

$$\begin{tikzcd} \Gamma \arrow[r, "M"] \arrow[dr, "\eta_\Gamma"'] & A \arrow[d, "\pi"] \\ & \underline{\Gamma} \end{tikzcd}$$

In the LCCC interpretation of nonlinear dependent type theory, one would use an identity arrow $\mathrm{id}_\Gamma : \Gamma \to \Gamma$ on the left of the triangle. In our semantics, it is crucial to use $\eta_\Gamma : \Gamma \to \underline{\Gamma}$ instead—because, while M is dependent on parameters, it may use resources contained in its domain. That is, to generalize the LCCC interpretation to the linear dependent case, we need to split the context object into two: a parameterized terminal $\underline{\Gamma}$ as the base of the slice $\mathbf{E}/\underline{\Gamma}$ to express dependency, and a (generally) linear object Γ as the domain of a term to embody linear resources.

One may still note that, when we apply the functor $\sharp$ to the above triangle, $\underline{\eta_\Gamma} = \mathrm{id}_{\underline{\Gamma}}$ becomes the usual triangle from the nonlinear case. In other words, our semantics uses a fibration $\sharp$ of the linear type theory of $\mathbf{E}$ over the nonlinear dependent type theory of parameters of the LCCC $\mathbf{B}$.

Now, keeping in mind that $\pi : B \to \underline{A}$ means the dependency of B on (parameters of) A, we define:

Definition 2.12. Given $\pi : B \to \underline{A}$, i.e., when B is in $\mathbf{E}/\underline{A}$, we call $A \otimes_{\underline{A}} B$ a *dependent monoidal product*, and write $A \overleftarrow{\otimes} B$ for it.

Dependent monoidal products will be used to interpret linear Σ-types $(x : A) \otimes B$, but they will also prove necessary in interpreting contexts. If $\Gamma, x : A \vdash$ is a well-formed context, we must have $\Gamma \vdash A : *$, and therefore an arrow $\pi : A \to \underline{\Gamma}$; then $\Gamma \overleftarrow{\otimes} A$ is the interpretation of $\Gamma, x : A \vdash$. In the nonlinear case, an object X would interpret the context $\Gamma, x : X \vdash$, but in the linear case, we must not drop Γ from $\Gamma \overleftarrow{\otimes} A$, since the linear resource contained in Γ is crucial. Observe however that $\underline{A \overleftarrow{\otimes} B} = \underline{A} \times_{\underline{A}} \underline{B} \cong \underline{B}$. Thus, as far as parameters are concerned, the context $\Gamma, x : A \vdash$ has a parameter space $\underline{A}$, agreeing with the nonlinear dependent type theory of $\mathbf{B}$. One may in addition observe that $\overleftarrow{\otimes}$ is associative.

For linear Π-types, we introduce the following definition:

Definition 2.13. Let A be an object in $\mathbf{E}/qX$ and B an object in $\mathbf{E}/\underline{A}$ (and hence in $\mathbf{E}/qX$). We define the *dependent function space* $\Pi_{qX,A}B$ in $\mathbf{E}$ as the following cartesian lifting, where $\underline{\Pi}_{X,\underline{A}}\underline{B}$ is the dependent function space in $\mathbf{B}/X$.

$$\begin{tikzcd}
\Pi_{qX,A}B \arrow[r, tail] \arrow[d, mapsto] & A \multimap_{qX} B \arrow[d, mapsto] \\
\underline{\Pi}_{X,\underline{A}}\underline{B} \arrow[r, tail] & \underline{A} \Rightarrow_X \underline{B}
\end{tikzcd}$$

This can interpret linear Π-types $(x : A) \multimap B$ due to the following theorem, which provides a semantics for linear dependent function abstraction and application.

Theorem 2.14. *There is an adjunction*

$$\begin{tikzcd}
\bar{\bar{\mathbf{M}}}/\underline{A} \arrow[r, bend right, "\Pi_{qX,A}"'] \arrow[r, phantom, "\perp"] & \bar{\bar{\mathbf{M}}}/qX \arrow[l, bend right, "- \otimes_{qX} A"']
\end{tikzcd}$$

2.5 Parameters and shapes

The linear type theory of $\mathbf{E}$ is fibered over the nonlinear dependent type theory of parameters in $\mathbf{B}$, so that dependent types are dependent on parameters. This role of parameters will be reflected in the syntax of the type theory we will propose in Section 3, which specifies special subclasses of types and terms, namely, *parameter types* and *parameter terms*. They refer primarily to objects and arrows in $\mathbf{B}$. Nonetheless, given our objective of treating parameter types and other types uniformly, we need to formally interpret parameter types and terms in $\mathbf{E}$. We do this by embedding $\mathbf{B}$ into $\mathbf{E}$ in two ways, namely, by the functors q and p: while the image of q provides slices $\mathbf{E}/qX$ and sublocally monoidal closed structure, parameterized monoidal units pX and reindexing maps pf interact with other objects and arrows within the monoidal structure of $\mathbf{E}$ or $\mathbf{E}/qX$, in the way that was explained in Section 2.4.

For this reason, we interpret the shape operation on types and terms by $\sharp$ but also by the two endofunctors $q \circ \sharp$ and $p \circ \sharp$ on $\mathbf{E}$, which we dub the *shape-terminal functor* and *shape-unit functor*, respectively. While $q \circ \sharp$ is a monad (of $\sharp \dashv q$), $p \circ \sharp$ is not, but one may note that $p \circ \sharp \dashv q \circ \flat$. The following equalities regarding $p \circ \sharp$ will be useful in interpreting shapes in Section 3.

Fact 2.15. The shape-unit functor $p \circ \sharp$ satisfies the following for any X in $\mathbf{B}$ and any A, B in $\mathbf{E}$:

- $p(\underline{pX}) = pX$.
- $p\underline{S} = I$ for any S in $\mathbf{E}$ such that $\underline{S} = 1$.
- $p(\underline{A \otimes B}) = p\underline{A} \otimes p\underline{B}$.
- $p(\underline{A \otimes_{qX} B}) = p\underline{A} \otimes_{qX} p\underline{B}$.
- $p(\underline{!A}) = !A$.
- $p(\underline{A \multimap B}) = p(\underline{A} \Rightarrow \underline{B})$.
- $p(\underline{\Pi_{qX,A}B}) = p(\underline{\Pi}_{X,\underline{A}}\underline{B})$.

Note that $p(\underline{A \multimap B}) \neq p\underline{A} \multimap p\underline{B}$, since p does not map $\Rightarrow$ to $\multimap$. Although $p(\underline{A \multimap B}) = p(\underline{A} \Rightarrow \underline{B})$ is not an exponential in $\mathbf{E}$, it is (trivially) an exponential in $p\mathbf{B}$, the (non-full) subcategory of images of p. Parameterized units and cartesian arrows in $p\mathbf{B}$ correspond to a state-free fragment of $\mathbf{E}$, in which (nonlinear) function abstraction and application are interpreted by the exponential structure of $\mathbf{B}$ (or its image under p).

3 A linear dependent type system

We now consider a type system for the state-parameter fibration $\sharp : \mathbf{E} \to \mathbf{B}$.

3.1 Syntax and typing rules

As was mentioned in Section 2.5, the syntax of our type theory specifies subclasses of types and terms, namely, *parameter types* and *parameter terms*, in parallel to the LCCC $\mathbf{B}$ being the subcategory of $\mathbf{E}$ with no linear resources or state. Furthermore, our syntax allows only parameter terms to appear in types. This restriction has the consequence that the evaluation of types during type checking will not change quantum states.

Definition 3.1 (Syntax).

Types $A, B ::= C \mid \mathbf{Unit} \mid !A \mid (x : A) \multimap B[x]$
  $\mid (x : A) \otimes B[x] \mid (x : P_1) \to P_2[x]$
Parameter types $P ::= \mathbf{Unit} \mid !A \mid (x : P_1) \otimes P_2[x]$
  $\mid (x : P_1) \to P_2[x]$
Terms $M, N, L ::= x \mid \lambda x.M \mid MN \mid \mathsf{force}\, M \mid \mathsf{force}'\, R$
  $\mid \mathsf{lift}\, M \mid (M, N) \mid \mathsf{let}\, (x, y) = N \,\mathsf{in}\, M \mid \lambda' x.R \mid R_1 @ R_2$
Parameter terms $R ::= x \mid \lambda' x.R \mid R_1 @ R_2 \mid \mathsf{force}'\, R \mid \mathsf{lift}\, M$
  $\mid (R_1, R_2) \mid \mathsf{let}\, (x, y) = R_1 \,\mathsf{in}\, R_2$
Values $V ::= x \mid \lambda x.M \mid \lambda' x.R \mid \mathsf{lift}\, M$
Indices $k ::= 0 \mid 1 \mid \omega$
Contexts $\Gamma ::= \cdot \mid x :_k A, \Gamma$
Parameter contexts $\Phi ::= \cdot \mid x :_k A, \Phi$, where $k = 1$ only if A is a parameter type.

Here, C denotes a basic type of states, e.g., qubits or bits. We have the usual linear exponential type $!A$, which is a parameter type and is duplicable. The term $\mathsf{lift}\, M$ introduces the type $!A$, and the term $\mathsf{force}\, M$ eliminates $!A$ to A. In the linear Π- and Σ-types $(x : A) \multimap B[x]$ and $(x : A) \otimes B[x]$, we allow A to be any type. We write $A \multimap B$ and $A \otimes B$ for them in the special case in which B does not depend on $x : A$.

The linear Σ-type $(x : A) \otimes B[x]$ is a parameter type when A and $B[x]$ are. The same is not the case for the Π-type: we introduce a separate, intuitionistic Π-type $(x : P_1) \to P_2[x]$.

This is introduced and eliminated by parameter terms $\lambda' x.R$ and $R_1 @ R_2$, respectively.

The term $\mathsf{force}'\, R$ eliminates $!A$ to the shape of A, i.e., $\mathrm{Sh}(A)$. We interpret force' as the shape of the counit force : $! \to \mathrm{id}_{\mathrm{E}}$ of the adjunction $p \dashv \flat$.

The operations λ', $@$, and force', as well as the corresponding type former $\to$, are not part of the "surface language", i.e., they are not intended to be used directly by the user of the programming language. Rather, they are generated by the type checker and only play a role in type checking and during evaluation.

We use the indices 0, 1, ω to keep track of the use of variables in a typing context. In a well-formed context, a linear type can only have the index 0 or 1, and not ω. On the other hand, a parameter type can have any index. Addition and multiplication of the indices are defined to be commutative so that 0 and 1 are the additive and multiplicative units, with $k + \ell = \omega$ for $k, \ell \neq 0$ and with $0 \cdot k = 0$ and $\omega \cdot \omega = \omega$. It is straightforward to extend these operations pointwise to contexts: given two contexts $\Gamma_1 = (x_1 :_{k_1} A_1, \ldots, x_n :_{k_n} A_n)$ and $\Gamma_2 = (x_1 :_{\ell_1} A_1, \ldots, x_n :_{\ell_n} A_n)$ with the same sequence of types, we write $\Gamma_1 + \Gamma_2 = (x_1 :_{k_1+\ell_1} A_1, \ldots, x_n :_{k_n+\ell_n} A_n)$, while $k\Gamma_1 = (x_1 :_{k \cdot k_1} A_1, \ldots, x_n :_{k \cdot k_n} A_n)$.

Parameter contexts Φ are typing contexts in which all linear types have the index 0. Note that parameter contexts are just a special kind of contexts, rather than being formally distinct from other contexts Γ.

One of the most significant pieces of machinery in our type system is the *shape operation*, which maps a term to a parameter term and a type to a parameter type. It is interpreted by the shape-unit functor $p \circ \sharp$, and the type-part of the operation corresponds to equalities in Fact 2.15.

Definition 3.2 (Shape operation).

$$\begin{aligned}
&\mathrm{Sh}(C) = \mathbf{Unit}\\
&\mathrm{Sh}(!A) = !A\\
&\mathrm{Sh}((x : A) \multimap B[x]) = (x : \mathrm{Sh}(A)) \to \mathrm{Sh}(B[x])\\
&\mathrm{Sh}((x : A) \otimes B[x]) = (x : \mathrm{Sh}(A)) \otimes \mathrm{Sh}(B[x])\\
&\mathrm{Sh}(P) = P
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Sh}(x) = x\\
&\mathrm{Sh}(\lambda x.N) = \lambda' x.\, \mathrm{Sh}(N)\\
&\mathrm{Sh}(MN) = \mathrm{Sh}(M) @ \mathrm{Sh}(N)\\
&\mathrm{Sh}(\mathsf{force}\, M) = \mathsf{force}'\, \mathrm{Sh}(M)\\
&\mathrm{Sh}(\mathsf{lift}\, M) = \mathsf{lift}\, M\\
&\mathrm{Sh}(M, N) = (\mathrm{Sh}(M), \mathrm{Sh}(N))\\
&\mathrm{Sh}(\mathsf{let}\, (x, y) = N \,\mathsf{in}\, M) = \mathsf{let}\, (x, y) = \mathrm{Sh}(N) \,\mathsf{in}\, \mathrm{Sh}(M)\\
&\mathrm{Sh}(R) = R
\end{aligned}$$

Observe that the shape operation is idempotent, just like the shape-unit functor $p \circ \sharp$. Also note that the shape operation is a meta-operation on terms, like substitution, and not a term constructor. The shape operation is intended to be applied to well-typed terms. Although we define $\mathrm{Sh}(x) = x$, the type of the variable x is changed according to the shape operation (see Theorem 3.6).

Let us write $\mathrm{Sh}(\Gamma)$ to mean applying the shape operation to all the types in Γ. So $\mathrm{Sh}(\Gamma)$ is a parameter context. We often ignore the index information of a variable if it is of a parameter type P.

The well-formedness of the context depends on kinding. The main purpose of a well-formed context is to make sure a linear type cannot have index ω. The next three definitions are mutually recursive, as is common in dependent type theory.

Definition 3.3 (Well-formed context). $\boxed{\Gamma \vdash}$

$$\frac{}{\cdot \vdash} \qquad \frac{\Gamma \vdash \quad \mathrm{Sh}(\Gamma) \vdash A : *}{\Gamma, x :_k A \vdash} \ \text{where } k = \omega \text{ only if } A \text{ is a parameter type}$$

In the above definition, we use the shape of the context $\mathrm{Sh}(\Gamma)$ to kind check a type.

Definition 3.4 (Selected kinding rules). $\boxed{\Phi \vdash A : *}$

$$\frac{\Phi, x : \mathrm{Sh}(A) \vdash B[x] : *}{\Phi \vdash (x : A) \multimap B[x] : *} \qquad \frac{\Phi, x : \mathrm{Sh}(A) \vdash B[x] : *}{\Phi \vdash (x : A) \otimes B[x] : *}$$

$$\frac{\Phi \vdash A : *}{\Phi \vdash !A : *} \qquad \frac{\Phi, x : P_1 \vdash P_2[x] : *}{\Phi \vdash (x : P_1) \to P_2[x] : *}$$

We only use parameter contexts for kinding, since only parameter terms can appear in types. The kinding rules for $(x : A) \otimes B[x]$ and $(x : A) \multimap B[x]$ suggest that the type B only depends on the shape of A, i.e., $\mathrm{Sh}(A)$, which is a parameter type.

Definition 3.5 (Selected typing rules). $\boxed{\Gamma \vdash M : A}$

$$\frac{\Gamma_1 \vdash M : (x : A) \multimap B[x] \quad \Gamma_2 \vdash N : A}{\Gamma_1 + \Gamma_2 \vdash MN : B[\mathrm{Sh}(N)]} \qquad \frac{0\Gamma, x :_1 A, 0\Gamma' \vdash}{0\Gamma, x :_1 A, 0\Gamma' \vdash x : A}$$

$$\frac{\Gamma, x :_k A \vdash M : B[x] \quad k \neq 1 \Rightarrow A = P}{\Gamma \vdash \lambda x.M : (x : A) \multimap B[x]} \qquad \frac{\Phi \vdash M : A}{\Phi \vdash \mathsf{lift}\, M : !A}$$

$$\frac{\Gamma_1 \vdash M : A \quad \Gamma_2 \vdash N : B[\mathrm{Sh}(M)]}{\Gamma_1 + \Gamma_2 \vdash (M, N) : (x : A) \otimes B[x]} \qquad \frac{\Gamma \vdash M : !A}{\Gamma \vdash \mathsf{force}\, M : A}$$

$$\frac{\Phi, x : P_1 \vdash R : P_2[x]}{\Phi \vdash \lambda' x.R : (x : P_1) \to P_2[x]} \qquad \frac{\Phi \vdash R : !A}{\Phi \vdash \mathsf{force}'\, R : \mathrm{Sh}(A)}$$

$$\frac{\Phi \vdash R_1 : (x : P_1) \to P_2[x] \quad \Phi \vdash R_2 : P_1}{\Phi \vdash R_1 @ R_2 : B[R_2]}$$

Remark. (i) In the typing rule for MN, the parameter term $\mathrm{Sh}(N)$ is substituted into the type, therefore the term N only occurs once in the term MN and its shape $\mathrm{Sh}(N)$ can occur many times in the type $B[\mathrm{Sh}(N)]$.

(ii) In the typing rule for $\lambda x.M$, we do not allow abstracting over a variable of linear type with index 0, because this violates linearity. This differs from the formulation used by McBride and by Atkey, where abstracting over zero uses of a resource is allowed.

(iii) In the typing rule for $\mathsf{lift}\, M$, we require the context to be a parameter context Φ. Although the term M may not be a parameter term, the term $\mathsf{lift}\, M$ is considered a parameter term because it has an interpretation in $p\mathbf{B}$.

(iv) In the typing rule for (M, N), the term M also appears as a parameter term $\mathrm{Sh}(M)$ in the type B. All linear variables of M will have index 0 in the context Γ_2 and index 1 in Γ_1, so M is considered used exactly once in the term (M, N).

(v) We identify a subset of typing rules $\Phi \vdash R : P$ for typing parameter terms. We do not formally separate the typing rules into two fragments. In the semantics, the interpretation of $\Phi \vdash R : P$ is an arrow in $p\mathbf{B}$.

(vi) The typing rules for the parameter terms $R_1 @ R_2$ and $\lambda' x.R$ are the usual ones from intuitionistic dependent types, where the contexts are required to be parameter contexts. The parameter term $\mathsf{force}'\, R$ has a type of the form $\mathrm{Sh}(A)$.

(vii) We implicitly assume all contexts appearing in typing rules to be well-formed (i.e., if it is not well-formed, the rule does not apply). In particular, this applies to contexts of the form $\Gamma_1 + \Gamma_2$, which carry the side condition that a type with index ω must be a parameter type.

We have the following standard syntactic results: a well-formed typing judgment implies that the type and the context are well-formed, and the shape operation preserves typing.

Theorem 3.6.

- *If* $\Gamma \vdash M : A$, *then* $\mathrm{Sh}(\Gamma) \vdash A : *$ *and* $\Gamma \vdash$.
- *If* $\Gamma \vdash M : A$, *then* $\mathrm{Sh}(\Gamma) \vdash \mathrm{Sh}(M) : \mathrm{Sh}(A)$.

Since parameter terms have no side effects (i.e., do not affect states), we can freely substitute them into a term or a type. We cannot freely substitute a general term because side effects may get duplicated after the substitution. Since a value can be of a linear type, when substituting it into a type, we must apply the shape operation to the value. To summarize, we have:

Theorem 3.7 (Substitution)**.**

- *If* $\Phi, x : P \vdash B[x] : *$ *and* $\Phi \vdash R : P$, *then* $\Phi \vdash B[R] : *$.
- *If* $\Phi, x : P, \Gamma \vdash M : B[x]$ *and* $\Phi \vdash R : P$, *then* $\Phi, [R/x]\Gamma \vdash [R/x]M : B[R]$.
- *If* $\Gamma_1, x :_k A, \Gamma' \vdash M : B[x]$ *and* $\Gamma_2 \vdash V : A$, *then* $\Gamma_1 + k\Gamma_2, [\mathrm{Sh}(V)/x]\Gamma' \vdash [V/x]M : B[\mathrm{Sh}(V)]$.

3.2 Operational semantics

We adopt a big-step call-by-value operational semantics for our type system. A term in our language may in principle modify the state. However, since we have not yet introduced any primitive operations that can actually update the state, the state has been omitted from the statement of the following evaluation rules. We will return to the state change in Section 4.2, when we discuss a concrete language with specific primitive operations.

Definition 3.8 (Selected evaluation rules). $\boxed{M \Downarrow N}$

$$\frac{M \Downarrow \lambda x.M' \quad N \Downarrow V \quad [V/x]M' \Downarrow N'}{MN \Downarrow N'} \qquad \frac{M \Downarrow \mathsf{lift}\, M' \quad M' \Downarrow N}{\mathsf{force}\, M \Downarrow N}$$

$$\frac{R_1 \Downarrow \lambda' x.R_1' \quad R_2 \Downarrow V \quad [V/x]R_1' \Downarrow R'}{R_1 @ R_2 \Downarrow R'} \qquad \frac{R \Downarrow \mathsf{lift}\, M \quad \mathrm{Sh}(M) \Downarrow R'}{\mathsf{force}'\, R \Downarrow R'}$$

Note that a term of the form $\mathsf{lift}\, M$ is a parameter term, even when M is not. So to ensure that the resulting parameter term $\mathsf{force}'(\mathsf{lift}\, M)$ does not modify state, we must evaluate it to $\mathrm{Sh}(M)$, which is again a parameter term. This motivates the evaluation rule $\mathsf{force}'\, R$.

Since the evaluation rules are also used during type checking, the evaluation is defined on open terms. Thus it may not always evaluate a term to a value. We treat variables as values to facilitate type-level evaluation.

We have the following type conversion rule for equality of types.

Definition 3.9 (Type conversion)**.**

$$\frac{\Gamma \vdash M : A[R] \quad R \Downarrow R'}{\Gamma \vdash M : A[R']}$$

The type preservation for the big-step evaluation can be proved by induction on the evaluation rules. The main insight that is needed to prove type preservation is that only values and parameter terms can be substituted into another term.

Theorem 3.10 (Type preservation)**.** *If* $\Gamma \vdash M : A$ *and* $M \Downarrow M'$, *then we have* $\Gamma \vdash M' : A$.

3.3 Denotational semantics

We interpret our type system with a state-parameter fibration, by providing contexts $\Gamma \vdash$, dependent types $\Gamma \vdash A : *$, and terms $\Gamma \vdash M : A$ with interpretations $[\![-]\!]$ in the manner described in Section 2.4, and by interpreting the shape operation by the shape-unit functor $p \circ \sharp$ as in Section 2.5. In addition, the substitution of a term into a type corresponds to a pullback in the standard fashion. This semantics is encapsulated in the following theorem (and its proof). Recall that $\underline{A}$ is a notation for $\sharp A$. We often write $[\![\Gamma]\!]$ instead of $[\![\Gamma \vdash]\!]$.

Theorem 3.11. *An interpretation* $[\![-]\!]$ *can be defined in such a way that*

(10) $[\![\Gamma]\!]$ *is an object of* $\mathbf{E}$.
(11) $[\![\Phi \vdash A : *]\!]$ *is an object of* $\mathbf{E}/\underline{[\![\Phi]\!]}$. *We may write* $\pi : [\![A]\!] \to \underline{[\![\Phi]\!]}$ *for it.*
(12) $[\![\Gamma \vdash M : A]\!]$ *is an arrow of* $\mathbf{E}/\underline{[\![\Gamma]\!]}$ *from* $\eta_{[\![\Gamma]\!]}$ *to* $[\![\Gamma \vdash A : *]\!]$.

(13) $\llbracket \mathrm{Sh}(\Gamma) \rrbracket = p\llbracket \Gamma \rrbracket$, *an object of* $p\mathbf{B}$.

(14) $\llbracket \Phi \vdash \mathrm{Sh}(A) : * \rrbracket = p\underline{\llbracket \Phi \vdash A : * \rrbracket}$, *an object of* $p\mathbf{B}/\underline{\llbracket \Phi \rrbracket}$.

(15) $\llbracket \mathrm{Sh}(\Gamma) \vdash \mathrm{Sh}(M) : \mathrm{Sh}(A) \rrbracket = p\underline{\llbracket \Gamma \vdash M : A \rrbracket}$, *an arrow of* $p\mathbf{B}/\underline{\llbracket \Gamma \rrbracket}$.

(16) *Suppose* $\pi = \llbracket \Phi, x : P \vdash B : * \rrbracket : \llbracket B \rrbracket \to \underline{\llbracket \Phi, x : P \rrbracket}$ *and* $\llbracket R \rrbracket = \llbracket \Phi \vdash R : P \rrbracket : \llbracket \Phi \rrbracket \to \llbracket P \rrbracket$ *in* $p\mathbf{B}/\underline{\llbracket \Phi \rrbracket}$. *Then* $\pi' = \llbracket \Phi \vdash [R/x]B : * \rrbracket : \llbracket [R/x]B \rrbracket \to \underline{\llbracket \Phi \rrbracket}$ *is the pullback of* π *along* $\underline{\llbracket R \rrbracket}$ *as in*

$$\begin{tikzcd}
\llbracket [R/x]B \rrbracket \arrow[r] \arrow[d, "\pi'"'] \arrow[dr, phantom, "\lrcorner", very near start] & \llbracket B \rrbracket \arrow[d, "\pi"] \\
\underline{\llbracket \Phi \rrbracket} \arrow[r, "\underline{\llbracket R \rrbracket}"] & \underline{\llbracket \Phi, x : P \rrbracket} = \underline{\llbracket P \rrbracket}
\end{tikzcd}$$

The proof of this theorem goes by simultaneous induction on the derivation, with each case of the inductive step showing how exactly the semantics works. Here we show some examples of the cases (for the other cases please see [8]).

(i) The first is

$$\frac{\Gamma \vdash \quad \mathrm{Sh}(\Gamma) \vdash A : *}{\Gamma, x :_k A \vdash} \quad k = \omega \text{ only if } A \text{ is a parameter type}$$

By the assumptions, we have objects $\llbracket \Gamma \rrbracket$ of $\mathbf{E}$ and $\pi : \llbracket A \rrbracket \to \underline{\llbracket \mathrm{Sh}(\Gamma) \rrbracket} = \underline{\llbracket \Gamma \rrbracket}$ of $\mathbf{E}/\underline{\llbracket \Gamma \rrbracket}$. We therefore define

$$\llbracket \Gamma, x :_k A \rrbracket = \llbracket \Gamma \rrbracket \overleftarrow{\otimes} \llbracket A \rrbracket^k$$

using a dependent monoidal product, where $\llbracket A \rrbracket^k = p\underline{\llbracket A \rrbracket}$ if $k = 0$ and $\llbracket A \rrbracket^k = \llbracket A \rrbracket$ otherwise.

(ii) The second case is

$$\frac{\Phi, x :_k \mathrm{Sh}(A) \vdash B[x] : *}{\Phi \vdash (x : A) \multimap B[x] : *}$$

The assumption implies $\Phi \vdash A : *$. Hence we have arrows $\pi : \llbracket B \rrbracket \to \underline{\llbracket \Phi, x :_k \mathrm{Sh}(A) \rrbracket} = \underline{\llbracket A \rrbracket}$ and $\llbracket A \rrbracket \to \underline{\llbracket \Phi \rrbracket}$. We therefore define

$$\llbracket \Phi \vdash (x : A) \multimap B[x] : * \rrbracket = \Pi_{\underline{\llbracket \Phi \rrbracket}, \llbracket A \rrbracket} \llbracket B \rrbracket,$$

using a dependent function space that gives an object in $\mathbf{E}/\underline{\llbracket \Phi \rrbracket}$.

(iii) The third case is

$$\frac{0\Gamma, x :_1 A, 0\Gamma' \vdash}{0\Gamma, x :_1 A, 0\Gamma' \vdash x : A}$$

Since $\mathrm{Sh}(\Gamma) \vdash A : *$, by (11) we have $\pi : \llbracket A \rrbracket \to \underline{\llbracket \Gamma \rrbracket}$. Note that there is a canonical morphism $f : p\underline{\llbracket \Gamma \rrbracket} \overleftarrow{\otimes} \llbracket A \rrbracket \overleftarrow{\otimes} p\underline{\llbracket \Gamma' \rrbracket} \to \llbracket A \rrbracket$ over $\underline{\llbracket \Gamma \rrbracket}$. We define $\llbracket x \rrbracket$ as the following u over $\underline{\llbracket \Gamma, x : A, \Gamma' \rrbracket}$.

$$\begin{tikzcd}
p\underline{\llbracket \Gamma \rrbracket} \overleftarrow{\otimes} \llbracket A \rrbracket \overleftarrow{\otimes} p\underline{\llbracket \Gamma' \rrbracket} \arrow[d, dashed, "u"] \arrow[drr, "f"] \arrow[dd, bend right=60, "\eta"'] & & \\
\underline{\llbracket \Gamma' \rrbracket} \times_{\underline{\llbracket \Gamma \rrbracket}} \llbracket A \rrbracket \arrow[rr] \arrow[d, "\pi'"] \arrow[drr, phantom, "\lrcorner", very near start] & & \llbracket A \rrbracket \arrow[d, "\pi"] \\
\underline{\llbracket \Gamma, x : A, \Gamma' \rrbracket} = \underline{\llbracket \Gamma' \rrbracket} \arrow[rr] & & \underline{\llbracket \Gamma \rrbracket}
\end{tikzcd}$$

(iv) The fourth case is

$$\frac{\Gamma, x :_k A \vdash M : B[x] \quad k = 0 \Rightarrow A = P}{\Gamma \vdash \lambda x.M : (x : A) \multimap B[x]}$$

By the assumptions, we have a morphism $\llbracket M \rrbracket : \llbracket \Gamma \rrbracket \otimes_{\underline{\llbracket \Gamma \rrbracket}} \llbracket A \rrbracket^k \to \llbracket B \rrbracket$ over $\underline{\llbracket \Gamma, x : A \rrbracket}$. By the adjunction in Theorem 2.14, we define $\llbracket \lambda x.M \rrbracket = \widetilde{\llbracket M \rrbracket} : \llbracket \Gamma \rrbracket \to \prod_{\underline{\llbracket \Gamma \rrbracket}, \llbracket A \rrbracket^k} \llbracket B \rrbracket$ as a morphism over $\underline{\llbracket \Gamma \rrbracket}$.

(v) The last case we review here is

$$\frac{\Gamma_1 \vdash M : (x : A) \multimap B[x] \quad \Gamma_2 \vdash N : A}{\Gamma_1 + \Gamma_2 \vdash MN : B[\mathrm{Sh}(N)]}$$

The assumptions give arrows $\llbracket M \rrbracket : \llbracket \Gamma_1 \rrbracket \to \Pi_{\underline{\llbracket \Gamma \rrbracket}, \llbracket A \rrbracket} \llbracket B \rrbracket$ and $\llbracket N \rrbracket : \llbracket \Gamma_2 \rrbracket \to \llbracket A \rrbracket$ of $\mathbf{E}/\underline{\llbracket \Gamma \rrbracket}$ for $\underline{\llbracket \Gamma \rrbracket} = \underline{\llbracket \Gamma_1 \rrbracket} = \underline{\llbracket \Gamma_2 \rrbracket}$, as well as $\pi : \llbracket B \rrbracket \to \underline{\llbracket \mathrm{Sh}(\Gamma), x :_k \mathrm{Sh}(A) \rrbracket} = \underline{\llbracket A \rrbracket}$. Note that $\underline{\llbracket \mathrm{Sh}(N) \rrbracket} = \underline{\llbracket N \rrbracket}$, since (15) implies $\llbracket \mathrm{Sh}(N) \rrbracket = p\underline{\llbracket N \rrbracket}$. Hence (16) gives the pullback square in the following. We therefore define $\llbracket MN \rrbracket$ as the unique arrow u that makes the diagram commute.

$$\begin{tikzcd}
\llbracket \Gamma_1 + \Gamma_2 \rrbracket \arrow[rr, "\llbracket M \rrbracket \otimes_{\underline{\llbracket \Gamma \rrbracket}} \llbracket N \rrbracket"] \arrow[dr, dotted, "u"'] \arrow[ddr, bend right, "\eta"'] & & (\Pi_{\underline{\llbracket \Gamma \rrbracket}, \llbracket A \rrbracket} \llbracket B \rrbracket) \otimes_{\underline{\llbracket \Gamma \rrbracket}} \llbracket A \rrbracket \arrow[dr, "\epsilon"] & \\
& \llbracket B[\mathrm{Sh}(N)] \rrbracket \arrow[rr] \arrow[d, "\pi"] \arrow[drr, phantom, "\lrcorner", very near start] & & \llbracket B \rrbracket \arrow[d] \\
& \llbracket \Gamma \rrbracket \arrow[rr, "\underline{\llbracket N \rrbracket}"'] & & \underline{\llbracket A \rrbracket}
\end{tikzcd}$$

It is crucial to observe that $\llbracket \Gamma_1 + \Gamma_2 \rrbracket = \llbracket \Gamma_1 \rrbracket \overleftarrow{\otimes} \llbracket \Gamma_2 \rrbracket$. This then enables us to use $\epsilon : (\Pi_{\underline{\llbracket \Gamma \rrbracket}, \llbracket A \rrbracket} \llbracket B \rrbracket) \otimes_{\underline{\llbracket \Gamma \rrbracket}} \llbracket A \rrbracket \to \llbracket B \rrbracket$, the counit of the adjunction of Theorem 2.14.

For this semantics, we have the following soundness theorem. The main step in proving this theorem is proving the semantic version of the substitution theorem (Theorem 3.7).

Theorem 3.12 (Soundness)**.** *If* $\Gamma \vdash M : A$ *and* $M \Downarrow M'$, *then we have* $\llbracket M \rrbracket = \llbracket M' \rrbracket$.

4 Dependently typed Proto-Quipper

The type system we defined in Section 3 is abstract. For example, there is no mention of quantum data types, concrete representation of quantum circuits, circuit boxing and unboxing. This is because the language is derived directly from the general semantics of Section 2.

In this section we show how to support programming quantum circuits. On the semantic side, this means that we will work with a concrete state-parameter fibration, i.e., $\sharp : \bar{\bar{\mathbf{M}}} \to \mathbf{Set}$. On the syntactic side, this means that we will extend the type system of Section 3 with several constructs to work with quantum circuits. We call the resulting type system "dependently typed Proto-Quipper", or sometimes Proto-Quipper-D.[1]

4.1 Simple types and quantum circuits

Recall from Section 2.3 that the model $\bar{\bar{\mathbf{M}}}$ of [28] contains a (full) monoidal subcategory $\mathbf{M}$ of (generalized) circuits. Objects S of $\mathbf{M}$ (or $(1, S)$ of $\bar{\bar{\mathbf{M}}}$) are called *simple*. It follows that S is simple iff $\sharp S = 1$. Such objects represent states without parameters (i.e., a single state space, rather than an indexed family of them). We call a type simple if it is interpreted by a simple object. For any simple objects S_1 and S_2, we have the isomorphism

$$!(S_1 \multimap S_2) \overset{\text{box/unbox}}{\cong} p\mathbf{M}(S_1, S_2).$$

This isomorphism means that the function type $!(S_1 \multimap S_2)$ literally represents a hom-set of the category $\mathbf{M}$, which we can think of as a set of circuits with input S_1 and output S_2. We equip the programming language with a type $\mathbf{Circ}(S_1, S_2)$ representing such circuits as first-class citizens. These linear functions of simple types correspond to quantum circuits.

The type $\mathbf{Qubit}$ is defined as $(1, \mathbf{Q})$, where $\mathbf{Q}$ is a designated object in $\mathbf{M}$ representing the qubit type. As a result of being a free coproduct completion, $\bar{\bar{\mathbf{M}}}$ has coproducts. Thus Proto-Quipper-M admits quantum data types. For example, the type $\mathbf{List\,Qubit}$ can be interpreted as an object $\Sigma_{n\in\mathbb{N}}(1, \mathbf{Q}^{\otimes n}) = (\mathbb{N}, (\mathbf{Q}^{\otimes n})_{n\in\mathbb{N}})$.

A problem of quantum data types. The box/unbox isomorphism only holds for simple types. However, $\mathbf{List\,Qubit}$ is not a simple type since $\sharp[\![\mathbf{List\,Qubit}]\!] = \sharp(\mathbb{N}, (\mathbf{Q}^{\otimes n})_{n\in\mathbb{N}}) = \mathbb{N}$. Thus, there is no such type as $\mathbf{Circ}(\mathbf{List\,Qubit}, \mathbf{List\,Qubit})$ in Proto-Quipper-M, and a function of type $!(\mathbf{List\,Qubit} \multimap \mathbf{List\,Qubit})$ cannot be converted into a circuit. This makes sense, because such a function actually represents a *family* of circuits, indexed by the length of the input list. In practice, however, we want programmers to be able to work with quantum data types and linear functions between them. The above limitation makes boxing circuits in Proto-Quipper-M awkward.

Solution via dependent quantum data types. We solve this problem by working with *dependent quantum data types*, such as $\mathbf{Vec\,Qubit}\,n$. For a value n of type $\mathbf{Nat}$, we define $[\![\mathbf{Vec\,Qubit}\,n]\!]$ as $V_n = (1, \mathbf{Q}^{\otimes n})$ as in the following pullback square. Note that the existence of the pullback is guaranteed by the state-parameter fibration $\sharp : \bar{\bar{\mathbf{M}}} \to \mathbf{Set}$.

$$\begin{tikzcd}
{[\![\mathbf{Vec\,Qubit}\,n]\!] = V_n} \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & {[\![\mathbf{List\,Qubit}]\!] = \Sigma_{n\in\mathbb{N}}(1, \mathbf{Q}^{\otimes n})} \arrow[d] \\
q1 \arrow[r, "qn"] & q\mathbb{N}
\end{tikzcd}$$

Thus $\sharp[\![\mathbf{Vec\,Qubit}\,n]\!] = \sharp V_n = 1$ and $\mathbf{Vec\,Qubit}\,n$ is a simple type. Now a linear function of the type $!(\mathbf{Vec\,Qubit}\,n \multimap \mathbf{Vec\,Qubit}\,n)$ can be boxed into a circuit, which has the type $\mathbf{Circ}(\mathbf{Vec\,Qubit}\,n, \mathbf{Vec\,Qubit}\,n)$. Together with the dependent function type, the programmer can now define a function of the type

$$!((n : \mathbf{Nat}) \multimap \mathbf{Circ}(\mathbf{Vec\,Qubit}\,n, \mathbf{Vec\,Qubit}\,n)).$$

Such a function represents a family of circuits indexed by n.

We extend the syntax of Definition 3.1 with simple types, circuit types, dependent data types, and the boxing and unboxing operations for circuits.

Definition 4.1 (Extended syntax)**.**

Types A, B ::= ... | $\mathbf{Nat}$ | $\mathbf{List}\,A$ | $\mathbf{Vec}\,A\,R$ | $\mathbf{Circ}(S_1, S_2)$
Simple Types S ::= $\mathbf{Qubit}$ | $\mathbf{Unit}$ | $S_1 \otimes S_2$ | $\mathbf{Vec}\,S\,R$
Parameter types P ::= ... | $\mathbf{Nat}$ | $\mathbf{List}\,P$ | $\mathbf{Vec}\,P\,R$ | $\mathbf{Circ}(S_1, S_2)$
Terms M ::= ... | ℓ | (a, C, b) | $\mathsf{box}_S\,M$ | $\mathsf{unbox}\,M$

Note that the vector data type $\mathbf{Vec}\,A\,R$ requires the length index to be a parameter term R, as only parameter terms can appear in types. The type $\mathbf{Circ}(S_1, S_2)$ is considered a parameter type because $[\![\mathbf{Circ}(S_1, S_2)]\!] = p\mathbf{M}([\![S_1]\!], [\![S_2]\!])$, which is a parameter object.

Following Proto-Quipper-M, we extend terms with labels ℓ, which are distinct from variables. Labels are used to represent wires of a quantum circuit (or more generally, components of a tensor product). Unlike variables, labels cannot be substituted. We also extend contexts with label contexts of the form $\Sigma = \ell_1 :_{k_1} \mathbf{Qubit}, ..., \ell_n :_{k_n} \mathbf{Qubit}$, where $k_i \in \{0, 1\}$. The semantics of Σ is a tensor product of qubits, and we identify each Σ with the corresponding object in $\mathbf{M}$.

The canonical inhabitants of $\mathbf{Circ}(S_1, S_2)$ are *boxed circuits* of the form (a, C, b), where C is a morphism of $\mathbf{M}$ and a, b are interfaces connecting the inputs and outputs of C to the types S_1 and S_2, respectively (see [28] for more details).

The typing rules for $\mathbf{Circ}(S_1, S_2)$ are the same as in Proto-Quipper-M:

$$\frac{\Sigma_1 \vdash a : S_1 \quad \Sigma_2 \vdash b : S_2 \quad C : \Sigma_1 \to \Sigma_2}{\Phi \vdash (a, C, b) : \mathbf{Circ}(S_1, S_2)}$$

$$\frac{\Gamma \vdash M : !(S_1 \multimap S_2)}{\Gamma \vdash \mathsf{box}_{S_1}\,M : \mathbf{Circ}(S_1, S_2)} \qquad \frac{\Gamma \vdash M : \mathbf{Circ}(S_1, S_2)}{\Gamma \vdash \mathsf{unbox}\,M : !(S_1 \multimap S_2)}$$

Boxed circuits are not part of the surface language; rather they are built and consumed by the box and unbox operations. Certain built-in boxed circuits, called *gates*, may be bound to constant symbols of the language (or provided by a standard library). An implementation may also provide

[1] A full specification of dependently typed Proto-Quipper is available in [8].

additional primitive operations on boxed circuits, such as reverse : $\mathbf{Circ}(S_1, S_2) \to \mathbf{Circ}(S_2, S_1)$. Indeed, although the types $!(S_1 \multimap S_2)$ and $\mathbf{Circ}(S_1, S_2)$ are semantically isomorphic, they are operationally distinct, because $\mathbf{Circ}(S_1, S_2)$ has canonical inhabitants whereas $!(S_1 \multimap S_2)$ does not.

The shape operation can be extended naturally.

Definition 4.2 (Extended shape operation)**.**

$$\begin{aligned}
&\mathrm{Sh}(\mathbf{Nat}) = \mathbf{Nat}\\
&\mathrm{Sh}(\mathbf{List}\,A) = \mathbf{List}\,\mathrm{Sh}(A)\\
&\mathrm{Sh}(\mathbf{Vec}\,A\,R) = \mathbf{Vec}\,\mathrm{Sh}(A)\,R\\
&\mathrm{Sh}(\mathbf{Circ}(S_1, S_2)) = \mathbf{Circ}(S_1, S_2)\\
&\mathrm{Sh}(\ell) = \mathsf{unit}\\
&\mathrm{Sh}(a, C, b) = (a, C, b)\\
&\mathrm{Sh}(\mathsf{box}_S\, M) = \mathsf{box}_S\, \mathrm{Sh}(M)\\
&\mathrm{Sh}(\mathsf{unbox}\, M) = \mathsf{unbox}\, \mathrm{Sh}(M)
\end{aligned}$$

This definition is semantically sound for **List** and **Vec** because the shape-unit functor $p\sharp$ preserves tensor products and coproducts in $\bar{\bar{\mathbf{M}}}$. We also observe that $\mathrm{Sh}(\mathbf{List}\,\mathbf{Qubit}) = \mathbf{List}\,\mathbf{Unit} \cong \mathbf{Nat}$ and $\mathrm{Sh}(\mathbf{Vec}\,\mathbf{Qubit}\,R) = \mathbf{Vec}\,\mathbf{Unit}\,R \cong \mathbf{Unit}$.

Safe list-to-vector conversion. Allowing types to depend on the *shape* of any other types allows us to define a function to *safely* convert a list of qubits into a vector of qubits.[2]

$$\begin{aligned}
&\mathsf{conv} : !((x : \mathbf{List}\,\mathbf{Qubit}) \multimap \mathbf{Vec}\,\mathbf{Qubit}\,(\mathsf{toNat}\,x))\\
&\mathsf{conv} =\\
&\quad \mathsf{lift}\,(\lambda x.\,\mathsf{case}\,x\,\mathsf{of}\\
&\qquad\qquad \mathsf{Nil} \to \mathsf{VNil}\\
&\qquad\qquad \mathsf{Cons}\,y\,ys \to \mathsf{VCons}\,y\,(\mathsf{force}\,\mathsf{conv}\,ys))
\end{aligned}$$

Let us consider the type

$$(x : \mathbf{List}\,\mathbf{Qubit}) \multimap \mathbf{Vec}\,\mathbf{Qubit}\,(\mathsf{toNat}\,x).$$

Here $\mathsf{toNat} : \mathbf{List}\,\mathbf{Unit} \to \mathbf{Nat}$ is the isomorphism that converts a list of units to a natural number. Using the kinding rules of Definition 3.4 to kind check this type, we only need to kind check the following

$$x : \mathrm{Sh}(\mathbf{List}\,\mathbf{Qubit}) = \mathbf{List}\,\mathbf{Unit} \vdash \mathbf{Vec}\,\mathbf{Qubit}\,(\mathsf{toNat}\,x) : *,$$

which is a valid kinding judgment. So a function of the type $(x : \mathbf{List}\,\mathbf{Qubit}) \multimap \mathbf{Vec}\,\mathbf{Qubit}\,(\mathsf{toNat}\,x)$ takes a list of qubits as input and outputs a vector of qubits, where the length of the vector is the length of the input list.

The above conversion would not be possible if we required dependent types to be only of the form $(x : P) \to B[x]$, where P is a parameter type. McBride's conversion function has a different flavor. In [19, Section 5], he defines $\mathsf{conv} :_\omega (x :_1 \mathbf{List}\,X) \multimap \mathbf{Vec}\,X(\mathsf{length}\,x)$, where $\mathsf{length} :_0 (x :_0 \mathbf{List}\,X) \multimap \mathbf{Nat}$. We cannot define a length function of type $\mathbf{List}\,\mathbf{Qubit} \multimap \mathbf{Nat}$ because it violates linearity.

[2]A dependently typed Proto-Quipper program for conv can be found in [8].

4.2 Operational semantics

In Proto-Quipper-M [28], the call-by-value big-step evaluation is defined on a *configuration* of the form (C, M), where C denotes a morphism of $\mathbf{M}$, and M is a term that can append quantum gates to C when evaluated.

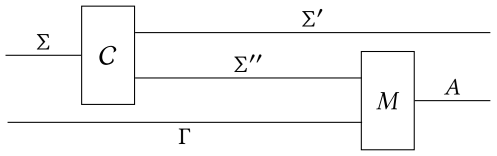
Definition 4.3. We define a *well-typed configuration*

$$\Gamma; \Sigma \vdash (C, M) : A; \Sigma'$$

to mean there exists Σ'' such that $C : \Sigma \to \Sigma'' \otimes \Sigma'$ and $\Gamma, \Sigma'' \vdash M : A$.

Our definition of well-typed configurations allows M to be an open term, with free variables from the context Γ. The reason for this is that the evaluation rules can be used during type checking, where there can be free variables in types.

We can visualize the configuration (C, M) in the above definition as the following diagram, where the term M in (C, M) is using the output wires from Σ''.



Dependently typed Proto-Quipper has the same evaluation rules as Proto-Quipper-M, plus rules for reducing parameter terms.[3]

Definition 4.4 (Selected evaluation rules)**.**

$$\frac{\begin{array}{l}(C_1, M) \Downarrow (C_2, \mathsf{force}(\mathsf{unbox}\,(a, \mathcal{D}, b)))\\ (C_2, N) \Downarrow (C_3, a')\\ C_4 = \mathrm{append}(C_3, a', (a, \mathcal{D}, b)), \mathrm{FV}(MN) = \emptyset\end{array}}{(C_1, MN) \Downarrow (C_4, b)}$$

$$\frac{\begin{array}{ll}(C, M) \Downarrow (C', \mathsf{lift}\ M') & (\mathrm{id}_I, R) \Downarrow (\_, V)\\ (\mathrm{id}_a, M'\,a) \Downarrow (\mathcal{D}, b) & \\ a = \mathrm{gen}(S[V]) & \mathrm{FV}(\mathsf{box}_{S[R]}\,M) = \emptyset\end{array}}{(C, \mathsf{box}_{S[R]}\,M) \Downarrow (C', (a, \mathcal{D}, b))}$$

$$\frac{\begin{array}{ll}(C, M) \Downarrow (C', \lambda x.M') & (C', N) \Downarrow (C'', V)\\ (C'', [V/x]M') \Downarrow (C''', N') & \end{array}}{(C, MN) \Downarrow (C''', N')}$$

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

$$\frac{(C, M) \Downarrow (C', \mathsf{unbox}\,M')}{(C, \mathsf{box}_S\,M) \Downarrow (C', M')}$$

[3]A detailed definition of the evaluation rules is in [8].

$$\frac{\begin{array}{l}(C_1, R_1) \Downarrow (C_2, \lambda' x.R_1') \qquad (C_2, R_2) \Downarrow (C_3, V) \\ (C_3, [V/x]R_1') \Downarrow (C_4, R')\end{array}}{(C_1, R_1 @ R_2) \Downarrow (C_4, R')}$$

$$\frac{(C_1, R) \Downarrow (C_2, \mathsf{lift}\, M) \quad (C_1, \mathsf{Sh}(M)) \Downarrow (C_3, R')}{(C_1, \mathsf{force}'\, R) \Downarrow (C_3, R')}$$

In the first evaluation rule, the term MN appends a circuit $\mathcal{D}$ to the circuit C_3, resulting in an updated circuit C_4. The term N evaluates to a label tuple a', pointing to some of the outputs of the then-current circuit C_3. This tuple a' must match the input interface a of the boxed circuit $(a, \mathcal{D}, b)$. Finally, the operation append$(C_3, a', (a, \mathcal{D}, b))$ constructs a new circuit C_4 by connecting the outputs a' of C_3 to the inputs a of $(a, \mathcal{D}, b)$. As a result, C_4 will expose b as part of its output interface.

The second evaluation rule is for boxing a circuit. The notation $S[R]$ denotes a simple type S that may contain a parameter term R. The type annotation $S[R]$ on the keyword box is used to generate a data structure $a = \mathrm{gen}(S[V])$ holding fresh wire labels. The evaluator then evaluates $M'\ a$ under the identity circuit id_a, which produces a circuit $\mathcal{D}$ and a circuit output b.

The third to fifth evaluation rules are the same as Proto-Quipper-M. The last two rules are for evaluating parameter terms. Since parameter terms do not change states, the evaluation of these terms does not append any quantum gates. We have the following theorem.

Theorem 4.5. *If $(C, R) \Downarrow (C', R')$, then we have $C = C'$.*

The semantics of well-typed configurations is given below. It corresponds to the above diagram for (C, M), where vertical composition is interpreted as the fibered monoidal product $\otimes_{\underline{[\![\Gamma]\!]}}$.

Definition 4.6. Let $\Gamma; \Sigma \vdash (C, M) : A; \Sigma'$ be a well-typed configuration. We define

$$\begin{aligned}[\![(C, M)]\!] &: [\![\Gamma]\!] \overset{\leftarrow}{\otimes} [\![\Sigma]\!] \to [\![A]\!] \overset{\leftarrow}{\otimes} [\![\Sigma']\!] \\ &= ([\![M]\!] \overset{\leftarrow}{\otimes} [\![\Sigma']\!]) \circ ([\![\Gamma]\!] \overset{\leftarrow}{\otimes} C),\end{aligned}$$

a morphism over $\underline{[\![\Gamma]\!]}$ in $\bar{\bar{\mathbf{M}}}/\underline{[\![\Gamma]\!]}$.

Dependently typed Proto-Quipper satisfies type preservation and soundness.

Theorem 4.7 (Type preservation and soundness)**.** *If $\Gamma; \Sigma \vdash (C, M) : A; \Sigma'$ and $(C, M) \Downarrow (C', M')$, then we have $\Gamma; \Sigma \vdash (C', M') : A; \Sigma'$ and $[\![(C, M)]\!] = [\![(C', M')]\!]$.*

4.3 Prototype implementation

We have implemented a prototype for dependently typed Proto-Quipper, called Proto-Quipper-D. We will now discuss some essential features that enable the type system in this paper to scale to a usable programming language. For more information about programming in Proto-Quipper-D, please see [8] and the tutorial introduction [7].

Type inference and elaboration. It is a well-known issue that programming with linear types can be quite *invasive*: the programmer must supply linearity annotations, which is burdensome for large programs. Recent research tries to address this problem. For example, Paykin and Zdancewic [25] propose embedding a linear-nonlinear type system in Haskell. Bernardy et al. [5] propose extending Haskell with linear types.

In our implementation, every top-level declaration must be of parameter type (since top-level functions are reusable). Programs can be written without using annotations such as lift and force. We implemented a type elaborator that extends the well-known bi-directional type inference algorithm [26] with the ability to insert linearity annotations. The basic idea is that when *checking* a program M with type $!A$, the elaborator produces a new term lift M', where M' is the result of checking M against the type A. When *inferring* a type for the application MN, where M has the type $!(A \multimap B)$, the elaborator will produce a new term $(\mathsf{force}\, M)\ N'$ and a type B, where N' is the result of *checking* N against A. We also implemented a secondary checker to recheck the annotated terms produced by the elaborator. We have found that this two-step elaboration-and-checking works well in practice and linear annotations are manageable with the help of the type elaborator.

Simple data types and parameter data types. We implement Haskell 98 style [15] data types (which begin with the keyword `data`). For example, list data type can be declared as `data List a = Nil | Cons a (List a)`. We also implemented a special kind of dependent data types that we call simple data type declarations (which begin with the keyword `simple`). For example, the following is a declaration of a vector data type.

```
simple Vec a : Nat -> Type where
   Vec a Z = VNil
   Vec a (S n) = VCons a (Vec a n)
```

As we mentioned in Section 4.1, only linear functions of simple types can be boxed into circuits and simple types can be characterized semantically by $\sharp S = 1$. Simple types can also be described syntactically as types that uniquely determine the size of their inhabitants. For example, **List** is not a simple type because **List Qubit** may have inhabitants of different sizes, while inhabitants of **Vec Qubit** 3 must have size 3, so it is a simple type. In Proto-Quipper-D, the simple data type declaration is checked by the compiler using a syntactic characterization of simple types. Note that such a simplicity check is undecidable in general, as it is equivalent to checking termination of a recursive function (e.g., the vector data type declaration above can be viewed as a primitive recursive function which recurs on a second input of type

`Nat`). We implemented a simplicity checker that only accepts certain declarations that correspond to primitive recursive functions.

Since we allow user-defined types, the notion of parameter types and simple types must account for such extensions. We implement parameter types and simple types as type classes [35], and their instances are automatically generated by the compiler upon defining a data type.

For example, the list data type declaration will generate an instance `(Parameter a) => Parameter (List a)`, which means that `List a` is a parameter type if the type `a` is a parameter type. Similarly, the vector data type declaration will generate an instance `(Simple a) => Simple (Vec a n)`, which means `Vec a n` is a simple type if the type `a` is simple.

With simple type constraints, we can give the circuit boxing operator box the following type `(Simple a, Simple b) => !(a -> b) -> Circ(a, b)`. This way the type checker will be able check at compile time whether the linear functions we are boxing are indeed linear functions of simple types. With parameter type constraints, we can safely discard reusable variables without violating linearity. For example, we can define a function `fst` to retrieve the left element of a pair, which will have type `!((Parameter b) => a * b -> a)`. Here `a * b` means $a \otimes b$. Since type `b` is a parameter type, it is safe to discard the right element.

Irrelevant quantification. Besides the linear dependent types described in this paper, we also implemented a version of Miquel's irrelevant quantification [3, 20]. Although the usual dependent quantification is enough in theory, it is beneficial to implement irrelevant quantification. Because irrelevant arguments are erased during evaluation, length-indexed data types such as **Vec** will not store their length at runtime, hence making the evaluator more efficient.

In Proto-Quipper-D, a dependent quantification takes the form `(n : Nat) -> T`, and an irrelevant quantification takes the form `forall (n : Nat) -> T`. We implement the following typing rules for irrelevant quantification.

$$\frac{\Phi, x : P, \Gamma \vdash M : B[x]}{\Phi, \Gamma \vdash \lambda\{x\}.M : \forall(x : P).B[x]}$$

$$\frac{\Phi, \Gamma \vdash M : \forall(x : P).B[x] \quad \Phi \vdash R : A}{\Phi, \Gamma \vdash M\{R\} : B[R]}$$

Both irrelevant abstraction $\lambda\{x\}.M$ and application $M\{R\}$ are erased to $|M|$ during evaluation. So we must be able to discard the irrelevant argument R. Hence we require the irrelevant quantification to be of the form $\forall(x : P).B[x]$ (where P must be a parameter type), and the irrelevant arguments to be parameter terms. The shape operation can be extended for irrelevant quantification.

$$\begin{aligned} \mathrm{Sh}(\forall(x : P).B[x]) &= \forall(x : P).\, \mathrm{Sh}(B[x]) \\ \mathrm{Sh}(\lambda\{x\}.M) &= \lambda\{x\}.\, \mathrm{Sh}(M) \\ \mathrm{Sh}(M\{R\}) &= \mathrm{Sh}(M)\{R\} \end{aligned}$$

Other than annotating types using the keyword `forall`, programmers do not need to work with irrelevant abstraction and application explicitly. Our type elaborator automatically generates the irrelevance annotations, just like linearity annotations. For example, in Proto-Quipper-D, the vector append function has type `!(forall a (n m : Nat) -> Vec a n -> Vec a m -> Vec a (add n m))`, but its definition is the same as that of the list append function.

5 Conclusion

We defined state-parameter fibrations, and showed that they model a general notion of linear dependent types. We derived a linear dependent type system based on the categorical structures obtained from the state-parameter fibration. We showed that Rios and Selinger's $\bar{\bar{\mathbf{M}}}$ admits a state-parameter fibration $\sharp : \bar{\bar{\mathbf{M}}} \to \mathbf{Set}$. We further extended the linear dependent type system to obtain a programming language for quantum circuits, called dependently typed Proto-Quipper. We showed how the fibration $\sharp : \bar{\bar{\mathbf{M}}} \to \mathbf{Set}$ can be used to interpret dependently typed Proto-Quipper. We proved the type preservation and the soundness for the big-step evaluation. We implemented a prototype for dependently typed Proto-Quipper and discussed practical aspects of the implementation.

Acknowledgments

We thank Neil J. Ross for helpful discussions of a draft of this paper, and Marco Gaboardi and Aaron Stump for pointing out related works. This work was supported by the Air Force Office of Scientific Research under award number FA9550-15-1-0331. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Department of Defense.

References

[1] Robert Atkey. 2018. Syntax and semantics of quantitative type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, 56–65.

[2] Andrew Barber. 1996. *Dual intuitionistic linear logic.* Technical Report ECS-LFCS-96-347. University of Edinburgh, Department of Computer Science.

[3] Bruno Barras and Bruno Bernardo. 2008. The implicit calculus of constructions as a programming language with dependent types. In *International Conference on Foundations of Software Science and Computational Structures.* Springer, 365–379.

[4] P. Nick Benton. 1994. A mixed linear and non-linear logic: Proofs, terms and models. In *International Workshop on Computer Science Logic.* Springer, 121–135.

[5] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: practical linearity in a higher-order polymorphic language. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 1–29.

[6] Iliano Cervesato and Frank Pfenning. 1996. A linear logical framework. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 264–275.

[7] Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. 2020. A tutorial introduction to quantum circuit programming in dependently typed Proto-Quipper. In *Proceedings of the 12th International Conference on Reversible Computation*. Springer. To appear.

[8] Peng Fu, Kohei Kishida, and Peter Selinger. 2020. Linear dependent type theory for quantum programming languages. (2020). Available from arXiv:2004.13472.

[9] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce. 2013. Linear dependent types for differential privacy. In *Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (ACM SIGPLAN Notices)*, Vol. 48. ACM, 357–370.

[10] Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101.

[11] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. 1992. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science* 97, 1 (1992), 1–66.

[12] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. An introduction to quantum programming in Quipper. In *Proceedings of the 5th International Conference on Reversible Computation*. Springer, 110–124.

[13] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: a scalable quantum programming language. In *Proceedings of the 34th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (ACM SIGPLAN Notices)*, Vol. 48. ACM, 333–342.

[14] C. Barry Jay and J. Robin B. Cockett. 1994. Shapely types and shape polymorphism. In *European Symposium on Programming*. Springer, 302–316.

[15] Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

[16] G. Max Kelly. 1974. Doctrinal adjunction. In *Category Seminar*. Springer, 257–280.

[17] Neelakantan R Krishnaswami, Pierre Pradic, and Nick Benton. 2015. Integrating linear and dependent types. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (ACM SIGPLAN Notices)*, Vol. 50. ACM, 17–30.

[18] Per Martin-Löf and Giovanni Sambin. 1984. *Intuitionistic type theory*. Bibliopolis Naples.

[19] Conor McBride. 2016. I got plenty o' nuttin'. In *A List of Successes That Can Change the World*. Springer, 207–233. Available from https://personal.cis.strath.ac.uk/conor.mcbride/PlentyO-CR.pdf.

[20] Alexandre Miquel. 2001. The implicit calculus of constructions extending pure type systems with an intersection type binder and subtyping. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 344–359.

[21] Michele Mosca, Martin Roetteler, and Peter Selinger. 2018. Quantum Programming Languages (Dagstuhl Seminar 18381). *Dagstuhl Reports* 8, 9 (2018), 112–132. https://doi.org/10.4230/DagRep.8.9.112

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

[23] Bengt Nordström, Kent Petersson, and Jan M. Smith. 1990. *Programming in Martin-Löf's type theory*. Oxford University Press Oxford.

[24] 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.

[25] Jennifer Paykin and Steve Zdancewic. 2017. The linearity monad. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (ACM SIGPLAN Notices)*, Vol. 52. ACM, 117–132.

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

[27] Robert Rand, Jennifer Paykin, and Steve Zdancewic. 2018. QWIRE practice: Formal verification of quantum circuits in Coq. (2018). Available from arXiv:1803.00699.

[28] Francisco Rios and Peter Selinger. 2017. A categorical model for a quantum circuit description language. In *Proceedings of the 14th International Conference on Quantum Physics and Logic, QPL 2017, Nijmegen, The Netherlands, 3-7 July 2017*. 164–178.

[29] 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.

[30] Robert A. G. Seely. 1984. Locally cartesian closed categories and type theory. In *Mathematical Proceedings of the Cambridge Philosophical Society*, Vol. 95. Cambridge University Press, 33–48.

[31] Peter Selinger and Benoît Valiron. 2006. A Lambda Calculus for Quantum Computation with Classical Control. *Mathematical Structures in Computer Science* 16, 3 (2006), 527–552.

[32] Matthijs Vákár. 2015. A categorical semantics for linear logical frameworks. In *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015. London, UK, April 11-18, 2015. Proceedings*. 102–116.

[33] Benoît Valiron, Neil J. Ross, Peter Selinger, D. Scott Alexander, and Jonathan M. Smith. 2015. Programming the quantum future. *Commun. ACM* 58, 8 (2015), 52–61. https://doi.org/10.1145/2699415

[34] Philip Wadler. 1990. Linear types can change the world!. In *Programming Concepts and Methods*, Vol. 3. Citeseer, 5.

[35] Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 60–76.