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

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 \(\tilde{M}\), 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 \((x : A) \rightarrow B[x]\) and that we are given a linear resource \(a : A\). How should we understand \(fa : B[a]\) (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 \(\lambda x.M : A \multimap B\), the context \(x : A\) must be a linear one. On the other hand, to form a lambda term \(\lambda x.M : (x : A) \rightarrow B[x]\) of a dependent type, the context \(x : A\) must be an intuitionistic one. When applying the linear lambda term \(\lambda x.M\) 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, \(\omega\) to decorate the context. The indices 0, 1, and \(\omega\) 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 \((x : A) \rightarrow B[x]\), 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 \(\Gamma \vdash M : A\), where the index \(k\) on the right of turnstile separates the set of typing judgments into two kinds: \(\Gamma \vdash M : A\) for terms that occur in types and \(\Gamma \vdash M : A\) for terms that will be evaluated at runtime. McBride’s understanding of \(fa : B[a]\) 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 \((x : A) \rightarrow B[x]\) 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 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 Sh\((A)\) for the type of shapes of \(A\), so that Sh\((N) : Sh(A)\). With these conventions, the dependence of \(B\) on \(x\) in \((x : A) \rightarrow B[x]\) is interpreted as \(x : Sh(A) \multimap B[x] : \ast\). It is understood that, for any resource \(N\) (linear or otherwise), its shape Sh\((N)\) is duplicable. Accordingly, our typing rule for application has the following form (where \(\Gamma_i\) has the form
This is beyond Jay and Cockett’s treatment of shapely types.

dependent type theory.

gory over a locally cartesian closed category, can be viewed

So instead of separating typing contexts into linear and nonlinear parts like Cervesato and Pfennig, or of having two kinds of typing judgments like McBride and like Atkey [1], we identify a subset of typing judgment of the form $\text{Sh}(\Gamma) \vdash \text{Sh}(N) : \text{Sh}(A)$. This subfragment of typing accounts for the (duplicate) 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 $\text{Sh}(\Gamma) \vdash \text{Sh}(N) : \text{Sh}(A)$ is only used for typing and kinding purposes, in practice programmers do not directly program with terms such as $\text{Sh}(N)$.

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 $\tilde{\mathcal{M}}$, 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 $F : E \to B$, a morphism $f : B \to A$ of $E$ is said to be cartesian if

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

\[
\begin{align*}
C & \xrightarrow{u} F(C) \\
F(C) & \xrightarrow{f} A \\
F(B) & \xrightarrow{f \circ b} F(A)
\end{align*}
\]

Given an arrow $a : X \to F(A)$ of $B$, 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 $F : E \to B$ has a right adjoint $G$. Then an arrow $f$ of $E$ is cartesian iff the following square is a pullback, where $\eta$ is the unit of the adjunction.

\[
\begin{align*}
B & \xrightarrow{F(f)} GF(A) \\
\eta_B & \downarrow \quad \eta_A \\
A & \xrightarrow{f} G(A)
\end{align*}
\]

Indeed, the following subclass of fibrations can be characterized in terms of adjoints and pullbacks.
Definition 2.3. Let us call a Grothendieck fibration \( \mathcal{F} : E \to B \) a pullback fibration if

1. \( B \) has pullbacks, and
2. \( E \) has a terminal object 1 and \( \mathcal{F} \) preserves it.

Fact 2.4. Given any functor \( \mathcal{F} : E \to B \), it is a pullback fibration if

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

\[
\begin{array}{ccc}
B & \xrightarrow{\pi_2} & A \\
\downarrow & & \downarrow h \\
qY & \xrightarrow{qf} & qX \\
\end{array}
\]

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

It is useful to note how a pullback fibration \( \mathcal{F} \) defines its right adjoint \( q \) in (3): since \( \mathcal{F}1 \) is terminal in \( B \), we set \( \mathcal{F}q1 = 1 \) and define \( qX \) and \( qf \) by cartesian liftings:

\[
\begin{array}{ccc}
qY & \xrightarrow{qf} & qX \\
\downarrow & & \downarrow \\
Y & \xrightarrow{f} & X \\
\end{array}
\]

\( \mathcal{F}1 \).

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

We use the total category \( E \) of such a fibration \( \mathcal{F} \) as a category of quantum spaces. It may not have all pullbacks, but the ones given by \( \mathcal{F} \) are enough for interpreting dependent types.

2.2 State-parameter fibration

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

6. \( \mathcal{F} \) is a monoidal closed functor.

This means that \( \mathcal{F} \) preserves the monoidal closed structure, i.e., it maps \( \otimes, \to, 1 \) in \( E \) to \( \times, \Rightarrow, 1 \) in \( B \).

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

\[
\begin{array}{ccc}
A \otimes_{qX} B & \xrightarrow{\mathcal{F}} & A \times B \\
\downarrow & & \downarrow \\
A \Rightarrow_{qX} B & \xrightarrow{\mathcal{F}} & A \Rightarrow B \\
\end{array}
\]

\( A \otimes_{qX} B \) and \( A \Rightarrow_{qX} B \) are also objects of \( E/qX \). We will assume that

(8) \( - \otimes_{qX} A + A \Rightarrow_{qX} - \) for every object \( X \) of \( B \), so that each \( E/qX \) is monoidal closed. We call \( E \) “sublocally” monoidal closed: while \( E/A \) is not monoidal closed for every object \( A \), it is if \( A \) is in the subcategory \( B \).

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

Definition 2.7. Given a monoidal closed pullback fibration \( \mathcal{F} : E \to B \), we define a functor \( p : B \to E \) by setting \( p1 = 1 \) and taking cartesian liftings of the entire \( B \), or equivalently, pulling back \( p1 = \eta1 : 1 \to \mathcal{F}1 \).

\[
\begin{array}{ccc}
pY & \xrightarrow{pf} & pX \\
\downarrow & & \downarrow \\
Y & \xrightarrow{f} & X \\
\end{array}
\]

\( \mathcal{F}1 \).

It follows that \( p \circ \mathcal{F} = \text{id}_E \). We see \( pX \) as a family of the monoidal unit \( I \) indexed by parameters in \( X \); it is indeed the monoidal unit of \( E/qX \). (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 \( b \),

which is enough to make \( p \dashv b \) a linear-nonlinear adjunction as defined by Benton [4]. We then write \( ! \) for the counad \( p \circ b \) and force \( ![I] \to \text{id}_E \) for the counit.

Putting everything together, we arrive at the following definition.

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

Fact 2.9. Given any state-parameter fibration \( \mathcal{F} \), its parameterized-unit functor \( p \) is strong monoidal, i.e., \( p(X \times Y) \cong \)
\[ pX \otimes pY \text{ and } p1 \equiv I, \text{ and } p + b \text{ is a linear-nonlinear adjunction}. \]

**Proof.** \( p1 \equiv I \) by definition. To show \( p(X \times Y) \equiv pX \otimes pY \), note that \( pX : pX \rightarrow p1 \equiv I \) and \( pY \) are cartesian; hence \( pX \otimes p1 : p(X \times Y) \rightarrow p1 \otimes p1 \equiv p1 \) is also cartesian by (7), while \( pX \otimes pY = p(X \times Y) \) by Definition 2.7. Then, by \[16, \text{Theorem 1.5}], strong monoidal \( p \) makes \( b \) monoidal and \( p + b \) a linear-nonlinear adjunction. \( \square \)

### 2.3 Example: \( \vec{M} \) 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 \( M \) of “generalized circuits” (which could be, for example, a category of quantum circuits), and fully embeds it into some symmetric monoidal closed category \( \vec{M} \) with products, e.g., by taking the Yoneda embedding. Then

**Definition 2.10.** Let \( \vec{M} \) be the free coproduct completion of \( M \). More concretely, it is given as follows:

- An object \( A \) in \( M \) is a pair \( (A, (A, x) \in \text{Set}) \) of a set \( A \) and an indexed family of objects \( A_x \) of the category \( \vec{M} \).
- An arrow \( f : A \rightarrow B \) in \( M \) is a pair \( (f, f_X : A_X \rightarrow B) \) of a function \( f : A \rightarrow B \) and an indexed family of arrows \( f_X : A_x \rightarrow B_{f(x)} \) of \( \vec{M} \).

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

The category \( \vec{M} \) is symmetric monoidal closed, with \( I, \otimes, \rightarrow \) defined by indexed families of \( I, \otimes, \rightarrow \) of \( \vec{M} \) with parameter sets that make \( \# \) monoidal closed: \( I = (1,1) \), \( A \otimes B = (A \times B, (A_X \otimes B_Y)_{x \in \text{Set} B_{x \in \text{Set}}} \), and \( A \rightarrow B = (A \mapsto B_x(x = A_x \otimes B_{f(x)})) \). This definition and the characterization of cartesian arrows above imply (7). Moreover, \( A \otimes qX \) and \( A \rightarrow qX \) are defined to be subfamilies of \( A \otimes B \) and \( A \rightarrow B \) (with smaller parameter sets \( A \times X \) and \( A \rightarrow X \)), which implies (8), that \( \vec{M}/qX \) is monoidal closed. To sum up,

**Fact 2.11.** The functor \( \# : \vec{M} \rightarrow \text{Set} \) 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 subobject monoidal closed structure of a state-parameter fibration \( \#: E \rightarrow B \) enables us to interpret dependent types with several constructions.

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

\[
\begin{array}{c}
\Gamma \\
\eta_A \\
\pi \\
\frac{M}{\Gamma}
\end{array}
\]

In the LCCC interpretation of nondeterministic dependent type theory, one would use an identity arrow \( id_{\Gamma} : \Gamma \rightarrow \Gamma \) on the left of the triangle. In our semantics, it is crucial to use \( \eta_A : \Gamma \rightarrow \Gamma \) 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 \( \Gamma_{\#} \) as the base of the slice \( E/\Gamma \) to express dependency, and a (generally) linear object \( \Gamma \) as the domain of a term to embody linear resources.

One may still note that, when we apply the functor \( \# \) to the above triangle, \( \eta_A = id_{\Gamma} \) becomes the usual triangle from the nonlinear case. In other words, our semantics uses a fibration \( \# \) of the linear type theory of \( E \) over the nonlinear dependent type theory of parameters of the LCCC \( B \).

Now, keeping in mind that \( \pi : B \rightarrow A \) means the dependency of \( B \) on (parameters of) \( A \), we define:

**Definition 2.12.** Given \( \pi : B \rightarrow A \), i.e., when \( B \) is in \( E/A \), we call \( A \otimes A B \) a dependent monoidal product, and write \( A \otimes B \) for it.

Dependent monoidal products will be used to interpret linear \( \Sigma \)-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 \( \Gamma \vdash A : \star \), and therefore an arrow \( \pi : A \rightarrow \Gamma_{\#} \). Then \( \Gamma \otimes A \) 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 \( \Gamma \) from \( \Gamma \otimes A \), since the
Theorem 2.14. They refer primarily to objects and arrows in interpreting shapes in Section 3. of given our objective of treating parameter types and other pose in Section 3, which specifies special subclasses of types following theorem, which provides a semantics for linear space in

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

\[
\begin{align*}
\Pi_{qX,A}B & \to A - qX B \\
\Pi_{qX,A}B & \to A \Rightarrow X B
\end{align*}
\]

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

Theorem 2.14. There is an adjunction

\[
\begin{array}{c}
\mathcal{M}/A \\
\Pi_{qX,A}
\end{array} \rightleftharpoons \begin{array}{c}
\mathcal{M}/qX
\end{array}
\]

2.5 Parameters and shapes

The linear type theory of \( E \) is fibered over the nonlinear dependent type theory of parameters in \( B \), 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 \( B \). Nonetheless, given our objective of treating parameter types and other types uniformly, we need to formally interpret parameter types and terms in \( E \). We do this by embedding \( B \) into \( E \) in two ways, namely, by the functors \( p \) and \( q \): while the image of \( q \) provides slices \( E/qX \) and sublocally monoidal closed structure, parameterized monoidal units \( pX \) and reindexing maps \( pf \) interact with other objects and arrows within the monoidal structure of \( E \) or \( E/qX \), 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 \( E \), which we dub the \textit{shape-terminal functor} and \textit{shape-unit functor}, respectively. While \( q \circ \sharp \) is a monad (of \( \sharp \circ q \)), \( p \circ \sharp \) is not, but one may note that \( p \circ \sharp = q \circ b \).

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 \( B \) and any \( A, B \) in \( E \):

- \( p(pX) = pX \).
- \( p\sharp I = I \) for any \( S \) in \( E \) such that \( S = 1 \).

3 A linear dependent type system

We now consider a type system for the state-parameter fibration \( \sharp : E \to B \).

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 \( B \) being the subcategory of \( E \) 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 \( A, B ::= C | \text{Unit} | !A | (x : A) \Rightarrow B[x] \mid (x : A) \otimes B[x] | (x : P_1) \to P_2[x] \)

Parameter types \( P ::= \text{Unit} | !A | (x : P_1) \otimes P_2[x] \mid (x : P_1) \to P_2[x] \)

Terms \( M, N, L ::= x | \lambda x.M | MN | \text{force} M | \text{force'} R \mid \text{lift} M | (M, N) | \text{let} (x, y) = N \ in \ M | \lambda' x.R | R_1 @ R_2 \mid (R_1, R_2) \mid \text{let} (x, y) = R_1 \ in \ R_2 \)

Parameter terms \( R ::= x | \lambda' x.R | R_1 @ R_2 | \text{force' } R | \text{lift} M \mid (R_1, R_2) \mid \text{let} (x, y) = R_1 \ in \ R_2 \)

Values \( V ::= x | \lambda x.M | \lambda' x.R | \text{lift} M \)

Indices \( k ::= 0 | 1 | \omega \)

Contexts \( \Gamma ::= \cdot \mid x : k, A, \Phi \), 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 existential type \( !A \), which is a parameter type and is duplicable. The term lift \( M \) introduces the type \( !A \), and the term force \( M \) eliminates \( !A \) to \( A \). In the linear \( \Pi \)- and \( \Sigma \)-types \( (x : A) \Rightarrow B[x] \) and \( (x : A) \otimes B[x] \), we allow \( A \) to be any type. We write \( A \Rightarrow B \) and \( A \otimes B \) for them in the special case in which \( B \) does not depend on \( x : A \).

The linear \( \Sigma \)-type \( (x : A) \otimes B[x] \) is a parameter type when \( A \) and \( B[x] \) are. The same is not the case for the \( \Pi \)-type: we introduce a separate, intuitionistic \( \Pi \)-type \( (x : P_1) \to P_2[x] \).
This is introduced and eliminated by parameter terms \( \lambda^\prime x.R \) and \( R_1@R_2 \), respectively. The term force \( ^\prime R \) eliminates \( !A \) to the shape of \( A \), i.e., \( \text{Sh}(A) \). We interpret force \( \lambda^\prime x.\) as the shape of the counit force \( ! \rightarrow id_\text{g} \) of the adjunction \( p \vdash b \).

The operations \( \lambda^\prime, @, \) and force \( ^\prime \), as well as the corresponding type former \( \rightarrow \), 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, \( \omega \) 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 and not \( \omega \). 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 \( \Gamma_1 = (x_1: i_{k_1}, A_1, \ldots, x_n: i_{k_n}, A_n) \) and \( \Gamma_2 = (x_1: i_{k_1}, A_1, \ldots, x_n: i_{k_n}, A_n) \) with the same sequence of types, we write \( \Gamma_1 + \Gamma_2 = (x_1: i_{k_1}, A_1, \ldots, x_n: i_{k_n}, A_n) \), while \( k\Gamma_1 = (x_1: i_{k_1}, A_1, \ldots, x_n: i_{k_n}, A_n) \).

Parameter contexts \( \Phi \) 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 \( \Gamma \).

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 \( p \circ \mu \), and the type-part of the operation corresponds to equalities in Fact 2.15.

**Definition 3.2 (Shape operation).**

\[
\begin{align*}
\text{Sh}(A) &= \text{Unit} \\
\text{Sh}(!A) &= !A \\
\text{Sh}(x:A) &\to B[x]) = (x: \text{Sh}(A)) \to \text{Sh}(B[x]) \\
\text{Sh}(x:A) &\to B[x]) = (x: \text{Sh}(A)) \to \text{Sh}(B[x]) \\
\text{Sh}(P) &= P \\
\text{Sh}(x) &= x \\
\text{Sh}(\lambda x.N) &= \lambda^\prime x. \text{Sh}(N) \\
\text{Sh}(MN) &= \text{Sh}(M) \circ \text{Sh}(N) \\
\text{Sh}(\text{force } M) &= \text{force } \text{Sh}(M) \\
\text{Sh}(\text{lift } M) &= \text{lift } M \\
\text{Sh}(M, N) &= (\text{Sh}(M), \text{Sh}(N)) \\
\text{Sh}(\text{let } (x,y) = N \in M) &= \text{let } (x,y) = \text{Sh}(N) \text{ in } \text{Sh}(M) \\
\text{Sh}(R) &= R
\end{align*}
\]

Observe that the shape operation is idempotent, just like the shape-unit functor \( p \circ \mu \). 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 \( \text{Sh}(x) = x \), the type of the variable \( x \) is changed according to the shape operation (see Theorem 3.6).

Let us write \( \text{Sh}(\Gamma) \) to mean applying the shape operation to all the types in \( \Gamma \). So \( \text{Sh}(\Gamma) \) 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 many times in the type \( B \) occurs once in the term \( M \).

**Definition 3.3 (Well-formed context).**

\[
\begin{align*}
\Gamma \vdash \text{Sh}(\Gamma) + A : * & \quad \text{where } k = \omega \text{ only if } A \text{ is a parameter type} \\
\Gamma \vdash x: \text{Sh}(\Gamma) + A & \quad \text{In the above definition, we use the shape of the context } \text{Sh}(\Gamma) \text{ to kind check a type.}
\end{align*}
\]

**Definition 3.4 (Selected kinding rules).**

\[
\begin{align*}
\Phi, x: \text{Sh}(A) + B[x] : * & \quad \Phi, x: \text{Sh}(A) + B[x] : * \\
\Phi \vdash (x:A) \rightarrow B[x] : * & \quad \Phi \vdash (x:A) \rightarrow B[x] : *
\end{align*}
\]

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

**Definition 3.5 (Selected typing rules).**

\[
\begin{align*}
\Gamma_1 + \Gamma_2 + M:N & \quad \Phi, x: P_1 + P_2[x] : * \\
0 \Gamma_1 + \Gamma_2 + MN : B[\text{Sh}(N)] & \quad \Phi \vdash !A : * \\
0 \Gamma_1 + \Gamma_2 + P_1 + P_2[x] & \quad \Phi \vdash (x:P_1) \rightarrow P_2[x] : *
\end{align*}
\]

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

(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...
Theorem 3.6. (Substitution) 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 \( \Gamma \vdash A : \ast \), and terms \( \Gamma \vdash M : A \) with interpretations \( \llbracket - \rrbracket \) 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 \( A \) is a notation for \( \sharp A \). We often write \( \llbracket \Gamma \rrbracket \) instead of \( \llbracket \Gamma \vdash \rrbracket \).

Theorem 3.11. An interpretation \( \llbracket - \rrbracket \) can be defined in such a way that

10. \( \llbracket \Gamma \rrbracket \) is an object of \( E \).
11. \( \llbracket \Phi + A : \ast \rrbracket \) is an object of \( E/\llbracket \Phi \rrbracket \). We may write \( \pi : \llbracket A \rrbracket \to \llbracket \Phi \rrbracket \) for it.
12. \( \llbracket \Gamma \vdash M : A \rrbracket \) is an arrow of \( E/\llbracket \Gamma \rrbracket \) from \( \eta_{\llbracket \Gamma \rrbracket} \) to \( \llbracket \Gamma \vdash A : \ast \rrbracket \).

### Definition 3.8 (Selected evaluation rules)

\[
\begin{align*}
M \Downarrow N & \quad N \Downarrow N' & M \Downarrow N' \\
MN & \Downarrow N' & M \Downarrow M' \quad M' \Downarrow N \\
R_1 \Downarrow L' \quad R_2 \Downarrow V & \quad [V/x]R_1 \Downarrow R' & R \Downarrow M \quad \text{Sh}(M) \Downarrow R' \\
R_1 \circ R_2 & \Downarrow R' & & \text{force} \quad \text{force}' \\
\end{align*}
\]

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

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.

\[
\begin{align*}
\Gamma \vdash M : A[R] & \quad R \Downarrow R' \\
\Gamma \vdash M : A[R'] \\
\end{align*}
\]

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.

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. (Substitution) We have:

- If \( \Gamma \vdash M : A \), then \( \text{Sh}(\Gamma) \vdash A : \ast \) and \( \Gamma \vdash \).
- If \( \Gamma \vdash M : A \), then \( \text{Sh}(\Gamma) \vdash \text{Sh}(M) : \text{Sh}(A) \).

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). We have:

- If \( \Phi, x : P \vdash B[x] : \ast \) and \( \Phi \vdash R : P \), then \( \Phi \vdash B[R] : \ast \).
- If \( \Phi, x : P, \Gamma \vdash M : B[x] \) and \( \Phi \vdash R : P \), then \( \Phi, [R/x]\Gamma \vdash B[R] \).
- If \( \Gamma_1, x : A, \Gamma_1 \vdash M : B[x] \) and \( \Gamma_2 \vdash V : A \), then \( \Gamma_1 \vdash k_{\Gamma_2} [\text{Sh}(V)/x] \Gamma' \vdash [V/x]M : B[\text{Sh}(V)] \).

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.
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

\[ \frac{\Gamma \vdash \text{Sh}(\Gamma) \vdash A : *}{\Gamma, x :_k A \vdash k = \omega} \]

By the assumptions, we have objects \([\Gamma]\) of E and \(\pi : [A] \to [\text{Sh}(\Gamma)] = [\Gamma]\) of \(E/\text{[\Gamma]}\). We therefore define

\[ [\Gamma, x :_k A] = [\Gamma] \otimes [A]^k \]

using a dependent monoidal product, where \([A]^k = \rho([A])\) if \(k = 0\) and \([A]^k = [A]\) otherwise.

(ii) The second case is

\[ \frac{\Phi, x :_k \text{Sh}(A) \vdash B[x] : *}{\Phi \vdash (x : A) \to B[x] : *} \]

The assumption implies \(\Phi \vdash A : *\). Hence we have arrows \(\pi : [B] \to [\Phi, x :_k \text{Sh}(A)] = [A]\) and \([A] \to [\Phi]\). We therefore define

\[ [\Phi \vdash (x : A) \to B[x] : *] = \Pi_{[\Phi][A]}[B] \]

using a dependent function space that gives an object in \(E/\text{[\Phi]}\).

(iii) The third case is

\[ \frac{0 \vdash x : A, 0 \vdash x : A}{0 \vdash x : A, 0 \vdash x : A} \]

Since \(\text{Sh}(\Gamma) \vdash A : *\), by (11) we have \(\pi : [A] \to [\Gamma]\). Note that there is a canonical morphism \(f : \rho(\Gamma) \otimes [A] \otimes \rho(\Gamma') \to [A] \text{ over } [\Gamma]\). We define \([x] \text{ as the following } u \text{ over}\)

\[ [\Gamma, x : A, \Gamma'] \]

By the assumptions, we have a morphism \([M] : [\Gamma] \otimes [\Gamma'] \to [B]\) over \([\Gamma, x : A]\). By the adjunction in Theorem 2.14, we define \([\lambda x.M] = [M] : [\Gamma] \to \Pi_{[\Gamma][A]}[B]\) as a morphism over \([\Gamma]\).

(v) The last case we review here is

\[ \frac{\Gamma_1 \vdash M : (x : A) \to B[x]}{\Gamma_1 + \Gamma_2 \vdash [N] \to [\text{Sh}(\Gamma)]} \]

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

\[ \begin{array}{c}
\Pi_{[\Gamma][A]}[B] \\
\downarrow \pi \\
\Gamma \\
\uparrow \eta \\
\Gamma' \\
\end{array} \]

\[ \begin{array}{c}
\Pi_{[\Gamma][A]}[B] \\
\downarrow \pi \\
\Gamma \\
\uparrow \eta \\
\Gamma' \\
\end{array} \]

It is crucial to observe that \([\Gamma_1 + \Gamma_2] = [\Gamma_1] \otimes [\Gamma_2]\). This then enables us to use \(\epsilon : (\Pi_{[\Gamma][A]}[B]) \otimes [\Gamma][A] \to [B]\), 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 \([M] = [M']\).

### 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 un-boxing. 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., \( \# : \mathbb{M} \rightarrow \text{Set} \). 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 \( \mathbb{M} \) of \([28]\) contains a (full) monoidal subcategory \( \mathbb{M} \) of (generalized) circuits. Objects \( S \) of \( \mathbb{M} \) (or \( (1, S) \) of \( \mathbb{M} \)) are called simple. It follows that \( S \) is simple iff \( \# S = 1 \). 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

\[
\mathsf{box/unbox}(S_1 \rightarrow S_2) \cong p\mathbb{M}(S_1, S_2).
\]

This isomorphism means that the function type \( !(S_1 \rightarrow S_2) \) literally represents a hom-set of the category \( \mathbb{M} \), 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 \( \text{Circ}(S_1, S_2) \) representing such circuits as first-class citizens. These linear functions of simple types correspond to quantum circuits.

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

**A problem of quantum data types.** The box/unbox isomorphism only holds for simple types. However, \( \text{List Qubit} \) is not a simple type since \( \#(\text{List Qubit}) = \#((\mathbb{N}, Q^{\otimes n})_{n \in \mathbb{N}}) = \mathbb{N} \). Thus, there is no such type as \( \text{Circ}(\text{List Qubit}) \) in Proto-Quipper-M, and a function of type \( !(\text{List Qubit} \rightarrow \text{List Qubit}) \) 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 \( \text{Vec Qubit} n \). For a value \( n \) of type \( \text{Nat} \), we define \( \llbracket \text{Vec Qubit} n \rrbracket \) as \( V_n = (1, Q^{\otimes n}) \) as in the following pullback square. Note that the existence of the pullback is guaranteed by the state-parameter fibration \( \# : \mathbb{M} \rightarrow \text{Set} \).

\[
\begin{array}{ccc}
\text{Vec Qubit} n & \rightarrow & \text{List Qubit} \\
\downarrow & & \downarrow \\
q1 & \\
\end{array}
\]

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

\[
!(n : \text{Nat}) \rightarrow \text{Circ}(\text{Vec Qubit} n, \text{Vec Qubit} n).
\]

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 \( A, B ::= \ldots | \text{Nat} | \text{List} A | \text{Vec} A R | \text{Circ}(S_1, S_2) \)
- Simple Types \( S ::= \text{Qubit} | \text{Unit} | S_1 \otimes S_2 | \text{Vec} S R \)
- Parameter types \( P ::= \ldots | \text{Nat} | \text{List} P | \text{Vec} P R | \text{Circ}(S_1, S_2) \)
- Terms \( M ::= \ldots | \ell | (a, C, b) | \text{box} M | \text{unbox} M \)

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

Following Proto-Quipper-M, we extend terms with labels \( \ell \), 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 \( \Sigma = \ell_1, \ldots, \ell_n, S_1, S_2, \ldots \). The semantics of a label context is a tensor product of qubits, and we identify each \( \Sigma \) with the corresponding object in \( \mathbb{M} \).

The canonical inhabitants of \( \text{Circ}(S_1, S_2) \) are *boxed circuits* of the form \( (a, C, b) \), where \( C \) is a morphism of \( \mathbb{M} \) 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 \( \text{Circ}(S_1, S_2) \) are the same as in Proto-Quipper-M:

\[
\begin{array}{c}
\Sigma_1 \vdash a : S_1 \quad \Sigma_2 \vdash b : S_2 \\
\Phi \vdash (a, C, b) : \text{Circ}(S_1, S_2) \\
\Gamma \vdash M = !(S_1 \rightarrow S_2) \\
\Gamma \vdash \text{box} S_1 M : \text{Circ}(S_1, S_2) \\
\Gamma \vdash \text{unbox} M : !(S_1 \rightarrow S_2)
\end{array}
\]

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 : Circ(S₁, S₂) → Circ(S₂, S₁). Indeed, although the types !(S₁ → S₂) and Circ(S₁, S₂) are semantically isomorphic, they are operationally distinct, because Circ(S₁, S₂) has canonical inhabitants whereas !(S₁ → S₂) does not.

The shape operation can be extended naturally.

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

\[
\begin{align*}
\text{Sh}(\text{Nat}) &= \text{Nat} \\
\text{Sh}(\text{List } A) &= \text{List } \text{Sh}(A) \\
\text{Sh}(\text{Vec } A R) &= \text{Vec } \text{Sh}(A) R \\
\text{Sh}(\text{Circ}(S_1, S_2)) &= \text{Circ}(\text{Sh}(S_1), \text{Sh}(S_2)) \\
\text{Sh}(t) &= \text{unit} \\
\text{Sh}(a, C, b) &= (a, C, b) \\
\text{Sh}((\text{box}_S M)) &= \text{box}_S \text{Sh}(M) \\
\text{Sh}((\text{unbox}_M)) &= \text{unbox}_S \text{Sh}(M)
\end{align*}
\]

This definition is semantically sound for List and Vec because the shape-unit functor \( p^\sharp \) preserves tensor products and coproducts in \( \overline{\mathbf{S}} \). We also observe that \( \text{Sh}((\text{List } \text{Qubit})) = \text{List } \text{Unit} \) or \( \text{Sh}((\text{Vec } \text{Qubit } R)) = \text{Vec } \text{Unit } R \equiv \text{Unit} \).

**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.\(^2\)

\[
\begin{align*}
\text{conv}(x : \text{List } \text{Qubit}) &\rightarrow \text{Vec } \text{Qubit} \text{ (toNat } x) \\
\text{conv} &= \begin{cases} \\
\text{lift}(\lambda x. \text{case } x \text{ of} \\
\text{Nil} &\rightarrow \text{VNil} \\
\text{Cons } y s &\rightarrow \text{VCons } y (\text{force conv } y s) \\
\end{cases}
\end{align*}
\]

Let us consider the type

\[
(x : \text{List } \text{Qubit}) \rightarrow \text{Vec } \text{Qubit} \text{ (toNat } x).
\]

Here toNat : List Unit \rightarrow 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

\[
x : \text{Sh}(\text{List } \text{Qubit}) = \text{List } \text{Unit} \sqsubseteq \text{Vec } \text{Qubit} \text{ (toNat } x) : *,
\]

which is a valid kinding judgment. So a function of the type \((x : \text{List } \text{Qubit}) \rightarrow \text{Vec } \text{Qubit} \text{ (toNat } x)\) 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 \((x : P) \rightarrow B[x]\), where \(P\) is a parameter type. McBride’s conversion function has a different flavor. In [19, Section 5], he defines \(\text{conv} \quad \text{toNat} \quad (x : \text{List } X) \rightarrow \text{Vec } X(\text{length } x)\), where length \(\text{toNat} \quad (x : \text{List } X) \rightarrow \text{Nat}.\) We cannot define a length function of type \(\text{List } \text{Qubit} \rightarrow \text{Nat}\) 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 \(M\), 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 \(\Sigma''\) such that \(C : \Sigma \rightarrow \Sigma'' \otimes \Sigma'\) 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 \(\Gamma\). 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 \(\Sigma''\).

\[
\begin{array}{c}
\Sigma \\
\downarrow \\
C \\
\downarrow \\
\Sigma'' \\
\downarrow \\
M \\
\downarrow \\
A
\end{array}
\]

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).**

\[
\begin{align*}
(C, M) \downarrow (C', \text{force (unbox } (a, D, b))) \\
(C, N) \downarrow (C', a') \\
C_i = \text{append}(C_i', a', (a, D, b)), FV(MN) = \emptyset \\
(C_1, MN) \downarrow (C_4, b) \\
(C, M) \downarrow (C', \text{lift } M') \\
(\text{id}_M, M') \downarrow (\emptyset, V) \\
(a = \text{gen}(S[V]),) FV(\text{box}_S[R] M) = \emptyset \\
(C, \text{box}_S[R] M) \downarrow (C', (a, D, b)) \\
(C, M) \downarrow (C', \lambda x.M') \\
(C', N) \downarrow (C'', V) \\
(C'', [V/x] M') \downarrow (C''', N') \\
(C, MN) \downarrow (C''', N''') \\
(C, M) \downarrow (C', \text{lift } M') \\
(C', M') \downarrow (C'', N) \\
(C, \text{force } M) \downarrow (C'', N) \\
(C, M) \downarrow (C', \text{unbox } M') \\
(C, \text{box}_S M) \downarrow (C', M')
\end{align*}
\]

\(^3\)A detailed definition of the evaluation rules is in [8].
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 \(!A \to B\), the elaborator will produce a new term (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.

```plaintext
simple Vec a : Nat -> Type where
  Vec a Z = VN1
  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 \( \exists S = 1 \). 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.

**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
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 (Simple a) => Simple (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 ⊗ 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.

\[
\begin{align*}
\Phi, x : P, \Gamma &\vdash M : B[x] \\
\Phi, \Gamma &\vdash \lambda x.M : \forall (x : P).B[x] \\
\Phi, \Gamma &\vdash M : \forall (x : P).B[x] \\
\Phi, \Gamma &\vdash R : A \\
\Phi, \Gamma &\vdash M[R] : B[R]
\end{align*}
\]

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.

\[
\begin{align*}
\text{Sh}(\forall (x : P).B[x]) &\equiv \forall (x : P).\text{Sh}(B[x]) \\
\text{Sh}(\lambda x.M) &\equiv \lambda x.\text{Sh}(M) \\
\text{Sh}(M[R]) &\equiv \text{Sh}(M)[R]
\end{align*}
\]

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 \(\overline{M}\) admits a state-parameter fibration \(\downarrow : \overline{M} \to \text{Set}\). 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 \(\downarrow : \overline{M} \to \text{Set}\) 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
