A linear/non-linear model for a quantum circuit description language

Francisco Rios
(joint work with Peter Selinger)

Department of Mathematics and Statistics
Dalhousie University

Category Theory OctoberFest 2017
Carnegie Mellon University
October 28th–29th, 2017

What is this talk about?

It is about a categorical model of a small, but useful fragment of Quipper called *Proto-Quipper-M*.
Quipper is a functional programming language for quantum computing.
The fragment Proto-Quipper-M is

- a stand-alone programming language (i.e., not embedded in a host language),
- it has its own custom type system and semantics, and
- it is also more general than Quipper: it can *describe families of morphisms* in *any* symmetric monoidal category, of which quantum circuits are but one example.

What is this talk about?

It is about a categorical model of a small, but useful fragment of Quipper called *Proto-Quipper-M*.
Quipper is a functional programming language for quantum computing.
The fragment Proto-Quipper-M is

- a stand-alone programming language (i.e., not embedded in a host language),
- it has its own custom type system and semantics, and
- it is also more general than Quipper: it can *describe families of morphisms* in *any* symmetric monoidal category, of which quantum circuits are but one example.

What it Quipper?

It is a *circuit description language*. This means two things:

- the language can be used to construct quantum circuits in a structured way, essentially by applying one gate at a time.
- the completed circuits themselves become data, which can be stored in variables, and on which meta-operations (such as circuit transformations, gate counts, inversion, error correction, etc) can be performed.

Quipper is *practical*: it has been used to implement several large-scale quantum algorithms, generating circuits containing trillions of gates.

What it Quipper?

It is a *circuit description language*. This means two things:

- the language can be used to construct quantum circuits in a structured way, essentially by applying one gate at a time.
- the completed circuits themselves become data, which can be stored in variables, and on which meta-operations (such as circuit transformations, gate counts, inversion, error correction, etc) can be performed.

Quipper is *practical*: it has been used to implement several large-scale quantum algorithms, generating circuits containing trillions of gates.

What it Quipper?

It is a *circuit description language*. This means two things:

- the language can be used to construct quantum circuits in a structured way, essentially by applying one gate at a time.
- the completed circuits themselves become data, which can be stored in variables, and on which meta-operations (such as circuit transformations, gate counts, inversion, error correction, etc) can be performed.

Quipper is *practical*: it has been used to implement several large-scale quantum algorithms, generating circuits containing trillions of gates.

What it Quipper?

It is a *circuit description language*. This means two things:

- the language can be used to construct quantum circuits in a structured way, essentially by applying one gate at a time.
- the completed circuits themselves become data, which can be stored in variables, and on which meta-operations (such as circuit transformations, gate counts, inversion, error correction, etc) can be performed.

Quipper is *practical*: it has been used to implement several large-scale quantum algorithms, generating circuits containing trillions of gates.

Quipper's limitations

However, there are some drawbacks:

- Quipper is *not* type-safe (there are some well-typed programs that lead to run-time errors)
- Quipper has *no* formal semantics; its behavior is only defined by its implementation.

Proto-Quipper-M deals some of these issues!

Parameters and states

A *fundamental* concept that arises in Quipper is the *distinction between parameters and states*.
A Quipper program has two distinct run times:

- *circuit generation time*: when the circuits are built, and
- *circuit execution time*: when circuits are run.

A *parameter* is a value that is known at circuit generation time.
A *state* is a value that is known at circuit execution time.

Example: given a list of qubits $[q_1, \ldots, q_n]$, the length n of the list is typically known at circuit construction time, and is therefore a parameter, whereas the actual qubits are state.

Parameters and states

A *fundamental* concept that arises in Quipper is the *distinction between parameters and states*.
A Quipper program has two distinct run times:

- *circuit generation time*: when the circuits are built, and
- *circuit execution time*: when circuits are run.

A *parameter* is a value that is known at circuit generation time.
A *state* is a value that is known at circuit execution time.

Example: given a list of qubits $[q_1, \ldots, q_n]$, the length n of the list is typically known at circuit construction time, and is therefore a parameter, whereas the actual qubits are state.

Parameters and states

A *fundamental* concept that arises in Quipper is the *distinction between parameters and states*.
A Quipper program has two distinct run times:

- *circuit generation time*: when the circuits are built, and
- *circuit execution time*: when circuits are run.

A *parameter* is a value that is known at circuit generation time.
A *state* is a value that is known at circuit execution time.

Example: given a list of qubits $[q_1, \ldots, q_n]$, the length n of the list is typically known at circuit construction time, and is therefore a parameter, whereas the actual qubits are state.

Key observation

A state can depend on a parameter, but since states are not known at circuit generation time, parameters cannot be a function of states!

Enforcing this condition *soundly* is one of the guiding principles in the design of Proto-Quipper-M.
We devise Proto-Quipper-M from the ground up by

- *first* giving a categorical model of parameters and states, and
- *then* defining the language to fit the model.

Some *advantages* of this approach:

- Our programming language is almost *"correct by design."*
- It can *describe families of morphisms* of an *arbitrary* symmetric monoidal category, rather than just quantum circuits.

Key observation

A state can depend on a parameter, but since states are not known at circuit generation time, parameters cannot be a function of states!

Enforcing this condition *soundly* is one of the guiding principles in the design of Proto-Quipper-M.
We devise Proto-Quipper-M from the ground up by

- *first* giving a categorical model of parameters and states, and
- *then* defining the language to fit the model.

Some *advantages* of this approach:

- Our programming language is almost *"correct by design."*
- It can *describe families of morphisms* of an *arbitrary* symmetric monoidal category, rather than just quantum circuits.

Key observation

A state can depend on a parameter, but since states are not known at circuit generation time, parameters cannot be a function of states!

Enforcing this condition *soundly* is one of the guiding principles in the design of Proto-Quipper-M.
We devise Proto-Quipper-M from the ground up by

- *first* giving a categorical model of parameters and states, and
- *then* defining the language to fit the model.

Some *advantages* of this approach:

- Our programming language is almost *"correct by design."*
- It can *describe families of morphisms* of an *arbitrary* symmetric monoidal category, rather than just quantum circuits.

A *cartesian* model of parameters and states

$\mathbf{Set}^{\mathbf{2}^{op}}$ model is a *simplified* categorical model of parameters and states!

- This model is *cartesian*, 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.

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

Let us take a more *concrete* look at the model $\mathbf{Set}^{2^{op}}$.

- $Obj(\mathbf{Set}^{2^{op}})$: $A = (A_0, A_1, a)$, where A_0, A_1 are sets and $a : A_1 \to A_0$ is a function.
- $Mor(\mathbf{Set}^{2^{op}})$: $f : A \to B$ is a commutative diagram

$$\begin{array}{ccc} A_1 & \xrightarrow{f_1} & B_1 \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle b} \\ A_0 & \xrightarrow{f_0} & B_0. \end{array} \tag{1}$$

- For each $x \in A_0$, we define the *fiber* of A over x:

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

- We call the elements of A_0 *parameters* and the elements of A_x *states*.

Some observations on $\mathbf{Set}^{2^{op}}$

- An object $A = (A_0, A_1, a)$ describes a family of sets: up to isomorphism, A is uniquely determined by the family $(A_x)_{x \in A_0}$.
- The elements of A_1 can be identified with pairs (x, s), where $x \in A_0$ and $s \in A_x$, the *generalized elements* of A.
- *Key point*: 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"!

States may depend on parameters, but not viceversa!

To see this, just consider the effect of a morphism $f : A \to B$ on a *parameter-state* pair (x, s):

- Let $y = f_0(x)$ and $t = f_1(s)$. Then $y \in B_0$ and $t \in B_1$.
- By the commutativity of (1), $y = f_0(x) = f_0(a(s)) = b(f_1(s)) = b(t)$. So, $y = b(t)$, i.e., $t \in B_y = B_{f_0(x)}$. Thus, for each $x \in A_0$, $f_1 : A_1 \to B_1$ restricts to a function $f_x : A_x \to B_{f_0(x)}$.
- Note that t is a function of both x and s, because $t = f_x(s)$.
- *Therefore, states may depend on parameters and states.*
- Now, y is only a function of x, because $y = f_0(x)$.
- *Therefore, parameters may not depend on states.*

States may depend on parameters, but not viceversa!

To see this, just consider the effect of a morphism $f : A \to B$ on a *parameter-state* pair (x, s):

- Let $y = f_0(x)$ and $t = f_1(s)$. Then $y \in B_0$ and $t \in B_1$.
- By the commutativity of (1), $y = f_0(x) = f_0(a(s)) = b(f_1(s)) = b(t)$. So, $y = b(t)$, i.e., $t \in B_y = B_{f_0(x)}$. Thus, for each $x \in A_0$, $f_1 : A_1 \to B_1$ restricts to a function $f_x : A_x \to B_{f_0(x)}$.
- Note that t is a function of both x and s, because $t = f_x(s)$.
- *Therefore, states may depend on parameters and states.*
- Now, y is only a function of x, because $y = f_0(x)$.
- *Therefore, parameters may not depend on states.*

Some objects and morphisms in $\mathbf{Set}^{2^{op}}$

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:

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

Note:

- 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".

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

$$\begin{array}{ccc} 2 & \xrightarrow{\mathrm{id}} & 2 \\ {\scriptstyle \mathrm{id}}\downarrow & & \downarrow{\scriptstyle u} \\ 2 & \xrightarrow{u} & 1, \end{array} \tag{3}$$

and it satisfies $f(0,0) = (*,0)$ and $f(1,1) = (*,1)$.

What is the difference between **bool** and **bit**?

On the other hand, there exists *no* morphism $g : \mathbf{bit} \to \mathbf{bool}$ mapping false to false and true to true: the diagram

$$\begin{array}{ccc} 2 & \xrightarrow{\mathrm{id}} & 2 \\ {\scriptstyle u}\downarrow & & \downarrow{\scriptstyle \mathrm{id}} \\ 1 & \xrightarrow{?} & 2 \end{array} \tag{4}$$

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 states!

Parameter and state objects in $\mathbf{Set}^{2^{op}}$

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);
- an object A is called a *state object* or *simple* if A_0 is a singleton.

Note:

- **bool** is a parameter object and **bit** is a state object.

So, what can we interpret in $\mathbf{Set}^{2^{op}}$?

- Since the category $\mathbf{Set}^{2^{op}}$ is cartesian closed, we can interpret the simply-typed lambda calculus in it.
- Also, we can add base types such as **bool** and **bit**, and basic operations such as **init** : **bool** $\to$ **bit**, all of which have obvious interpretations in the model.
- Moreover, $\mathbf{Set}^{2^{op}}$ is co-complete which allows us to interpret sum types as well as inductive data types such as **list** (A) using initial algebra semantics.

So, we can model a simple lambda calculus for the description of boolean circuits.

The model $\mathbf{Set}^{2^{op}}$ is good, but not good enough!

- The model $\mathbf{Set}^{2^{op}}$ is useful for formalizing some basic intuitions about parameters and state.
- *However*, it is not yet an adequate model for describing families of *quantum* circuits. The main issue is that the model is *cartesian*!
  - There exist morphisms $\Delta_A : A \to A \times A$ 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.

The model $\mathbf{Set}^{2^{op}}$ is good, but not good enough!

- The model $\mathbf{Set}^{2^{op}}$ is useful for formalizing some basic intuitions about parameters and state.
- *However*, it is not yet an adequate model for describing families of *quantum* circuits. The main issue is that the model is *cartesian*!
  - There exist morphisms $\Delta_A : A \to A \times A$ 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.

How do we generalize $\mathbf{Set}^{2^{op}}$ to a monoidal setting?

Observe:

- $Obj(\mathbf{Set}^{2^{op}}) : A = (A_0, (A_x)_{x \in A_0})$, where A_0 is a set and $(A_x)_{x \in A_0}$ is a family of sets.
- $Mor(\mathbf{Set}^{2^{op}}) : f : A \to B$ 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 each A_x to be an object of a monoidal category instead of a set and each $f_x : A_x \to B_{f_0(x)}$ to be a morphism instead of a function of such category!

How do we generalize $\mathbf{Set}^{2^{op}}$ to a monoidal setting?

Observe:

- $Obj(\mathbf{Set}^{2^{op}}) : A = (A_0, (A_x)_{x \in A_0})$, where A_0 is a set and $(A_x)_{x \in A_0}$ is a family of sets.
- $Mor(\mathbf{Set}^{2^{op}}) : f : A \to B$ 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 each A_x to be an object of a monoidal category instead of a set and each $f_x : A_x \to B_{f_0(x)}$ to be a morphism instead of a function of such category!

The category **M**: generalized circuits

Before designing a circuit description language, we should be more precise about what we mean by a "circuit"!

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 assume that a symmetric monoidal category **M** is given call its morphisms *generalized circuits*.
- From this point of view, Proto-Quipper-M is simply a language for describing families of morphisms of **M**.

We will regard the morphisms of **M** as *concrete* data.

The category **M**: generalized circuits

Before designing a circuit description language, we should be more precise about what we mean by a "circuit"!
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 assume that a symmetric monoidal category **M** is given call its morphisms *generalized circuits*.
- From this point of view, Proto-Quipper-M is simply a language for describing families of morphisms of **M**.

We will regard the morphisms of **M** as *concrete* data.

The category $\overline{\mathbf{M}}$: states

We let $\overline{\mathbf{M}}$ be some symmetric monoidal closed, product-complete category in which $\mathbf{M}$ can be fully embedded.

- This can always be done.
- We do not specify any particular way of constructing $\overline{\mathbf{M}}$.

We will regard the category $\overline{\mathbf{M}}$ as *abstract*:

- It is *monoidal closed*, so we will be able to form higher-order objects such as $(A \multimap B \otimes C) \multimap D$.
- We do not imagine morphisms between such higher-order objects as being concrete things that can be printed, measured, etc.
- Instead, the higher-order structure only exists as a kind of "*scaffolding*" to support lower-order concrete operations.

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

We now define the category $\overline{\overline{\mathbf{M}}}$: *a model for parameters and states*.

- $\overline{\overline{\mathbf{M}}}$ will be the carrier of the categorical semantics of our circuit description language.

Definition

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 $\overline{\mathbf{M}}$.
- 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 rich in structure. To begin, $\overline{\overline{\mathbf{M}}}$ is the free coproduct completion of $\overline{\mathbf{M}}$.

Properties of $\overline{\overline{\mathbf{M}}}$

What is perhaps more surprising is that it is also monoidal closed.

Proposition

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

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

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 $\overline{\mathbf{M}}$.

Parameter and state objects in $\overline{\overline{\mathbf{M}}}$

Now parameter and state objects can be defined analogously to those of $\mathbf{Set}^{\mathbf{2}^{op}}$:

- 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 $A_0 \cong 1$.
- An object $A \in \overline{\overline{\mathbf{M}}}$ is an *M-object* if every fiber belongs to the category $\mathbf{M}$

Some canonical functors and basic types

Note that we have the following functors:

$$\mathbf{Set} \xrightarrow{p} \overline{\overline{\mathbf{M}}}, \qquad \mathbf{M} \xhookrightarrow{i} \overline{\mathbf{M}} \xhookrightarrow{j} \overline{\overline{\mathbf{M}}},$$

where $p(X) = (X, (I)_x)$, i is the canonical inclusion, and $j(A) = (1, (A))$.
The properties of $\overline{\overline{\mathbf{M}}}$ guarantee the existence of useful objects and morphisms:

- a parameter object $\mathbf{bool} = I + I$,
- morphisms $\mathbf{true}, \mathbf{false} : I \to \mathbf{bool}$,
- an object $\mathbf{nat} = (\mathbb{N}, (I)_n)$, indeed there is a parameter object $p(X)$ corresponding to every set X, arising from the functor $p : \mathbf{Set} \to \overline{\overline{\mathbf{M}}}$,
- an object $\mathbf{list}\,(A)$, for each $A \in \overline{\overline{\mathbf{M}}}$, the type of lists of A.

A relevant adjunction

Proposition

The functor $p : \mathbf{Set} \to \overline{\overline{\mathbf{M}}}$ *has a right adjoint* $\flat : \overline{\overline{\mathbf{M}}} \to \mathbf{Set}$ *given by*

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

Note that for simple M-objects T and U, we have

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

So $\flat(T \multimap U)$ is just a set of generalized circuits.
Also, 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

$$p(\mathbf{M}(T, U)) \cong p(\flat(T \multimap U)).$$

The boxing comonad

This motivates the following definition:

Definition

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 $\overline{\overline{\mathbf{M}}}$. We call it the *boxing comonad*. It is equipped with a natural transformation $\mathbf{force} : !A \to A$

Some useful morphisms:

- 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**.

Towards a circuit description language

Here we just give a brief *overview* of some of the most relevant features of the language:

- 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** can also be added to the language, along with the associated terms (such as **true**, **false**, and any basic gates that are present in the category **M**).

Towards a circuit description language

Here we just give a brief *overview* of some of the most relevant features of the language:

- 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** can also be added to the language, along with the associated terms (such as **true**, **false**, and any basic gates that are present in the category **M**).

Towards a circuit description language

- Moreover, since certain inductive types such as **list**(A) and **nat** exist in the model, we can add them to the language.
- Also the language can be equipped with a type operation "!" and terms "force", "box", and "unbox", arising from their categorical counterparts introduced earlier.
- The language has parameter types, simple types, and M-types, and their interpretation in the model will of course be parameter objects, simple objects, and M-objects, respectively.

Key point!

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 judgment, where Φ is a parameter context, and T and U are simple M-types.
- Then the interpretation of this judgement will be a morphism $[\![N]\!] : p(X) \to [\![T]\!] \multimap [\![U]\!]$ of $\overline{\overline{\mathbf{M}}}$, where $p(X) = [\![\Phi]\!]$ is a parameter object and $[\![T]\!]$ and $[\![U]\!]$ are simple M-objects.
- And thus 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!*

Key point!

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 judgment, where Φ is a parameter context, and T and U are simple M-types.
- Then the interpretation of this judgement will be a morphism $[\![N]\!] : p(X) \to [\![T]\!] \multimap [\![U]\!]$ of $\overline{\overline{\mathbf{M}}}$, where $p(X) = [\![\Phi]\!]$ is a parameter object and $[\![T]\!]$ and $[\![U]\!]$ are simple M-objects.
- And thus 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!*

Key point!

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 judgment, where Φ is a parameter context, and T and U are simple M-types.
- Then the interpretation of this judgement will be a morphism $[\![N]\!] : p(X) \to [\![T]\!] \multimap [\![U]\!]$ of $\overline{\overline{\mathbf{M}}}$, where $p(X) = [\![\Phi]\!]$ is a parameter object and $[\![T]\!]$ and $[\![U]\!]$ are simple M-objects.
- And thus 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!*

Key point!

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 judgment, where Φ is a parameter context, and T and U are simple M-types.
- Then the interpretation of this judgement will be a morphism $[\![N]\!] : p(X) \to [\![T]\!] \multimap [\![U]\!]$ of $\overline{\overline{\mathbf{M}}}$, where $p(X) = [\![\Phi]\!]$ is a parameter object and $[\![T]\!]$ and $[\![U]\!]$ are simple M-objects.
- And thus 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!*

Thank you for your attention!