A categorical model for a quantum circuit description language (extended abstract)

Francisco Rios and Peter Selinger
Dalhousie University
Halifax, Canada

Quipper is a practical programming language for describing families of quantum circuits. In this paper, we formalize a small, but useful fragment of Quipper called *Proto-Quipper-M*. Unlike its parent Quipper, this language is type-safe and has a formal denotational and operational semantics. Proto-Quipper-M is also more general than Quipper, in that it can describe families of morphisms in any symmetric monoidal category, of which quantum circuits are but one example. We design Proto-Quipper-M from the ground up, by first giving a general categorical model of *parameters* and *state*. The distinction between parameters and state is also known from hardware description languages. A parameter is a value that is known at circuit generation time, whereas a state is a value that is known at circuit execution time. After finding some interesting categorical structures in the model, we then define the programming language to fit the model. We cement the connection between the language and the model by proving type safety, soundness, and adequacy properties.

1 Introduction

Quipper is a functional programming language for quantum computing [3, 4]. What distinguishes Quipper from earlier quantum programming languages, such as the quantum lambda calculus [10], is that it is a *circuit description language*. This means two things: on the one hand, the language can be used to construct quantum circuits in a structured way, essentially by applying one gate at a time. On the other hand, the completed circuits themselves become data, which can be stored in variables and passed as components to subroutines, and on which meta-operations (such as circuit transformations, gate counts, inversion, error correction, etc) can be performed. These two levels of description (gate level operations and meta-operations on entire circuits) correspond quite closely to how many quantum algorithms are specified in the literature, and therefore provide a useful high-level paradigm for quantum programming.

Quipper is *practical*; it has been used to implement several large-scale quantum algorithms, generating circuits containing trillions of gates. However, there are some drawbacks. For efficiency reason, the original Quipper language was implemented as an embedded domain-specific language within the host language Haskell. As a result of mismatches between Quipper's type system and that of Haskell, Quipper is not type-safe (there are some well-typed programs that lead to run-time errors). In particular, Haskell is unable to enforce *linearity*, i.e., the requirement that a quantum state cannot be used more than once (also known as the no-cloning property of quantum information). Moreover, as an embedded language, Quipper has no formal semantics; its behavior is only defined by its implementation. Giving a formal semantics of Quipper would require giving a formal semantics of Haskell, which is not feasible.

In this paper, we formalize a small, but useful fragment of Quipper called *Proto-Quipper-M*. This fragment is a stand-alone programming language (i.e., not embedded in a host language), and it has its own custom type system and semantics. This research fits into a larger program of formalizing portions of Quipper, and we use the name *Proto-Quipper* more generally to refer to any such formalized languages. For example, another version of Proto-Quipper has appeared in Ross's Ph.D. thesis [9].

An important concept that arises in Quipper, and also in related settings such as hardware description languages, is the distinction between parameters and state. To understand the distinction, it is useful to keep in mind that Quipper is a language not just for describing individual quantum circuits, but parameterized *families* of quantum circuits: for example, in the case of Shor's algorithm, one circuit for each integer to be factored. A *parameter* is a value that is known at circuit generation time, thus potentially picking out a member of the family of circuits. A *state* is a value that is known at circuit execution time, such as the state of a qubit. Naturally, a state can depend on a parameter, but since states are not known at circuit generation time, parameters cannot be a function of states.

Conceptually, Quipper's gate level operations operate on states (for example, by applying a gate to a qubit), whereas its circuit level operations operate on parameters. One fundamental feature of Quipper is that there is only one kind of variable, which can hold both parameters and states (or any combination thereof, such as a list of qubits: here the length of the list is a parameter, and the individual qubits are states). We rely on the *type system* to ensure that parameters and states are used correctly. This distinguishes Quipper from related approaches, such as the QWire language [8], which enforce a strict separation between parameters and states by essentially defining two separate programming languages, one for parameters and one for states. The Quipper approach is more flexible, because the ability to mix parameters and states more freely gives the programmer access to useful abstractions, such as quantum data structures (like lists of qubits), or even pairs of entangled functions.

Our approach to the design of Proto-Quipper-M is from the ground up. We start by defining a general categorical model of parameters and state, and identifying a number of interesting categorical structures in the model. We then define the programming language to fit the model. One advantage of this approach is that our programming language is almost "correct by design". In fact, as a result of our abstract approach, Proto-Quipper-M is slightly more general than Quipper, in the sense that it can describe families of morphisms of an arbitrary monoidal category, rather than just quantum circuits. We give computational meaning to the language by defining an operational semantics. Finally, we establish the correctness of the operational semantics by proving type safety, soundness, and adequacy properties.

2 A cartesian model of parameters and state

In this section, we describe a simplified categorical model of parameters and state, which will be convenient for introducing some useful terminology, and will be a stepping stone toward the more general model of Section 3. The model presented here is cartesian, rather than monoidal, and therefore could be used to model a language for describing classical, rather than quantum circuits. Nevertheless, several important notions will already be visible in this model.

2.1 The model $\mathbf{Set}^{\mathbf{2}^{op}}$

Definition 2.1. Let $\mathbf{2}$ be the category with two objects, called 0 and 1, and a single non-identity arrow $0 \to 1$. Consider the functor category $\mathbf{Set}^{\mathbf{2}^{op}}$. Explicitly, an object of this category is a triple $A = (A_0, A_1, a)$, where A_0, A_1 are sets and $a : A_1 \to A_0$ is a function. A morphism $f : A \to B$ is a commutative diagram

$$\begin{tikzcd} A_1 \arrow[r, "f_1"] \arrow[d, "a"'] & B_1 \arrow[d, "b"] \\ A_0 \arrow[r, "f_0"] & B_0. \end{tikzcd} \tag{1}$$

We can think of an object $A = (A_0, A_1, a)$ as describing a family of sets, as follows. For each $x \in A_0$, we define

$$A_x = \{s \in A_1 \mid a(s) = x\}. \tag{2}$$

The set A_x is called the *fiber* of A over x. Up to isomorphism, the object A is uniquely determined by the family $(A_x)_{x \in A_0}$. We call the elements of A_0 *parameters*. We call the elements of A_x *states*, and we say that the state is *over* x. The elements of A_1 can therefore be identified with pairs (x, s), where $x \in A_0$ and $s \in A_x$. We also call the pairs (x, s) the *generalized elements* of A.

The requirement that the diagram (1) commutes is exactly equivalent to the statement "states may depend on parameters, but parameters may not depend on states". To see this, consider the effect of a morphism $f : A \to B$ on a parameter-state pair (x, s) as defined above. Let $y = f_0(x)$ and $t = f_1(s)$. Then $y \in B_0$ and $t \in B_1$. Moreover, by the commutativity of (1), we have $y = f_0(x) = f_0(a(s)) = b(f_1(s)) = b(t)$. In other words, for each $x \in A_0$,

$f_1 : A_1 \to B_1$ restricts to a function $f_x : A_x \to B_{f_0(x)}$. We write $f(x,s) = (y,t)$. We note that y is only a function of x, because $y = f_0(x)$. Therefore, parameters may not depend on states. On the other hand, t is a function of both x and s, because $t = f_x(s)$. Therefore, states may depend on parameters and states.

Example 2.2. We define some particular objects of $\mathbf{Set}^{\mathbf{2}^{op}}$. Let $\mathbf{bool} = (2,2,\mathrm{id})$, where $2 = \{0,1\}$ is a 2-element set and id is the identity function. Let $\mathbf{bit} = (1,2,u)$, where $1 = \{*\}$ is a 1-element set and $u : 2 \to 1$ is the unique function. In diagrams:

$$\mathbf{bool} = \begin{array}{c} 2 \\ \downarrow \scriptstyle\mathrm{id} \\ 2 \end{array} \qquad \mathbf{bit} = \begin{array}{c} 2 \\ \downarrow \scriptstyle u \\ 1 \end{array}$$

The two generalized elements of **bool** are $(0,0)$ and $(1,1)$, which we identify with "false" and "true", respectively. The two generalized elements of **bit** are $(*,0)$ and $(*,1)$, which we again identify with "false" and "true".

So what is the difference between **bool** and **bit**? Informally, a boolean is only a parameter and has no state, whereas a bit is only state and has no parameters. Note that there is an "identity" function $f : \mathbf{bool} \to \mathbf{bit}$, mapping false to false and true to true. This function is given by the commutative diagram

$$\begin{tikzcd} 2 \arrow[r, "\mathrm{id}"] \arrow[d, "\mathrm{id}"'] & 2 \arrow[d, "u"] \\ 2 \arrow[r, "u"] & 1, \end{tikzcd} \tag{3}$$

and it satisfies $f(0,0) = (*,0)$ and $f(1,1) = (*,1)$. On the other hand, there exists no morphism $g : \mathbf{bit} \to \mathbf{bool}$ mapping false to false and true to true: the diagram

$$\begin{tikzcd} 2 \arrow[r, "\mathrm{id}"] \arrow[d, "u"'] & 2 \arrow[d, "\mathrm{id}"] \\ 1 \arrow[r, "?"] & 2 \end{tikzcd} \tag{4}$$

cannot be made to commute. Therefore, a boolean can be used to initialize a bit, but not the other way round. This precisely captures our basic intuition about parameters and state.

Generalizing the example of **bool** and **bit**, we say that an object A is a *parameter object* if it is of the form (A,A,id), and it is called a *state object* or *simple* if A_0 is a singleton. Note that **bool** is a parameter object and **bit** is a state object. Informally, a simple object corresponds to a single piece of data, rather than a parameterized family of data. Every object A is isomorphic to a sum of simple objects, namely, $A \cong \sum_{x \in A_0} A_x$.

2.2 A lambda calculus for parameters and state

Since the category $\mathbf{Set}^{\mathbf{2}^{op}}$ is cartesian closed, we can interpret the simply-typed lambda calculus in it. We can also add sum types, base types such as **bool** and **bit**, and basic operations such as $\mathbf{init} : \mathbf{bool} \to \mathbf{bit}$, all of which have obvious interpretations in the model. In this way, we obtain a very simple and semantically sound lambda calculus for the description of boolean (non-reversible) circuits. Moreover, the category $\mathbf{Set}^{\mathbf{2}^{op}}$ is co-complete, which allows us to interpret inductive datatypes such as $\mathbf{list}(A)$ using initial algebra semantics.

3 A categorical model of circuit families

While the model of Section 2.1 is useful for formalizing some basic intuitions about parameters and state, it is not yet a good model for describing families of quantum circuits. The main issue is that the model is cartesian, rather than monoidal. For example, there exist morphisms $\Delta : A \to A \times A$ for all objects A, including state objects. This is not appropriate if we want to describe quantum circuits, where the no-cloning property prevents us from duplicating quantum states.

To see how to generalize the model $\mathbf{Set}^{\mathbf{2}^{op}}$ to a monoidal setting, recall from (2) that the objects of $\mathbf{Set}^{\mathbf{2}^{op}}$ can be equivalently described as pairs $(A_0, (A_x)_{x \in A_0})$, where A_0 is a set and $(A_x)_{x \in A_0}$ is a family of sets. Then a morphism $f : A \to B$ can be equivalently described as a pair $(f_0, (f_x)_{x\in A_0})$, where $f_0 : A_0 \to B_0$, and for each $x \in A_0$, $f_x : A_x \to B_{f_0(x)}$. We generalize this by considering A_x to be an object of a monoidal category instead of a set.

3.1 The category $\mathbf{M}$: generalized circuits

Before designing a circuit description language, we should be more precise about what we mean by a "circuit". Rather than specifying a particular class of circuits, for example as graphical representations of sequences of gates, we take a more general and abstract point of view: for us, a circuit is simply a morphism in a (fixed but arbitrary) symmetric monoidal category. We therefore assume that a symmetric monoidal category $\mathbf{M}$ is given once and for all, and we call its morphisms *generalized circuits*. From this point of view, Proto-Quipper is simply a language for describing families of morphisms of $\mathbf{M}$.

Remark 3.1. We will regard the morphisms of $\mathbf{M}$ as *concrete* data. We should imagine that the category $\mathbf{M}$ is equipped with additional "meta-operations", such as methods to print morphisms, determine their size or cost, invert them, and so on. These additional operations are external to the category $\mathbf{M}$, and take the form of set-theoretic functions such as $\mathit{size} : \mathbf{M}(T,U) \to \mathbb{N}$, $\mathit{print} : \mathbf{M}(T,U) \to \text{Document}$, $\mathit{invert} : \mathbf{M}(T,T) \to \mathbf{M}(T,T)$. Here $\mathbf{M}(T,U)$ denotes a hom-set of the category $\mathbf{M}$, and "Document" denotes a set of printable documents. If such meta-operations are present, we regard them as fixed and given.

3.2 The category $\overline{\mathbf{M}}$: state

As the first step in constructing our model, we choose a full embedding $\mathbf{M}$ in some symmetric monoidal closed, product-complete category $\overline{\mathbf{M}}$. This can be done in some fixed, but arbitrary way. For example, the Yoneda embedding has the required properties, using the Day tensor for the monoidal structure [2, 6]. However, we do not specify any particular way of constructing $\overline{\mathbf{M}}$.

Remark 3.2. Unlike the category $\mathbf{M}$, we regard the category $\overline{\mathbf{M}}$ as *abstract*. It is monoidal closed, so we will be able to form higher-order objects such as $(A \multimap B \otimes C) \multimap D$. However, we do not imagine morphisms between such higher-order objects as being concrete things that can be printed, measured, etc. Instead, we will only interact with such higher-order morphisms via the monoidal closed structure. In other words, the higher-order structure only exists as a kind of "scaffolding" to support lower-order concrete operations. By not specifying a particular way of constructing $\overline{\mathbf{M}}$, but only specifying its properties, we ensure that we will treat this category in the abstract, i.e., none of the theorems we will prove depend on any properties of $\overline{\mathbf{M}}$ other than those that were stated.

3.3 The category $\overline{\overline{\mathbf{M}}}$: parameters

We now define the category $\overline{\overline{\mathbf{M}}}$, which will serve as a model for parameters and state, and which will be the main carrier of the categorical semantics of our circuit description language.

Definition 3.3. The category $\overline{\overline{\mathbf{M}}}$ has the following objects and morphisms:

- An object is a pair $A = (X, (A_x)_{x\in X})$, where X is a set and $(A_x)_{x\in X}$ is an X-indexed family of objects of $\overline{\mathbf{M}}$. As before, we call A_x the *fiber* of A over x. We sometimes write $X = A_0$.
- A morphism $f : (X, (A_x)_{x\in X}) \to (Y, (B_y)_{y\in Y})$ is a pair $(f_0, (f_x)_{x\in X})$, where $f_0 : X \to Y$ is a function and each $f_x : A_x \to B_{f_0(x)}$ is a morphism of $\overline{\mathbf{M}}$.

This category is surprisingly rich in structure. We begin by stating some of its most fundamental properties. It is perhaps not very surprising that $\overline{\overline{\mathbf{M}}}$ has coproducts and a symmetric monoidal structure. What is perhaps more surprising is that it is also monoidal closed.

Proposition 3.4. *The category $\overline{\overline{\mathbf{M}}}$ has an initial object, given by $0 = (\emptyset, \emptyset)$, where $\emptyset$ denotes both the empty set and the empty family. It also has coproducts, given by $A + B = (A_0 + B_0, (C_i)_i)$, where $A_0 + B_0 = \{(0,x) \mid$*

$x \in A_0\} \cup \{(1,y) \mid y \in B_0\}$ is the disjoint union of sets, and $C_{(0,x)} = A_x$ and $C_{(1,y)} = B_y$. The category $\overline{\overline{\mathbf{M}}}$ also has infinite coproducts, defined in an analogous manner. Indeed, it is well-known that $\overline{\overline{\mathbf{M}}}$ is the free coproduct completion of $\overline{\mathbf{M}}$.

Proposition 3.5. *The category $\overline{\overline{\mathbf{M}}}$ is symmetric monoidal closed with the following structure:*

$$\begin{array}{lcl} I & = & (1,(I)) \\ A \otimes B & = & (A_0 \times B_0, (A_x \otimes B_y)_{(x,y)\in A_0\times B_0}) \\ A \multimap B & = & (A_0 \to B_0, (C_f)_f), \end{array}$$

where $C_f = \prod_{x\in A_0}(A_x \multimap B_{f(x)})$. Here, of course, $A_0 \to B_0$ denotes the set of all functions from A_0 to B_0, and $A_x \multimap B_y$ denotes the exponential object in the monoidal closed category $\overline{\mathbf{M}}$. Note that in the definition of C_f, we have used the fact that $\overline{\mathbf{M}}$ has products.

We note that the category $\mathbf{Set}^{\mathbf{2}^{op}}$ of Section 2 is a special case; indeed, if the initial monoidal category is $\mathbf{M} = \mathbf{Set}$, then it is already closed, so we can take $\overline{\mathbf{M}} = \mathbf{Set}$, and we get $\overline{\overline{\mathbf{M}}} \simeq \mathbf{Set}^{\mathbf{2}^{op}}$.

3.4 Properties of objects

The concepts of parameter and state objects can be defined analogously to Section 2.1. Namely, an object $A \in \overline{\overline{\mathbf{M}}}$ is a *parameter object* if each fiber is the tensor unit I, i.e., if $A = (X,(I)_{x\in X})$. An object $A \in \overline{\overline{\mathbf{M}}}$ is a *state object* or *simple* if $A_0 \cong 1$. Note that the full subcategory of $\overline{\overline{\mathbf{M}}}$ of simple objects is equivalent to $\overline{\mathbf{M}}$. An object $A \in \overline{\overline{\mathbf{M}}}$ is an *M-object* if every fiber belongs to the category $\mathbf{M}$ (regarded as a full subcategory of $\overline{\mathbf{M}}$). Thus, M-objects denote families of objects of $\mathbf{M}$. Note that the full subcategory of $\overline{\overline{\mathbf{M}}}$ of *simple M-objects* is equivalent to $\mathbf{M}$. The terminology “simple” is justified by the fact that every object $A = (X,(A_x)_x)$ of $\overline{\overline{\mathbf{M}}}$ can be written, essentially uniquely, as a coproduct of simple objects.

More systematically, note that we have functors $\mathbf{Set} \xrightarrow{p} \overline{\overline{\mathbf{M}}}$ and $\mathbf{M} \overset{i}{\hookrightarrow} \overline{\mathbf{M}} \overset{j}{\hookrightarrow} \overline{\overline{\mathbf{M}}}$, where $p(X) = (X,(I)_x)$, i is the canonical inclusion, and $j(A) = (1,(A))$. Then A is a parameter object iff it is in the image of the functor p, a simple object iff it is in the image of j, and a simple M-object iff it is in the image of $j \circ i$.

3.5 Some basic types and operations

The coproducts of $\overline{\overline{\mathbf{M}}}$ permit us to construct a parameter object $\mathbf{bool} = I + I$. This object is equipped with morphisms $\mathbf{true},\mathbf{false} : I \to \mathbf{bool}$ and an if-then-else construction, i.e., an operation mapping a pair of morphisms $f,g : A \to B$ to a morphism $h : \mathbf{bool} \otimes A \to B$ satisfying appropriate conditions. Similarly, there is an object $\mathbf{nat} = (\mathbb{N},(I)_n)$, and indeed there is a parameter object $p(X)$ corresponding to every set X, arising from the functor $p : \mathbf{Set} \to \overline{\overline{\mathbf{M}}}$.

Assume the category $\mathbf{M}$ has some distinguished objects, say **bit** and **qubit**, and some distinguished morphisms, for example $H : \mathbf{qubit} \to \mathbf{qubit}$ and $\mathbf{meas} : \mathbf{qubit} \to \mathbf{bit}$. The latter are called *gates*. Then there are corresponding objects and morphisms in $\overline{\overline{\mathbf{M}}}$, arising from the embedding $\mathbf{M} \hookrightarrow \overline{\overline{\mathbf{M}}}$.

3.6 Inductive types

Recall that a functor F on a category is *continuous* if it preserves colimits of diagrams of the form $A_0 \to A_1 \to \ldots$. In a category with colimits, every continuous functor has an initial algebra. This is the basis for the categorical semantics of inductive datatypes. Unfortunately, the category $\overline{\overline{\mathbf{M}}}$ only has coproducts, and not necessarily all colimits, so we cannot in general expect initial algebras of continuous functors to exist. Nevertheless, for many functors of interest, the required initial algebras exist. For example, consider the functor $F(X) = I + A \otimes X$. Its initial algebra is the infinite coproduct $I + A + A \otimes A + A \otimes A \otimes A + \ldots$. We denote it as $\mathbf{list}(A)$, the type of lists of A. An analogous construction also works for other functors constructed from $+$ and $\otimes$. Another example is the object **nat** of natural numbers, which arises as the initial algebra of $F(X) = I + X$, and is also isomorphic to $\mathbf{list}(I)$.

3.7 Boxing

Proposition 3.6. *The functor* $p : \mathbf{Set} \to \overline{\overline{\mathbf{M}}}$*, defined in Section 3.4, has a right adjoint* $\flat : \overline{\overline{\mathbf{M}}} \to \mathbf{Set}$*. It is given as follows, where* $\overline{\mathbf{M}}(A,B)$ *denotes a hom-set of the category* $\overline{\mathbf{M}}$*:*

$$\flat(X,(A_x)_{x\in X}) = \sum_{x\in X} \overline{\mathbf{M}}(I,A_x).$$

An important special case arises for simple M-objects T and U. In this case, we have

$$\flat(T \multimap U) \;\cong\; \overline{\mathbf{M}}(I, T \multimap U) \;\cong\; \overline{\mathbf{M}}(T,U) \;\cong\; \mathbf{M}(T,U). \tag{5}$$

In other words, $\flat(T \multimap U)$ is just a hom-set of the category $\mathbf{M}$, i.e., a set of generalized circuits. We would like to be able to use completed circuits as parameters in the construction of other circuits, i.e., we would like there to be a parameter object whose elements are circuits. Such an object is $p(\mathbf{M}(T,U)) \cong p(\flat(T \multimap U))$. This motivates the following definition:

Definition 3.7. The functor $! : \overline{\overline{\mathbf{M}}} \to \overline{\overline{\mathbf{M}}}$ is defined by $! = p \circ \flat$. Since p and $\flat$ are adjoints, the functor ! is a comonad on the category $\overline{\overline{\mathbf{M}}}$ [6]. We call it the *boxing comonad*. It is equipped with a natural transformation **force** : $!A \to A$ as well as a lifting operation

$$\frac{f : !A \to B}{\mathbf{lift}(f) : !A \to !B.}$$

Remark 3.8. From (5), we have an isomorphism $\mathbf{box} : !(T \multimap U) \to p(\mathbf{M}(T,U))$ for simple M-objects T and U. We denote its inverse by **unbox**. It will also be convenient to consider an operation $\mathbf{apply} : p(\mathbf{M}(T,U)) \otimes T \to U$, which definable from **unbox** using **force** and the monoidal closed structure.

Remark 3.9. In the category $\overline{\overline{\mathbf{M}}}$, every parameter object P is isomorphic to an object of the form $!A$. We can therefore generalize the lift operation to all parameter objects:

$$\frac{f : P \to B}{\mathbf{lift}(f) : P \to !B.}$$

Theorem 3.10. *The category* $\overline{\overline{\mathbf{M}}}$*, together with the adjunction given by* $p : \mathbf{Set} \to \overline{\overline{\mathbf{M}}}$ *and* $\flat : \overline{\overline{\mathbf{M}}} \to \mathbf{Set}$*, forms a linear-non-linear model in the sense of Benton [1, 7].*

3.8 Meta-operations on circuits

In Remark 3.1, we considered that the category $\mathbf{M}$ may be equipped with additional meta-operations on generalized circuits, such as

$$\begin{array}{lcl} \mathit{size} & : & \mathbf{M}(T,U) \to \mathbb{N}, \\ \mathit{print} & : & \mathbf{M}(T,U) \to \text{Document}, \\ \mathit{invert} & : & \mathbf{M}(T,T) \to \mathbf{M}(T,T) \end{array}$$

Note that these operations are *external* to the category $\mathbf{M}$, i.e., they are not morphisms of $\mathbf{M}$, but set-theoretic functions. Using the boxing monad "!", these meta-operations can be *internalized* in the category $\overline{\overline{\mathbf{M}}}$. Namely, given the above operations, the following are morphisms of $\overline{\overline{\mathbf{M}}}$, where T,U are simple M-objects:

$$\begin{array}{lcl} \mathbf{size} & = & !(T \multimap U) \xrightarrow{\mathbf{box}} p(\mathbf{M}(T,U)) \xrightarrow{p(size)} p(\mathbb{N}) \xrightarrow{\cong} \mathbf{nat} \\ \mathbf{print} & = & !(T \multimap U) \xrightarrow{\mathbf{box}} p(\mathbf{M}(T,U)) \xrightarrow{p(print)} p(\text{Document}) \\ \mathbf{invert} & = & !(T \multimap T) \xrightarrow{\mathbf{box}} p(\mathbf{M}(T,T)) \xrightarrow{p(invert)} p(\mathbf{M}(T,T)) \xrightarrow{\mathbf{unbox}} !(T \multimap T) \end{array}$$

This precisely captures our intuition that "boxed" circuits are concrete data that can be operated upon at circuit generation time.

4 Towards a circuit description language

4.1 Overview

As explained in the introduction, we will design a programming language for circuit description by making the language fit the denotational model. We start with a brief overview of all the features that will be added to the language. The formal definitions will follow in Section 4.3.

Since the category $\overline{\overline{\mathbf{M}}}$ is symmetric monoidal closed with coproducts, a standard linear lambda calculus with sum types can be interpreted in it. Basic types such as **bool**, **bit** and **qubit** (the latter two if present in the category $\mathbf{M}$ of generalized circuits) can also be added to the language, along with the associated terms (such as **true**, **false**, an if-then-else construction, and any basic gates that are present in the category $\mathbf{M}$). Moreover, certain inductive types such as $\mathbf{list}(A)$ and **nat** exist in the model and therefore can be added to the language. The language can further be equipped with a type operation "!" and terms "lift", "force", "box", and "apply", arising from their categorical counterparts in Section 3.7.

Certain types of the language will be designated as parameter types, simple types, and/or M-types. Their interpretation in the model will of course be parameter objects, simple objects, and/or M-objects, respectively. In Quipper, M-types were called *quantum data types*, but that name does not seem appropriate in the context of generalized circuits. We have not come up with a better name for them and are stuck with "M-types" for now.

Remark 4.1. Our claim that the resulting programming language is a language for describing families of circuits is justified by the following observation. Suppose $\Phi \vdash N : T \multimap U$ is a valid typing judgement, where Φ is a parameter context (i.e., a context where each type is a parameter type), and T and U are simple M-types. Then the interpretation of this typing judgement will be a morphism $[\![N]\!] : p(X) \to [\![T]\!] \multimap [\![U]\!]$ of the category $\overline{\overline{\mathbf{M}}}$, where $p(X) = [\![\Phi]\!]$ is a parameter object and $[\![T]\!]$ and $[\![U]\!]$ are simple M-objects. We have:

$$\overline{\overline{\mathbf{M}}}(p(X), [\![T]\!] \multimap [\![U]\!]) \;\cong\; \mathbf{Set}(X, \flat([\![T]\!] \multimap [\![U]\!])) \;\cong\; \mathbf{Set}(X, \mathbf{M}([\![T]\!], [\![U]\!])),$$

where the first isomorphism uses the fact that $\flat$ is the right adjoint of p, and the second isomorphism arises from (5). Therefore, the interpretation of N literally yields a function from X to $\mathbf{M}([\![T]\!], [\![U]\!])$, i.e., a parameterized family of generalized circuits.

4.2 Labelled circuits

As explained in Section 3.1, we assume that a symmetric monoidal category $\mathbf{M}$ of *generalized circuits* has been fixed once and for all. To make it more convenient for our programming language to manipulate morphisms of $\mathbf{M}$, it is useful to equip $\mathbf{M}$ with an additional *labelling structure*, which we now define.

Let us assume a fixed given set $\mathscr{W}$ of *wire types*, together with an interpretation function $[\![-]\!] : \mathscr{W} \to |\mathbf{M}|$, assigning an object of $\mathbf{M}$ to every wire type. In practical examples, we will sometimes assume that the set $\mathscr{W}$ contains two wire types called **bit** and **qubit**, but in general the set of wire types is arbitrary. We often denote wire types by Greek letters such as α and β. Let $\mathscr{L}$ be a fixed countably infinite set of *labels*, which we assume to be totally ordered. We often denote labels by the letters ℓ or k. A *label context* is a function from some finite set of labels to wire types. We write label contexts as $Q = \ell_1 : \alpha_1, \ldots, \ell_n : \alpha_n$. To each label context $Q = \ell_1 : \alpha_1, \ldots, \ell_n : \alpha_n$, we associate the object $[\![Q]\!] = [\![\alpha_1]\!] \otimes \ldots \otimes [\![\alpha_n]\!]$ of $\mathbf{M}$, where $\ell_1 < \ell_2 < \ldots < \ell_n$. In case $Q = \emptyset$, we set $[\![Q]\!] = I$.

Definition 4.2 (Labelled circuits)**.** Let $\mathbf{M}$ be a given symmetric monoidal category, and let $\mathscr{W}$, $\mathscr{L}$, and the function $[\![-]\!] : \mathscr{W} \to |\mathbf{M}|$ be given as above. The symmetric monoidal category $\mathbf{M}_{\mathscr{L}}$ of *labelled circuits* is defined as follows:

- The objects of $\mathbf{M}_{\mathscr{L}}$ are label contexts.
- A morphism $f : Q \to Q'$ in $\mathbf{M}_{\mathscr{L}}$ is by definition a morphism $f : [\![Q]\!] \to [\![Q']\!]$.

Identities and composition are defined so that $[\![-]\!] : \mathbf{M}_{\mathscr{L}} \to \mathbf{M}$ is a full and faithful functor. We equip $\mathbf{M}_{\mathscr{L}}$ with the unique (up to natural isomorphism) symmetric monoidal structure making this functor symmetric monoidal.

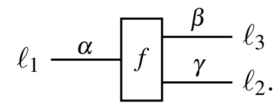
$$
\begin{array}{lrcl}
\text{Types} & A,B & ::= & \alpha \mid 0 \mid A+B \mid I \mid A\otimes B \mid A \multimap B \mid\, !A \mid \mathbf{nat} \mid \mathbf{list}\,A \mid \mathrm{Circ}(T,U) \\
\text{Parameter types} & P,R & ::= & 0 \mid P+R \mid I \mid P\otimes R \mid\, !A \mid \mathbf{nat} \mid \mathbf{list}\,P \mid \mathrm{Circ}(T,U) \\
\text{Simple M-types} & T,U & ::= & \alpha \mid I \mid T\otimes U
\end{array}
$$

Table 1: The types of Proto-Quipper-M

$$
\begin{array}{lrcl}
\text{Terms} & M,N & ::= & x \mid \ell \mid c \mid \mathrm{let}\,x = M\,\mathrm{in}\,N \\
 & & & \mid \Box_A M \mid \mathrm{left}_{A,B} M \mid \mathrm{right}_{A,B} M \mid \mathrm{case}\,M\,\mathrm{of}\,\{\mathrm{left}\,x \to N \mid \mathrm{right}\,y \to P\} \\
 & & & \mid * \mid M;N \mid \langle M,N\rangle \mid \mathrm{let}\,\langle x,y\rangle = M\,\mathrm{in}\,N \mid \lambda x^A.M \mid MN \\
 & & & \mid \mathrm{lift}\,M \mid \mathrm{force}\,M \mid \mathrm{box}_T\,M \mid \mathrm{apply}(M,N) \mid (\vec{\ell},C,\vec{\ell}') \\
\text{Label tuples} & \vec{\ell},\vec{k} & ::= & \ell \mid * \mid \langle \vec{\ell},\vec{k}\rangle \\
\text{Values} & V,W & ::= & x \mid \ell \mid c \mid \mathrm{left}_{A,B} V \mid \mathrm{right}_{A,B} V \mid * \mid \langle V,W\rangle \mid \lambda x^A.M \mid \mathrm{lift}\,M \mid (\vec{\ell},C,\vec{\ell}')
\end{array}
$$

Table 2: The terms of Proto-Quipper-M

Note that if Q and Q' have disjoint domains, then $Q\otimes Q' \cong Q\cup Q'$, i.e., the tensor of disjoint label contexts is given by their union. Two label contexts can always be made disjoint up to isomorphism by renaming their labels. We can visualize the morphisms of $\mathbf{M}_{\mathscr{L}}$ as generalized circuits with labelled and typed inputs and outputs, for example



Remark 4.3. Although it was convenient to assume that the set of labels is totally ordered, Definition 4.2 is actually independent of the particular order chosen. Specifically, if we change the ordering of the labels, the resulting category $\mathbf{M}_{\mathscr{L}}$ will be isomorphic to the original one.

4.3 The syntax of Proto-Quipper-M

The *types* of Proto-Quipper-M are shown in Table 1. Here, α ranges over the set $\mathscr{W}$ of wire types. $\mathrm{Circ}(T,U)$ is the type of generalized circuits with inputs T and outputs U. Denotationally, this type is the same as $!(T\multimap U)$, but it will receive special treatment in the operational semantics.

The *terms* of Proto-Quipper-M are shown in Table 2. Here, x ranges over a countable set of *variables*, ℓ ranges over the set $\mathscr{L}$ of labels, and c ranges over a given set of *constants*. We assume that these sets are pairwise disjoint. A label is a symbolic representation of a circuit state: it is a pointer to an output of a labelled circuit. Constants have a fixed interpretation, which depends on the chosen category $\mathbf{M}$. For example, it will be convenient to assume constants for built-in gates such as $H : \mathbf{qubit} \multimap \mathbf{qubit}$. If the category $\mathbf{M}$ is equipped with meta-operations as in Remark 3.1, then these can be represented by constants such as $\mathbf{size} : \mathrm{Circ}(T,U) \to \mathbf{nat}$, $\mathbf{invert} : \mathrm{Circ}(T,U) \to \mathrm{Circ}(U,T)$, and so on. Most of the other terms follow standard lambda calculus conventions. The operator $\Box_A : 0 \to A$ represents the unique function from to the empty type to any type A. The term $(\vec{\ell},C,\vec{\ell}')$ represents a *boxed circuit*, i.e., a value of type $\mathrm{Circ}(T,U)$. Specifically, C is a morphism of the category $\mathbf{M}_{\mathscr{L}}$, representing a generalized circuit, and $\vec{\ell}$ and $\vec{\ell}'$ are label tuples, establishing an interface between the inputs and outputs of C and the types T and U. Terms of the form $(\vec{\ell},C,\vec{\ell}')$ are not intended to be written directly by users of the programming language. Rather, such terms represent values that are computed by the programming language.

A *variable context* is a function from a finite set of variables to types. We write a variable context as $\Gamma = x_1 : A_1,\ldots,x_n : A_n$. A variable context in which all types are parameter types is called a *parameter context*; we sometimes denote parameter contexts by Φ. A *label context* was already defined in Section 4.2, and is a function from a finite set of labels to wire types. As usual, we write Γ,Δ to denote the union of contexts, provided that they have disjoint domain.

A *typing judgement* is of the form $\Gamma;Q \vdash M : A$, and it informally means that if the variables and labels have the types declared in Γ and Q, then M is well-typed of type A. A selection of the typing rules for Proto-Quipper-M is shown in Table 3. Most of the typing rules resemble the standard rules for a linear lambda calculus; we

$$\frac{}{\Phi, x:A;\emptyset \vdash x:A}\ (var) \qquad \frac{}{\Phi;\ell:\alpha \vdash \ell:\alpha}\ (label) \qquad \frac{}{\Phi;\emptyset \vdash c:A_c}\ (const)$$

$$\frac{\Gamma, x:A;Q \vdash M:B}{\Gamma;Q \vdash \lambda x^A.M : A \multimap B}\ (abs) \qquad \frac{\Phi,\Gamma_1;Q_1 \vdash M : A \multimap B \quad \Phi,\Gamma_2;Q_2 \vdash N:A}{\Phi,\Gamma_1,\Gamma_2;Q_1,Q_2 \vdash MN : B}\ (app)$$

$$\frac{\Phi;\emptyset \vdash M:A}{\Phi;\emptyset \vdash \mathrm{lift}\, M : !A}\ (lift) \qquad \frac{\Gamma;Q \vdash M : !A}{\Gamma;Q \vdash \mathrm{force}\, M : A}\ (force)$$

$$\frac{\Gamma;Q \vdash M : !(T \multimap U)}{\Gamma;Q \vdash \mathrm{box}_T\, M : \mathrm{Circ}(T,U)}\ (box) \qquad \frac{\Phi,\Gamma_1;Q_1 \vdash M : \mathrm{Circ}(T,U) \quad \Phi,\Gamma_2;Q_2 \vdash N:T}{\Phi,\Gamma_1,\Gamma_2;Q_1,Q_2 \vdash \mathrm{apply}(M,N) : U}\ (apply)$$

$$\frac{\emptyset;Q \vdash \vec{\ell}:T \quad \emptyset;Q' \vdash \vec{\ell}':U \quad C \in \mathbf{M}_{\mathscr{L}}(Q,Q')}{\Phi;\emptyset \vdash (\vec{\ell},C,\vec{\ell}') : \mathrm{Circ}(T,U)}\ (circ)$$

Table 3: The typing rules of Proto-Quipper-M (excerpt)

will comment on a few particular features of the type system. Note that in the typing rules, Φ stands for a parameter context, whereas Γ denotes an arbitrary variable context (which can contain both parameter types and non-parameter types). There is no formal distinction between these two kinds of contexts, so it is entirely possible to have a type derivation where a given type is part of Φ in one rule and part of Γ in another. Let us call a variable whose type is not a parameter type a *linear* variable. The type system enforces that labels and linear variables are used exactly once, whereas parameters may be used any number of times or not at all. In the typing rule for constants, we have assumed that each constant c is equipped with a fixed type A_c.

4.4 Categorical semantics

Because of the preparatory work we did in Section 3, the categorical semantics of Proto-Quipper-M is now straightforward. The semantics associates to each type A an object $[\![A]\!]$ of the category $\overline{\overline{\mathbf{M}}}$ in the obvious way: the interpretation $[\![\alpha]\!]$ of a wire type is assumed given, and each connective is interpreted as "itself", for example, $[\![0]\!] = 0$, $[\![A+B]\!] = [\![A]\!] + [\![B]\!]$, and so on. We also set $[\![\mathrm{Circ}(T,U)]\!] = p(\mathbf{M}([\![T]\!],[\![U]\!]))$. If $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ is a typing context, we write $[\![\Gamma]\!] = [\![A_1]\!] \otimes \ldots \otimes [\![A_n]\!]$.

Next, we associate a morphism $[\![\Gamma;Q \vdash M:A]\!] : [\![\Gamma]\!] \otimes [\![Q]\!] \to [\![A]\!]$ to each valid typing judgement. By abuse of notation, we sometimes denote this simply as $[\![M]\!]$. We assume that each constant c of type A_c is interpreted by a given fixed morphism $[\![c]\!] : I \to [\![A_c]\!]$. The interpretation of type derivations is defined by induction on the typing rules. For space reasons, we only show part of the interpretation in Table 4. The interpretation uses maps $\diamond : P \to I$ and $\Delta : P \to P \otimes P$, which exist in the category $\overline{\overline{\mathbf{M}}}$ whenever P is a parameter object. Each judgement of the form $Q \vdash \vec{\ell} : T$ induces an isomorphism $[\![\vec{\ell}]\!] : [\![Q]\!] \to [\![T]\!]$, from which we can define a morphism $f(\vec{\ell},C,\vec{\ell}') = [\![\vec{\ell}']\!] \circ C \circ [\![\vec{\ell}]\!]^{-1} : [\![T]\!] \to [\![U]\!]$. This is used in the last rule in Table 4.

5 Operational semantics

5.1 Evaluation rules

We define the operational semantics of Proto-Quipper-M as a big-step semantics [5]. A *configuration* is a pair (C,M) of a labelled circuit C and a term M. Recall that a labelled circuit is, by definition, a morphism of the category $\mathbf{M}_{\mathscr{L}}$. Intuitively, C is the circuit "currently being constructed" when the term M is run. A configuration is a *value configuration* if M is a value. Evaluation takes the form of an *evaluation relation* $(C,M) \Downarrow (C',V)$. Its intuitive meaning is: when the term M is evaluated in the context of a partially constructed circuit C, then it produces a circuit C' (obtained from C by appending zero or more gates) and a value V. We also define an *error relation* $(C,M) \Downarrow \mathrm{Error}$, meaning that the evaluation of M in the context of the circuit C produces a run-time error. Examples of run-time errors are:

- *Run-time type errors.* For example, evaluating an application MN, where M is not a function, or a projection $\pi_1 M$, when M is not a pair.

$$
\begin{array}{rcl}
[\![\Phi, x : A; \emptyset \vdash x : A]\!] & = & [\![\Phi]\!] \otimes [\![A]\!] \xrightarrow{\diamond\otimes\mathrm{id}} I \otimes [\![A]\!] \xrightarrow{\cong} A \\
[\![\Phi; \ell : \alpha \vdash \ell : \alpha]\!] & = & [\![\Phi]\!] \otimes [\![\alpha]\!] \xrightarrow{\diamond\otimes\mathrm{id}} I \otimes [\![\alpha]\!] \xrightarrow{\cong} \alpha \\
[\![\Phi; \emptyset \vdash \mathrm{lift}\, M : !A]\!] & = & [\![\Phi]\!] \xrightarrow{\mathbf{lift}[M]} ![\![A]\!] \\
[\![\Gamma; Q \vdash \mathrm{box}_T\, M : \mathrm{Circ}(T,U)]\!] & = & [\![\Gamma]\!] \otimes [\![Q]\!] \xrightarrow{[M]} !([\![T]\!] \multimap [\![U]\!]) \xrightarrow{\cong} p(\mathbf{M}([\![T]\!], [\![U]\!])) \\
[\![\Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash \mathrm{apply}(M,N) : U]\!] & = & [\![\Phi]\!] \otimes [\![\Gamma_1]\!] \otimes [\![\Gamma_2]\!] \otimes [\![Q_1]\!] \otimes [\![Q_2]\!] \\
 & & \xrightarrow{\Delta\otimes\mathrm{id}} [\![\Phi]\!] \otimes [\![\Phi]\!] \otimes [\![\Gamma_1]\!] \otimes [\![\Gamma_2]\!] \otimes [\![Q_1]\!] \otimes [\![Q_2]\!] \\
 & & \xrightarrow{\cong} ([\![\Phi]\!] \otimes [\![\Gamma_1]\!] \otimes [\![Q_1]\!]) \otimes ([\![\Phi]\!] \otimes [\![\Gamma_2]\!] \otimes [\![Q_2]\!]) \\
 & & \xrightarrow{[M]\otimes[N]} p(\mathbf{M}([\![T]\!], [\![U]\!])) \otimes [\![T]\!] \xrightarrow{\mathbf{apply}} [\![U]\!] \\
[\![\Phi; \emptyset \vdash (\vec{\ell}, C, \vec{\ell}') : \mathrm{Circ}(T,U)]\!] & = & [\![\Phi]\!] \xrightarrow{\diamond} I \xrightarrow{p(f(\vec{\ell},C,\vec{\ell}'))} p(\mathbf{M}([\![T]\!], [\![U]\!]))
\end{array}
$$

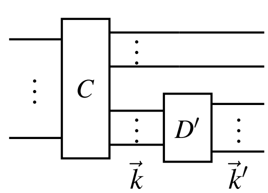
Table 4: The interpretation of type derivations (excerpt)

- *Unbound variable or label.* For example, using a variable x that has not been defined, or trying to append a gate to a wire ℓ that does not exist.
- *Cloning errors.* For example, trying to append a 2-input gate to wires ℓ and ℓ', where $\ell = \ell'$.

The evaluation and error relations are defined recursively. A selection of the evaluation rules are shown in Table 5, using some notations which we now explain. As usual, $M[V/x]$ denotes capture-avoiding substitution, i.e., the result of replacing the variable x by the value V in the term M. In the hypotheses of several rules, we have used the notation "$(C,M) \Downarrow$ otherwise". This is an abbreviation for $(C,M) \Downarrow (C',W)$, where W is not of one of the explicit forms mentioned in a previous rule for the same configuration. For example, in the rules for force, $(C,M) \Downarrow$ otherwise means $(C,M) \Downarrow (C',W)$ where W is not of the form lift M'. All such "otherwise" cases yield run-time type errors.

Most of the evaluation rules are those of a standard call-by-value lambda calculus. The rules that do all of the interesting "work" of Proto-Quipper-M are those for "box" and "apply". Namely, these rules are responsible for the construction of circuits. They rely on two auxiliary functions, which we now explain. The rules for "box" use a function *freshlabels*. Given a simple M-type T, the operation *freshlabels*(T) returns a pair $(Q, \vec{\ell})$ of a label context and a label tuple such that $Q \vdash \vec{\ell} : T$. Moreover, the labels in $\vec{\ell}$ are chosen to be *fresh*, which means that they do not occur in N.

The rule for "apply" uses a function *append*, defined as follows. First, let us say that two boxed circuits $(\vec{\ell}, D, \vec{\ell})$ and $(\vec{k}, D', \vec{k}')$ are *equivalent*, in symbols $(\vec{\ell}, D, \vec{\ell}) \cong (\vec{k}, D', \vec{k}')$, if they only differ by a renaming of labels. Given labelled circuits $C : Q_0 \to Q_1$ and $D : Q_2 \to Q_3$ and label tuples $\vec{k}$, $\vec{\ell}$, and $\vec{\ell}'$, the operation *append*$(C, \vec{k}, \vec{\ell}, D, \vec{\ell}')$ finds D' and $\vec{k}'$ such that $(\vec{k}, D', \vec{k}') \cong (\vec{\ell}, D, \vec{\ell}')$, and such that the labels in $\vec{k}'$ are fresh. It returns $(C', \vec{k}')$, where C' is the labelled circuit obtained by connecting the inputs of D' to the matching outputs of C like this:



(6)

5.2 Safety properties

The operational semantics satisfies the following safety properties: a well-typed configuration never produces a run-time error, and if it reduces to a value configuration, then the latter is well-typed of the same type. To make this more precise, we first define a notion of typing for configurations.

Definition 5.1. Let Q, Q' be label contexts, (C,M) a configuration, and A a type. We say that (C,M) is *well-typed* with input labels Q, output labels Q', and type A, in symbols $Q \vdash (C,M) : A; Q'$, if there exists Q'' disjoint from Q' such that $C : Q \to Q'' \cup Q'$ and $\emptyset; Q'' \vdash M : A$.

$$\frac{}{(C,x)\Downarrow \mathrm{Error}} \qquad \frac{}{(C,\ell)\Downarrow (C,\ell)} \qquad \frac{}{(C,c)\Downarrow (C,c)}$$

$$\frac{}{(C,\lambda x.M)\Downarrow (C,\lambda x.M)}$$

$$\frac{(C,M)\Downarrow (C',\lambda x.M') \quad (C',N)\Downarrow (C'',V) \quad (C'',M'[V/x])\Downarrow (C''',W)}{(C,MN)\Downarrow (C''',W)} \qquad \frac{(C,M)\Downarrow \text{otherwise}}{(C,MN)\Downarrow \mathrm{Error}}$$

$$\frac{}{(C,\mathrm{lift}\,M)\Downarrow (C,\mathrm{lift}\,M)} \qquad \frac{(C,M)\Downarrow (C',\mathrm{lift}\,M') \quad (C',M')\Downarrow (C'',V)}{(C,\mathrm{force}\,M)\Downarrow (C'',V)} \qquad \frac{(C,M)\Downarrow \text{otherwise}}{(C,\mathrm{force}\,M)\Downarrow \mathrm{Error}}$$

$$\frac{(C,M)\Downarrow (C',\mathrm{lift}\,N) \quad \mathit{freshlabels}(T)=(Q,\vec{\ell}) \quad (\mathrm{id}_Q,N\vec{\ell})\Downarrow (D,\vec{\ell}')}{(C,\mathrm{box}_T\,M)\Downarrow (C',(\vec{\ell},D,\vec{\ell}'))}$$

$$\frac{(C,M)\Downarrow (C',\mathrm{lift}\,N) \quad \mathit{freshlabels}(T)=(Q,\vec{\ell}) \quad (\mathrm{id}_Q,N\vec{\ell})\Downarrow \text{otherwise}}{(C,\mathrm{box}_T\,M)\Downarrow \mathrm{Error}} \qquad \frac{(C,M)\Downarrow \text{otherwise}}{(C,\mathrm{box}_T\,M)\Downarrow \mathrm{Error}}$$

$$\frac{(C,M)\Downarrow (C',(\vec{\ell},D,\vec{\ell}')) \quad (C',N)\Downarrow (C'',\vec{k}) \quad \mathit{append}(C'',\vec{k},\vec{\ell},D,\vec{\ell}')=(C''',\vec{k}')}{(C,\mathrm{apply}(M,N))\Downarrow (C''',\vec{k}')}$$

$$\frac{}{(C,(\vec{\ell},D,\vec{\ell}'))\Downarrow (C,(\vec{\ell},D,\vec{\ell}'))}$$

Table 5: The big-step semantics (excerpt)

Proposition 5.2 (Subject reduction)**.** *If $Q\vdash (C,M):A;Q'$ and $(C,M)\Downarrow (C',V)$, then $Q\vdash (C',V):A;Q'$.*
Proposition 5.3 (Error freeness)**.** *If $Q\vdash (C,M):A;Q'$, then $(C,M)\not\Downarrow \mathrm{Error}$.*

While the statement of these properties is succinct, the proofs are intricate and require a number of auxiliary lemmas, which we omit. Since our language does not have a recursion operator, we also have:
Proposition 5.4 (Termination)**.** *If $Q\vdash (C,M):A;Q'$, then there exists (C',V) such that $(C,M)\Downarrow (C',V)$.*

5.3 Soundness properties

While the safety properties of Section 5.2 relate the evaluation rules to the typing rules, the following *soundness properties* relate the evaluation rules to the categorical semantics. We first extend the categorical semantics from well-typed terms to well-typed configurations.
Definition 5.5. To each well-typed configuration $Q\vdash (C,M):A;Q'$, we associate a morphism $[\![(C,M)]\!]:[\![Q]\!]\to [\![A]\!]\otimes[\![Q']\!]$ of the category $\overline{\overline{\mathbf{M}}}$ as follows. By definition of well-typed configuration, there exists a unique Q'' such that $C:Q\to Q''\cup Q'$ and $\emptyset;Q''\vdash M:A$. Then

$$[\![(C,M)]\!]=[\![Q]\!]\xrightarrow{C}[\![Q''\cup Q']\!]\xrightarrow{\cong}[\![Q'']\!]\otimes[\![Q']\!]\xrightarrow{[\![M]\!]\otimes\mathrm{id}}[\![A]\!]\otimes[\![Q']\!]$$

Proposition 5.6 (Soundness)**.** *If $Q\vdash (C,M):A;Q'$ is a well-typed configuration and $(C,M)\Downarrow (C',V)$, then $[\![(C,M)]\!]=[\![(C',V)]\!]:[\![Q]\!]\to[\![A]\!]\otimes[\![Q']\!]$.*

The soundness property implies that the operational semantics coincides with the categorical semantics. An important special case arises if we have a closed term $M:\mathrm{Circ}(T,U)$. The categorical meaning of M is some generalized circuit $[\![M]\!]:[\![T]\!]\to[\![U]\!]$. Operationally, the term M evaluates to some boxed circuit $(\vec{\ell},D,\vec{\ell}')$, and soundness ensures that $[\![M]\!]=[\![(\vec{\ell},D,\vec{\ell}')]\!]$. Therefore, at *observable types* such as $\mathrm{Circ}(T,U)$, our evaluation rules are a constructive implementation of the categorical semantics. More generally, we have the following adequacy property.
Proposition 5.7 (Computational adequacy)**.** *If $\emptyset\vdash (C,M):A;\emptyset$ such that $[\![(C,M)]\!]=[\![(C',V)]\!]$ and A is an observable type, then $(C,M)\Downarrow (C',V)$ (possibly up to a renaming of labels in V).*

As promised in Remark 3.2, the proofs of the soundness and adequacy properties are independent of the choice of the category $\overline{\mathbf{M}}$. This justifies not having made such a choice in the first place.

6 Conclusions and future work

We systematically constructed a programming language for describing families of generalized circuits by first giving a categorical model and then defining the language to fit the model. The language has an operational semantics, and we proved safety, soundness, and adequacy properties showing that the computational and categorical meanings coincide.

A software implementation of Proto-Quipper-M is in progress and almost completed. In future work, we hope to extend the language to encompass successively larger sets of features of the original Quipper. For example, the current version of Proto-Quipper-M lacks a general recursion scheme, so that all programs are terminating (see Proposition 5.4). Adding recursion to the programming language is no problem at all, but how to add it to the categorical model while preserving soundness and adequacy is an open question.

Another question we hope to address in future work is the exact relationship between Proto-Quipper-M and the version of Proto-Quipper from Ross's thesis [9]. While these two languages have much in common, they differ in some important aspects: for example, Ross's Proto-Quipper uses a subtyping relation $!A <: A$ instead of explicit "lift" and "force" operators. This is convenient for programmers, but giving a categorical semantics for Ross's version of Proto-Quipper is left for future work.

7 Acknowledgements

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by the Air Force Office of Scientific Research, Air Force Material Command, USAF under Award No. FA9550-15-1-0331. This work was done in part while the authors were visiting the Simons Institute for the Theory of Computing.

References

[1] Nick Benton (1995): *A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models (Extended Abstract)*. In: *Proceedings of the 8th Workshop on Computer Science Logic, CSL'94, Selected Papers*, Springer Lecture Notes in Computer Science 933, pp. 121–135.

[2] Brian Day (1970): *On Closed Categories of Functors*. In: *Reports of the Midwest Category Seminar IV, Lecture Notes in Mathematics* 137, Springer, pp. 1–38.

[3] Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger & Benoît Valiron (2013): *Quipper: a Scalable Quantum Programming Language*. In: *Proceedings of the 34th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, ACM SIGPLAN Notices* 48(6), pp. 333–342, doi:10.1145/2499370.2462177. Also available from http://arxiv.org/abs/1304.3390.

[4] Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger & Benoît Valiron (2013): *The Quipper language*. Software implementation, available from http://www.mathstat.dal.ca/~selinger/quipper/.

[5] Gilles Kahn (1987): *Natural Semantics*. In: *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1987)*, Springer, pp. 22–39.

[6] S. Mac Lane (1998): *Categories for the Working Mathematician*, 2nd edition. Graduate Texts in Mathematics, Springer.

[7] Paul-André Melliès (2009): *Categorical semantics of linear logic*. In: *Interactive Models of Computation and Program Behaviour, Panoramas et Synthèses* 27, Société Mathématique de France, pp. 1–196.

[8] Jennifer Paykin, Robert Rand & Steve Zdancewic (2017): *QWIRE: A Core Language for Quantum Circuits*. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, ACM, New York, NY, USA, pp. 846–858, doi:10.1145/3009837.3009894.

[9] Neil J. Ross (2015): *Algebraic and Logical Methods in Quantum Computation*. Ph.D. thesis, Department of Mathematics and Statistics, Dalhousie University. Available from http://arxiv.org/abs/1510.02198.

[10] Peter Selinger & Benoît Valiron (2009): *Quantum Lambda Calculus*. In Simon Gay & Ian Mackie, editors: *Semantic Techniques in Quantum Computation*, chapter 4, Cambridge University Press, pp. 135–172.