

# Computational Adequacy for Substructural Lambda Calculi

Vladimir Zamdzhiev

Université de Lorraine, CNRS, Inria, LORIA, F 54000 Nancy, France

Substructural type systems, such as affine (and linear) type systems, are type systems which impose restrictions on copying (and discarding) of variables, and they have found many applications in computer science, including quantum programming. We describe one linear and one affine type systems and we formulate abstract categorical models for both of them which are sound and computationally adequate. We also show, under basic assumptions, that interpreting lambda abstractions via a monoidal closed structure (a popular method for linear type systems) necessarily leads to degenerate and inadequate models for call-by-value affine type systems with recursion. In our categorical treatment, a solution to this problem is clearly presented. Our categorical models are more general than linear/non-linear models used to study linear logic and we present a homogeneous categorical account of both linear and affine type systems in a call-by-value setting. We also give examples with many concrete models, including classical and quantum ones.

## 1 Introduction

Linear Logic [5] is a substructural logic where the rules for contraction and weakening are restricted. The logic has become very influential in computer science and it has inspired the development of linear type systems where discarding and copying of variables is restricted in accordance with the substructural rules of linear logic. Closely related to linear type systems, *affine* type systems are substructural type systems where only the rule for contraction is restricted, but weakening is completely unrestricted. Both linear and affine type systems have been used to design quantum programming languages [14, 16, 18, 20], because they enforce compliance with the laws of quantum mechanics, where uniform copying of quantum information is not possible [21].

General recursion is an important computational effect for (linear/affine) programming languages and it is especially useful in quantum programming, due to the probabilistic nature of many quantum algorithms and protocols which have to be repeated until the correct solution is found. When constructing categorical models for type systems with recursion, an important property is *computational adequacy*. Computational adequacy may be understood as formulating an equivalent purely denotational (i.e. mathematical) characterisation within the model of the operational notion of non-termination. That is, one should be able to determine whether a program terminates or not just by considering the interpretation of the program within the categorical model<sup>1</sup>.

In this paper, we consider two substructural type systems – one linear and one affine (§2) – and we show how we can interpret both of them within categorical models based on a double adjunction. We show that we can recover the linear/non-linear models of Benton [1, 2] as special cases of our models (§3). Furthermore, our treatment of both the linear and affine fragments of the lambda calculus we study is homogeneous – the interpretation of both languages are essentially the same and the models for the affine language require only a single additional axiom. We prove soundness and computational adequacy results for our categorical models (§4) and we present many concrete examples, both classical and quantum. In our models, we do not assume monoidal closure anywhere, and we show that, as a

---

<sup>1</sup>This does not imply decidability of termination, because the interpretation may not be computable.

Variables	$x, y, z$	
Types	$A, B, C$	$::= I \mid A + B \mid A \otimes B \mid A \multimap B \mid !A$
Non-linear types	$P, R$	$::= I \mid P + R \mid P \otimes R \mid !A$
Contexts	$\Gamma, \Sigma$	$::= x_1 : A_1, x_2 : A_2, \dots, x_n : A_n$
Non-linear contexts	$\Phi$	$::= x_1 : P_1, x_2 : P_2, \dots, x_n : P_n$
Terms	$m, n, p$	$::= x \mid * \mid m; n \mid \text{left}_{A,B} m \mid \text{right}_{A,B} m$ $\mid \text{case } m \text{ of } \{ \text{left } x \rightarrow n \text{ right } y \rightarrow p \}$ $\mid \langle m, n \rangle \mid \text{let } \langle x, y \rangle = m \text{ in } n \mid \lambda x^A. m \mid mn$ $\mid \text{lift } m \mid \text{force } m \mid \text{rec } z^!A. m$
Values	$v, w$	$::= x \mid * \mid \text{left}_{A,B} v \mid \text{right}_{A,B} v \mid \langle v, w \rangle \mid \lambda x^A. m \mid \text{lift } m$

Figure 1: Types, terms and contexts of the  $\lambda_l$  and  $\lambda_a$  calculi.

special case, if one wishes to use the monoidal closure of the computational category to interpret lambda abstractions in *linear lambda calculi* then this leads to a sound and adequate semantics, but doing so for the call-by-value *affine* language necessarily leads to degenerate models which are inadequate (§5).

The models that we study in this paper are inspired by the categorical models in [19], but there are some differences which we discuss in §3.1. Furthermore, the models we consider are also related to Moggi’s computational lambda calculus [13], Levy’s call-by-push-value [7] and the enriched effect calculus [4]. See [12] for a detailed analysis of the relationship between linear lambda calculi and their categorical models.

## 2 Syntax and Operational Semantics

We begin by describing the syntax of the calculi that we will study. We will consider two substructural lambda calculi. The first one is a mixed linear/non-linear lambda calculus which we name  $\lambda_l$  and the second one is a mixed affine/non-linear lambda calculus which we name  $\lambda_a$ . Figure 1 describes the term language of both calculi and also their types and contexts. The two calculi only differ in their formation rules, which we will introduce shortly. We note that  $\lambda_l$  has been studied in [17] (excluding recursion it appears as a fragment of Proto-Quipper) and also in [8, 9].

The non-linear types (ranged over by variables  $P, R$ ) form a subset of our types (ranged over by variables  $A, B, C$ ). We also distinguish between non-linear contexts (ranged over by  $\Phi$ ) and arbitrary contexts (ranged over by  $\Gamma, \Sigma$ ). Non-linear contexts contain only variables of non-linear types, whereas arbitrary contexts may contain variables of arbitrary types (which could be linear).

In both calculi, contraction is restricted to non-linear types only. That is, variables of non-linear type may always be duplicated, but in general, we do not allow copying of variables of arbitrary types (because such a type could be linear). The only difference between  $\lambda_l$  and  $\lambda_a$  is that in the former, weakening is restricted to non-linear types, whereas in the latter weakening is not restricted. This means, only variables of non-linear type may be discarded in  $\lambda_l$ , but in  $\lambda_a$  all variables are discardable. This is enforced by presenting different term formation rules for the two calculi (see Figure 2). In Figure 2, we write, as usual,  $\Gamma, \Sigma$  for the union of two disjoint contexts and  $\Gamma \vdash m : A$  to indicate that term  $m$  is well-formed under context  $\Gamma$  and has type  $A$ . The *values* are special terms which reduce to themselves in the operational semantics (see Figure 1). A value  $\Gamma \vdash v : A$  is said to be *non-linear* whenever  $A$  is a non-linear type and then it is easy to see that  $\Gamma$  must also be non-linear. See [9, 17] for a more detailed discussion of the syntax.

$$\begin{array}{c}
\frac{}{\Phi, x:A \vdash x:A} \text{ (for } \lambda_l) \quad \frac{}{\Phi \vdash * : I} \text{ (for } \lambda_l) \quad \frac{\Phi \vdash m : A}{\Phi \vdash \text{lift } m : !A} \text{ (for } \lambda_l) \\
\frac{}{\Gamma, x:A \vdash x:A} \text{ (for } \lambda_a) \quad \frac{}{\Gamma \vdash * : I} \text{ (for } \lambda_a) \quad \frac{\Phi \vdash m : A}{\Phi, \Gamma \vdash \text{lift } m : !A} \text{ (for } \lambda_a) \\
\frac{\Phi, \Gamma \vdash m : I \quad \Phi, \Sigma \vdash n : A}{\Phi, \Gamma, \Sigma \vdash m; n : A} \quad \frac{\Gamma \vdash m : !A}{\Gamma \vdash \text{force } m : A} \\
\frac{\Gamma \vdash m : A}{\Gamma \vdash \text{left}_{A,B} m : A+B} \quad \frac{\Gamma \vdash m : B}{\Gamma \vdash \text{right}_{A,B} m : A+B} \\
\frac{\Phi, \Gamma \vdash m : A+B \quad \Phi, \Sigma, x:A \vdash n : C \quad \Phi, \Sigma, y:B \vdash p : C}{\Phi, \Gamma, \Sigma \vdash \text{case } m \text{ of } \{\text{left } x \rightarrow n \mid \text{right } y \rightarrow p\} : C} \\
\frac{\Phi, \Gamma \vdash m : A \quad \Phi, \Sigma \vdash n : B}{\Phi, \Gamma, \Sigma \vdash \langle m, n \rangle : A \otimes B} \quad \frac{\Phi, \Gamma \vdash m : A \otimes B \quad \Phi, \Sigma, x:A, y:B \vdash n : C}{\Phi, \Gamma, \Sigma \vdash \text{let } \langle x, y \rangle = m \text{ in } n : C} \\
\frac{\Gamma, x:A \vdash m : B}{\Gamma \vdash \lambda x^A . m : A \multimap B} \quad \frac{\Phi, \Gamma \vdash m : A \multimap B \quad \Phi, \Sigma \vdash n : A}{\Phi, \Gamma, \Sigma \vdash mn : B} \\
\frac{\Phi, z : !A \vdash m : A}{\Phi \vdash \text{rec } z^{!A} . m : A}
\end{array}$$

where  $\Gamma \cap \Sigma = \emptyset$ .

Figure 2: Formation rules for  $\lambda_l$  and  $\lambda_a$  terms.

$$\begin{array}{c}
\frac{}{x \Downarrow x} \quad \frac{}{* \Downarrow *} \quad \frac{m \Downarrow * \quad n \Downarrow v}{m; n \Downarrow v} \\
\frac{m \Downarrow v}{\text{left } m \Downarrow \text{left } v} \quad \frac{m \Downarrow v}{\text{right } m \Downarrow \text{right } v} \\
\frac{m \Downarrow \text{left } v \quad n[v/x] \Downarrow w}{\text{case } m \text{ of } \{\text{left } x \rightarrow n \mid \text{right } y \rightarrow p\} \Downarrow w} \quad \frac{m \Downarrow v \quad n \Downarrow w}{\langle m, n \rangle \Downarrow \langle v, w \rangle} \\
\frac{m \Downarrow \text{right } v \quad p[v/y] \Downarrow w}{\text{case } m \text{ of } \{\text{left } x \rightarrow n \mid \text{right } y \rightarrow p\} \Downarrow w} \quad \frac{m \Downarrow \langle v, v' \rangle \quad n[v/x, v'/y] \Downarrow w}{\text{let } \langle x, y \rangle = m \text{ in } n \Downarrow w} \\
\frac{}{\lambda x . m \Downarrow \lambda x . m} \quad \frac{m \Downarrow \lambda x . m' \quad n \Downarrow v \quad m'[v/x] \Downarrow w}{mn \Downarrow w} \\
\frac{}{\text{lift } m \Downarrow \text{lift } m} \quad \frac{m \Downarrow \text{lift } m' \quad m' \Downarrow v}{\text{force } m \Downarrow v} \quad \frac{m[\text{lift } \text{rec } z^{!A} . m / z] \Downarrow v}{\text{rec } z^{!A} . m \Downarrow v}
\end{array}$$

Figure 3: Operational semantics of the  $\lambda_l$  and  $\lambda_a$  calculi.

The operational semantics of  $\lambda_l$  and  $\lambda_a$  is defined in the same way and it is standard. It is defined in terms of a big-step call-by-value reduction relation in Figure 3. Writing  $m \Downarrow v$  should be understood as saying that term  $m$  would eventually reduce to the value  $v$ , at which point termination occurs. We shall also say that a term  $m$  *terminates*, denoted by  $m \Downarrow$ , whenever there exists a value  $v$ , such that  $m \Downarrow v$ . Because of the presence of recursion, not all terms terminate. For example, the simplest non-terminating program of type  $A$  is  $\cdot \vdash \text{rec } z^!A. \text{force } z : A$ . As expected, our languages satisfy subject reduction, i.e., type assignment is preserved under term evaluation.

**Theorem 1** (Subject reduction). *If  $\Gamma \vdash m : A$  and  $m \Downarrow v$ , then  $\Gamma \vdash v : A$ .*

**Assumption 2.** *Throughout the remainder of the paper, we assume that all terms are well-formed.*

### 3 Categorical Models

In this section we describe the categorical models that we will use to interpret our substructural lambda calculi (§3.1). Afterwards, we consider the relationship of our models to other models of intuitionistic linear logic (§3.2), we then formulate some additional axioms that ensure computational adequacy holds (§3.3) and we conclude the section with concrete examples (§3.4).

#### 3.1 Definition of the Models

We start with the model for  $\lambda_l$  which serves as the basic model of our development. All subsequent models that we will present are specific instances of it where some additional structure is assumed. Our model is very similar to the one studied in [19], but in our language we allow lifting of terms (and not just values), so we treat " $!$ " in a more computational way by defining it as an endofunctor on  $\mathbf{C}$  (which is the usual interpretation of  $!$ ) instead of as an endofunctor on  $\mathbf{V}$  (which is done in [19] in order to ensure some coherence properties which we do not need in the present paper).

**Definition 3** ( $\lambda_l$ -model). *A (compact)  $\lambda_l$ -model is given by the following data:*

1. A cartesian category  $(\mathbf{B}, \times, 1)$  with finite coproducts  $(\mathbf{B}, \coprod, \emptyset)$ ;
2. A symmetric monoidal category  $(\mathbf{V}, \otimes_{\mathbf{V}}, I_{\mathbf{V}})$  with finite coproducts  $(\mathbf{V}, +_{\mathbf{V}}, 0_{\mathbf{V}})$ ;
3. A symmetric monoidal category  $(\mathbf{C}, \otimes, I)$  with finite coproducts  $(\mathbf{C}, +, 0)$ ;
4. A pair of symmetric monoidal adjunctions  $\mathbf{B} \begin{array}{c} \xrightarrow{J} \\ \perp \\ \xleftarrow{K} \end{array} \mathbf{V} \begin{array}{c} \xrightarrow{L} \\ \perp \\ \xleftarrow{R} \end{array} \mathbf{C}$ . We shall also write  $F := LJ : \mathbf{B} \rightarrow \mathbf{C}$ ,  $G := KR : \mathbf{C} \rightarrow \mathbf{B}$  and  $! := FG : \mathbf{C} \rightarrow \mathbf{C}$  and we write  $\eta : Id \Rightarrow GF$  and  $\varepsilon : ! \Rightarrow Id$  for the unit and counit, respectively, of the adjunction  $F \dashv G$ .
5. For every  $A \in Ob(\mathbf{V})$ , an adjunction  $L \circ (- \otimes_{\mathbf{V}} A) \dashv (A \multimap -) : \mathbf{C} \rightarrow \mathbf{V}$ , called currying;
- (6.) The comonad endofunctor  $! : \mathbf{C} \rightarrow \mathbf{C}$  is algebraically compact in a parameterised sense: for every  $B \in Ob(\mathbf{B})$ , the functor  $FB \otimes !(-) : \mathbf{C} \rightarrow \mathbf{C}$  has an initial algebra  $FB \otimes !\Omega \xrightarrow{\omega} \Omega$ , such that  $FB \otimes !\Omega \xleftarrow{\omega^{-1}} \Omega$  is its final coalgebra.

Let us now explain how the above data will be used for the interpretation of  $\lambda_l$ .

The category  $\mathbf{B}$  is the *base* category and it has sufficient structure to interpret non-linear values. Non-linear values are always discardable and duplicable and because of this,  $\mathbf{B}$  is assumed to be a cartesian category. Moreover, in all of our concrete models, the above adjunctions lift to  $\mathbf{B}$ -enriched adjunctions and  $\mathbf{B}$  serves as the *base* of enrichment.

The category  $\mathbf{V}$  is the category in which we interpret the *values* of  $\lambda_l$ , whether they are non-linear or not. The category  $\mathbf{C}$  is the category in which we interpret all terms or *computations* of  $\lambda_l$ . Because the language is call-by-value, we have that  $\lambda x^A.m$  is a value for any term  $m$ . Condition (5.) then allows us to interpret this by currying the interpretation of  $m$ . In order to interpret the  $!$  which is used for promotion of terms (especially lambda abstractions), we use condition (4.) which ensures this can be done in a coherent way, for both values and computations. Finally, condition (6.) is used to interpret recursion.

In many concrete  $\lambda_l$  models, the category  $\mathbf{V}$  is monoidal closed (this is the case for all concrete models we present) and the next lemma shows that condition (5.) is then automatically satisfied.

**Lemma 4.** *Assume we are given the same categorical data as in Definition 3 with the exception of condition (5.). Assume further  $\mathbf{V}$  is monoidal closed with  $(-\otimes_{\mathbf{V}} A) \dashv (A \multimap_{\mathbf{V}} -) : \mathbf{V} \rightarrow \mathbf{V}$ . It then follows condition (5.) is satisfied.*

*Proof.* Because  $L \circ (-\otimes_{\mathbf{V}} A) \dashv (A \multimap_{\mathbf{V}} -) \circ R : \mathbf{C} \rightarrow \mathbf{V}$ . □

Next, we formulate a categorical model for  $\lambda_a$ . It can be easily recovered from models of  $\lambda_l$  with one additional assumption.

**Definition 5** ( $\lambda_a$ -model). *A (compact)  $\lambda_a$ -model is given by a (compact)  $\lambda_l$ -model, where the tensor unit  $I_{\mathbf{V}}$  is a terminal object of  $\mathbf{V}$ .*

### 3.2 Relationship to LNL Models

We will compare our models to models of intuitionistic linear logic which are also known as linear/non-linear (LNL) models [1, 2]. Our models are tightly related to LNL models, but there are some subtle differences that stem from the choice of how to interpret lambda abstractions.

**Definition 6** (LNL model). *A (compact) linear/non-linear model is given by the following data:*

1. A cartesian category  $(\mathbf{B}, \times, 1)$  with finite coproducts  $(\mathbf{B}, \amalg, \emptyset)$ ;
2. A symmetric monoidal closed category  $(\mathbf{C}, \otimes, \multimap, I)$  with finite coproducts  $(\mathbf{C}, +, 0)$ ;
3. A symmetric monoidal adjunction  $\mathbf{B} \begin{array}{c} \xrightarrow{F} \\ \perp \\ \xleftarrow{G} \end{array} \mathbf{C}$ .

(4.) *The functor  $! = FG : \mathbf{C} \rightarrow \mathbf{C}$  is algebraically compact in a parameterised sense (Definition 3.6).*

**Remark 7.** *In the original definition of LNL models, the category  $\mathbf{B}$  is assumed to be cartesian closed. However, this is not necessary for our purposes, so we omit this from the definition. Nevertheless, in all concrete models we consider in this paper, the category  $\mathbf{B}$  is cartesian closed.*

In an LNL model, the category  $\mathbf{C}$  is assumed to be monoidal closed which is used for the interpretation of lambda abstractions, whereas in our models we do not assume monoidal closure anywhere. The other big difference is that values and computations are both interpreted in the same category  $\mathbf{C}$  of an LNL model. The implications of this on the semantics is discussed in §5.

We will now show that the notion of  $\lambda_l$ -model is more general than that of an LNL model.

**Proposition 8.** *Every (compact) LNL model  $\mathbf{B} \begin{array}{c} \xrightarrow{F} \\ \perp \\ \xleftarrow{G} \end{array} \mathbf{C}$  induces a (compact)  $\lambda_l$ -model given*

by  $\mathbf{B} \begin{array}{c} \xrightarrow{F} \\ \perp \\ \xleftarrow{G} \end{array} \mathbf{C} \begin{array}{c} \xrightarrow{Id} \\ \perp \\ \xleftarrow{Id} \end{array} \mathbf{C}$ , where  $\mathbf{V} = \mathbf{C}$  and  $L = R = Id$ . Moreover, in this case, the

denotational semantics in §4 collapses precisely to the denotational semantics of [8, 9].

Next, let us consider a  $\lambda_a$ -model (and therefore also a  $\lambda_l$ -model), which has been used to interpret the quantum lambda calculus (without recursion) [3] and the first-order quantum programming language QPL (which admits recursion, but not lambda abstractions) [15, 16].

**Example 9.** Let  $\mathbf{W}_{\text{NCPSU}}^*$  be the category of  $W^*$ -algebras and normal completely-positive subunital maps and let  $\mathbf{W}_{\text{NMIU}}^*$  be its full-on-objects subcategory of normal multiplicative involutive unital maps. Setting

$$\mathbf{V} = (\mathbf{W}_{\text{NMIU}}^*)^{\text{op}} \text{ and } \mathbf{C} = (\mathbf{W}_{\text{NCPSU}}^*)^{\text{op}}, \text{ one can define a } \lambda_a\text{-model } \mathbf{Set} \begin{array}{c} \xrightarrow{J} \\ \perp \\ \xleftarrow{K} \end{array} \mathbf{V} \begin{array}{c} \xleftarrow{L} \\ \perp \\ \xleftarrow{R} \end{array} \mathbf{C}$$

the details of which are described in [3]. Moreover, in this case, the category  $\mathbf{C}$  is not monoidal closed<sup>2</sup>.

Because the category  $\mathbf{C}$  is not monoidal closed, we see that we cannot interpret lambda abstractions as in an LNL model in this case. However, our  $\lambda_l$ -model does have sufficient structure and lambda abstractions in [3] are (concretely) interpreted in the same way as our (abstract) formulation in §4. Therefore, by interpreting linear lambda calculi within (compact)  $\lambda_l$ -models, instead of (compact) LNL models, we can discover a larger range of concrete models for these languages.

**Assumption 10.** Throughout the remainder of the paper we only consider compact models. For brevity, when we write "LNL/ $\lambda_l$ / $\lambda_a$ -model" we implicitly assume the model is also compact.

### 3.3 Computationally Adequate Models

It is possible to construct sound  $\lambda_l$ -models which are not computationally adequate. Let's consider an obvious example.

**Example 11.** The  $\lambda_a$ -model  $\mathbf{1} \begin{array}{c} \xrightarrow{Id} \\ \perp \\ \xleftarrow{Id} \end{array} \mathbf{1} \begin{array}{c} \xrightarrow{Id} \\ \perp \\ \xleftarrow{Id} \end{array} \mathbf{1}$  is not computationally adequate.

Of course, this model is completely degenerate and there is no way to distinguish between terminating and non-terminating computations within it. Computationally adequate models are often axiomatised in domain-theoretic terms and we shall do so as well. Let  $\mathbf{CPO}$  be the category with objects given by cpo's (posets which have suprema of increasing  $\omega$ -chains), and with morphisms given by Scott-continuous functions (monotone functions which preserve suprema of increasing  $\omega$ -chains). Let  $\mathbf{CPO}_{\perp!}$  be the subcategory of  $\mathbf{CPO}$  consisting of pointed cpo's (cpo's with a least element) and strict Scott-continuous functions (Scott-continuous functions that preserve the least element).

**Definition 12.** We shall say that a  $\lambda_l$ -model ( $\lambda_a$ -model)  $\mathbf{B} \begin{array}{c} \xrightarrow{J} \\ \perp \\ \xleftarrow{K} \end{array} \mathbf{V} \begin{array}{c} \xrightarrow{L} \\ \perp \\ \xleftarrow{R} \end{array} \mathbf{C}$  is order-enriched if:  $\mathbf{B}$  and  $\mathbf{V}$  are  $\mathbf{CPO}$ -enriched categories,  $\mathbf{C}$  is a  $\mathbf{CPO}_{\perp!}$ -enriched category, their coproduct and monoidal structures are  $\mathbf{CPO}$ -enriched and the functors  $L, R, J, K, \dashv$  are  $\mathbf{CPO}$ -enriched functors.

**Definition 13.** We shall say that a  $\lambda_l$ -model ( $\lambda_a$ -model)  $\mathbf{B} \begin{array}{c} \xrightarrow{J} \\ \perp \\ \xleftarrow{K} \end{array} \mathbf{V} \begin{array}{c} \xrightarrow{L} \\ \perp \\ \xleftarrow{R} \end{array} \mathbf{C}$  is adequate if it is order-enriched and  $id_I \neq \perp_{I,I}$ , where  $\perp_{A,B}$  is the least element in the hom-cpo  $\mathbf{C}(A, B)$ .

In the next section, we will show that these models are true to their name.

---

<sup>2</sup>Bert Lindenhovius. Personal Communication.

### 3.4 Concrete Models

We conclude the section by considering some concrete models.

**Example 14.** The adjunctions  $\mathbf{CPO} \xrightleftharpoons[U]{(-)_{\perp}} \mathbf{CPO}_{\perp!} \xrightleftharpoons[Id]{\perp} \mathbf{CPO}_{\perp!}$  form a computationally adequate  $\lambda_l$ -model, where  $U$  is the forgetful functor and  $(-)_{\perp}$  is domain-theoretic lifting (freely adding a least element).

The above data, in fact, determines an LNL model.

**Example 15.** The adjunctions  $\mathbf{CPO} \xrightleftharpoons[Id]{\perp} \mathbf{CPO} \xrightleftharpoons[U]{(-)_{\perp}} \mathbf{CPO}_{\perp!}$  form a computationally adequate  $\lambda_a$ -model.

In the above two examples, every object has a canonical comonoid structure and because of this, they are not truly representative models for linear and affine calculi. Next, we consider models where this does not hold.

**Example 16.** Let  $\mathbf{M}$  be an arbitrary symmetric monoidal category. We can see  $\mathbf{M}$  as a  $\mathbf{CPO}$ -enriched category when equipped with the discrete order (this is the free  $\mathbf{CPO}$ -enrichment of  $\mathbf{M}$ ). Let  $\mathbf{M}_{\perp}$  be the category obtained from  $\mathbf{M}$  by freely adding a least element to each hom-cpo (this is the free  $\mathbf{CPO}_{\perp!}$ -enrichment of  $\mathbf{M}$ ). Writing  $\mathbf{V} = [\mathbf{M}^{op}, \mathbf{CPO}]$  for the indicated  $\mathbf{CPO}$ -enriched functor category and  $\mathbf{C} = [\mathbf{M}_{\perp}^{op}, \mathbf{CPO}_{\perp!}]$  for the indicated  $\mathbf{CPO}_{\perp!}$ -functor category, we get a computationally adequate  $\lambda_l$ -model (see [8, 9] for more discussion and details). Moreover, if the tensor unit of  $\mathbf{M}$  is also a terminal object, then this data is a computationally adequate  $\lambda_a$ -model.

The above example shows a concrete model that has been used to interpret Proto-Quipper-M [8, 9, 17], a (quantum) circuit description language. The final model we consider is also inspired by quantum programming. It is a model of Proto-Quipper-M that supports recursive types.

**Example 17.** Let  $\mathbf{qCPO}$  be the category of quantum cpo's [6] and let  $\mathbf{qCPO}_{\perp!}$  be the subcategory of  $\mathbf{qCPO}$  of pointed objects and strict maps. Then the model

$$\mathbf{CPO} \xrightleftharpoons[K]{J} \mathbf{qCPO} \xrightleftharpoons[R]{L} \mathbf{qCPO}_{\perp!}$$

described in [6] is a computationally adequate  $\lambda_a$ -model.

## 4 Denotational Semantics

In this section we show how to interpret our substructural lambda calculi within the categorical models we discussed. Every type  $A$  admits an interpretation as an object  $\llbracket A \rrbracket_{\mathbf{V}} \in \text{Ob}(\mathbf{V})$  and as an object  $\llbracket A \rrbracket \in \text{Ob}(\mathbf{C})$ . In addition, every non-linear type  $P$  admits an interpretation  $\llbracket P \rrbracket_{\mathbf{B}} \in \text{Ob}(\mathbf{B})$ . These interpretations are defined in Figure 4 by simultaneous induction on the structure of types. The three different type interpretations are nicely related by coherent natural isomorphisms.

**Proposition 18.** For every type  $A : \llbracket A \rrbracket \cong L\llbracket A \rrbracket_{\mathbf{V}}$ . For every non-linear type  $P : \llbracket P \rrbracket_{\mathbf{V}} \cong J\llbracket P \rrbracket_{\mathbf{B}}$  and so  $\llbracket P \rrbracket \cong L\llbracket P \rrbracket_{\mathbf{V}} \cong F\llbracket P \rrbracket_{\mathbf{B}}$ .

$$\begin{array}{lll}
\llbracket I \rrbracket = I & \llbracket I \rrbracket_{\mathbf{V}} = I_{\mathbf{V}} & \llbracket I \rrbracket_{\mathbf{B}} = 1 \\
\llbracket !A \rrbracket = !\llbracket A \rrbracket & \llbracket !A \rrbracket_{\mathbf{V}} = JG\llbracket A \rrbracket & \llbracket !A \rrbracket_{\mathbf{B}} = G\llbracket A \rrbracket \\
\llbracket A + B \rrbracket = \llbracket A \rrbracket + \llbracket B \rrbracket & \llbracket A + B \rrbracket_{\mathbf{V}} = \llbracket A \rrbracket_{\mathbf{V}} +_{\mathbf{V}} \llbracket B \rrbracket_{\mathbf{V}} & \llbracket P + R \rrbracket_{\mathbf{B}} = \llbracket P \rrbracket_{\mathbf{B}} \coprod \llbracket R \rrbracket_{\mathbf{B}} \\
\llbracket A \otimes B \rrbracket = \llbracket A \rrbracket \otimes \llbracket B \rrbracket & \llbracket A \otimes B \rrbracket_{\mathbf{V}} = \llbracket A \rrbracket_{\mathbf{V}} \otimes_{\mathbf{V}} \llbracket B \rrbracket_{\mathbf{V}} & \llbracket P \otimes R \rrbracket_{\mathbf{B}} = \llbracket P \rrbracket_{\mathbf{B}} \times \llbracket R \rrbracket_{\mathbf{B}} \\
\llbracket A \multimap B \rrbracket = L(\llbracket A \rrbracket_{\mathbf{V}} \multimap \llbracket B \rrbracket) & \llbracket A \multimap B \rrbracket_{\mathbf{V}} = \llbracket A \rrbracket_{\mathbf{V}} \multimap \llbracket B \rrbracket & 
\end{array}$$

Figure 4: Interpretation of types.

These isomorphisms are also defined by induction on the structure of types, but we omit the details here (the construction is similar to the one in [10, 11]). Moreover, in order to avoid using excessive notation in the interpretation of terms, we make the following assumption.

**Assumption 19.** *From now on, we suppress the natural isomorphisms related to the monoidal structure of all categories, the strong monoidal functors of the adjunction and the preservation of colimits. With this in place, the isomorphisms of Proposition 18 become equalities and we will write them as such.*

Of course, our results continue to hold even without this assumption, but we do this for brevity of the presentation (see [10, 11] for more information on how to handle such isomorphisms).

The interpretation of a context  $\Gamma = \{x_1 : A_1, \dots, x_n : A_n\}$  within  $\mathbf{C}$  is defined in the usual way as  $\llbracket \Gamma \rrbracket = \llbracket A_1 \rrbracket \otimes \dots \otimes \llbracket A_n \rrbracket$ . Similarly, we may define its interpretation in  $\mathbf{V}$ . Every non-linear context  $\Phi = \{x_1 : P_1, \dots, x_n : P_n\}$  also admits an interpretation within  $\mathbf{B}$  by  $\llbracket \Phi \rrbracket_{\mathbf{B}} = \llbracket P_1 \rrbracket_{\mathbf{B}} \times \dots \times \llbracket P_n \rrbracket_{\mathbf{B}}$ . Then, just as in Proposition 18, we have  $\llbracket \Gamma \rrbracket = L\llbracket \Gamma \rrbracket_{\mathbf{V}}$  and for non-linear types contexts  $\Phi$  we also have  $\llbracket \Phi \rrbracket = L\llbracket \Phi \rrbracket_{\mathbf{V}} = F\llbracket \Phi \rrbracket_{\mathbf{B}}$ .

Before we may define the interpretation of terms, we have to explain how to construct morphisms for copying, deletion and promotion of non-linear primitives. We do this in the following way.

**Definition 20.** *For every non-linear type or context  $X$ , we define discarding ( $\diamond$ ), copying ( $\Delta$ ) and promotion ( $\square$ ) morphisms in all three categories:*

$$\begin{array}{lll}
\diamond_X^{\mathbf{B}} := \llbracket X \rrbracket_{\mathbf{B}} \xrightarrow{1} 1 & \diamond_X^{\mathbf{V}} := J\diamond_X^{\mathbf{B}} & \diamond_X^{\mathbf{C}} := F\diamond_X^{\mathbf{B}} \\
\Delta_X^{\mathbf{B}} := \llbracket X \rrbracket_{\mathbf{B}} \xrightarrow{\langle id, id \rangle} \llbracket X \rrbracket_{\mathbf{B}} \times \llbracket X \rrbracket_{\mathbf{B}} & \Delta_X^{\mathbf{V}} := J\Delta_X^{\mathbf{B}} & \Delta_X^{\mathbf{C}} := F\Delta_X^{\mathbf{B}} \\
\square_X^{\mathbf{B}} := \llbracket X \rrbracket_{\mathbf{B}} \xrightarrow{\eta} GF\llbracket X \rrbracket_{\mathbf{B}} = G\llbracket X \rrbracket = \llbracket !X \rrbracket_{\mathbf{B}} & \square_X^{\mathbf{V}} := J\square_X^{\mathbf{B}} & \square_X^{\mathbf{C}} := F\square_X^{\mathbf{B}}
\end{array}$$

The substructural morphisms  $\chi_X^{\mathbf{C}}$  are the ones directly used for the interpretation of terms, so we shall simply write them as  $\diamond_X : \llbracket X \rrbracket \rightarrow I$  and  $\Delta_X : \llbracket X \rrbracket \rightarrow \llbracket X \rrbracket \otimes \llbracket X \rrbracket$  and  $\square_X : \llbracket X \rrbracket \rightarrow \llbracket !X \rrbracket$ , omitting the superscript.

**Proposition 21.** *For every non-linear type or context  $X$ , the substructural maps for copying and discarding form cocommutative comonoids in their respective categories:*

1. The triple  $(\llbracket X \rrbracket_{\mathbf{B}}, \Delta_X^{\mathbf{B}}, \diamond_X^{\mathbf{B}})$  is a cocommutative comonoid in  $\mathbf{B}$ .
2. The triple  $(\llbracket X \rrbracket_{\mathbf{V}}, \Delta_X^{\mathbf{V}}, \diamond_X^{\mathbf{V}})$  is a cocommutative comonoid in  $\mathbf{V}$ .
3. The triple  $(\llbracket X \rrbracket, \Delta_X, \diamond_X)$  is a cocommutative comonoid in  $\mathbf{C}$ .

Moreover, the comonoid homomorphisms with respect to the above structures are:



$$\begin{array}{ccccc}
FX \otimes !FX & \xleftarrow{\text{id} \otimes F\eta} & FX \otimes FX & \xleftarrow{F\langle \text{id}, \text{id} \rangle} & FX \\
\text{id} \otimes !\gamma_{FX} \downarrow & & & & \downarrow \gamma_{FX} \\
FX \otimes !\Omega_{FX} & \xleftarrow{\omega_{FX}^{-1}} & & & \Omega_{FX} \\
\text{id} \downarrow & & & & \downarrow \text{id} \\
FX \otimes !\Omega_{FX} & \xrightarrow{\omega_{FX}} & & & \Omega_{FX} \\
\text{id} \otimes !\sigma_m \downarrow & & & & \downarrow \sigma_m \\
FX \otimes !A & \xrightarrow{m} & & & A
\end{array}$$

Figure 5: Definition of  $\sigma_m$  and  $\gamma_{FX}$ . Given an object  $FX$  and a morphism  $m$  as above,  $\sigma_m$  and  $\gamma_{FX}$  are the unique maps making the above diagram commute, where  $\Omega_{FX}$  is the initial (final)  $FX \otimes !(-)$ -algebra.

1. Every morphism of  $\mathbf{B}$  (because  $\mathbf{B}$  is cartesian).
2. The morphisms of  $\mathbf{V}$  in the image of  $J$ .
3. The morphisms of  $\mathbf{C}$  in the image of  $F$ .

So, we see that in any  $\lambda_l$ -model, we may define copy and discarding morphisms at every non-linear type. However, to interpret  $\lambda_a$ , we have to be able to construct discarding morphisms at all types (including linear ones). This is possible in a  $\lambda_a$ -model, because of the additional assumption that  $I_{\mathbf{V}}$  is a terminal object in  $\mathbf{V}$ . Therefore, in an  $\lambda_a$ -model, we simply define the discarding map to be the unique map  $\diamond_A^{\mathbf{V}} : [A]_{\mathbf{V}} \rightarrow I_{\mathbf{V}}$ , which then induces a discarding map  $\diamond_A := L \diamond_A^{\mathbf{V}} : [A] \rightarrow I$  in  $\mathbf{C}$ . Note that the latter map can then discard any morphism in the image of  $L$ , and we will see that the interpretation of values satisfies this.

We may now define the interpretation of terms of  $\lambda_l$ . As usual, a well-formed term  $\Gamma \vdash m : A$  is interpreted as a morphism  $\llbracket \Gamma \vdash m : A \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket A \rrbracket$  in  $\mathbf{C}$  which is defined by induction on the derivation of  $\Gamma \vdash m : A$  in Figure 6. We will also often abbreviate this by simply writing  $\llbracket m \rrbracket$ , instead of  $\llbracket \Gamma \vdash m : A \rrbracket$ . The interpretation of recursion makes use of the auxiliary definition in Figure 5 which is well-defined due to the assumption in Definition 3.6. In the interpretation of case terms, we use the fact that the tensor product distributes over coproducts, provided they are both in the image of  $L$ , which follows from the assumption in Definition 3.5.

The interpretation of  $\lambda_a$  terms within a  $\lambda_a$ -model is done in the same way as in Figure 6, but where we update the three rules that are different among the calculi to also handle the more general contexts, as follows:

$$\begin{aligned}
\llbracket \Gamma, x : A \vdash x : A \rrbracket &:= \llbracket \Gamma \rrbracket \otimes \llbracket A \rrbracket \xrightarrow{\circ \otimes \text{id}} I \otimes \llbracket A \rrbracket = \llbracket A \rrbracket \\
\llbracket \Gamma \vdash * : I \rrbracket &:= \llbracket \Gamma \rrbracket \xrightarrow{\circ} I = \llbracket I \rrbracket \\
\llbracket \Phi, \Gamma \vdash \text{lift } m : !A \rrbracket &:= \llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \xrightarrow{\text{id} \otimes \circ} \llbracket \Phi \rrbracket \otimes I = \llbracket \Phi \rrbracket \xrightarrow{\square} !\llbracket \Phi \rrbracket \xrightarrow{! \llbracket m \rrbracket} !\llbracket A \rrbracket = \llbracket !A \rrbracket
\end{aligned}$$

In order to show that our models are sound, we have to show that the interpretations of (non-linear) values interact nicely with the substructural morphisms we have defined (Proposition 24). This is done by showing that non-linear values admit an interpretation in  $\mathbf{B}$  and that all values admit an interpretation in  $\mathbf{V}$ , such that the interpretation of values in  $\mathbf{C}$  are in the image of the respective left adjoints.

$$\begin{aligned}
\llbracket \Phi, x : A \vdash x : A \rrbracket &:= \llbracket \Phi \rrbracket \otimes \llbracket A \rrbracket \xrightarrow{\circ \otimes \text{id}} I \otimes \llbracket A \rrbracket = \llbracket A \rrbracket \\
\llbracket \Phi \vdash * : I \rrbracket &:= \llbracket \Phi \rrbracket \xrightarrow{\circ} I = \llbracket I \rrbracket \\
\llbracket \Phi, \Gamma, \Sigma \vdash m; n : A \rrbracket &:= \llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \otimes \llbracket \Sigma \rrbracket \xrightarrow{\Delta \otimes \text{id}} \llbracket \Phi \rrbracket \otimes \llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \otimes \llbracket \Sigma \rrbracket \xrightarrow{\cong} \\
&\llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \otimes \llbracket \Phi \rrbracket \otimes \llbracket \Sigma \rrbracket \xrightarrow{\llbracket m \rrbracket \otimes \llbracket n \rrbracket} I \otimes \llbracket A \rrbracket = \llbracket A \rrbracket \\
\llbracket \Gamma \vdash \text{left}_{A,B} m : A + B \rrbracket &:= \llbracket \Gamma \rrbracket \xrightarrow{\llbracket m \rrbracket} \llbracket A \rrbracket \xrightarrow{\text{left}} \llbracket A \rrbracket + \llbracket B \rrbracket = \llbracket A + B \rrbracket \\
\llbracket \Gamma \vdash \text{right}_{A,B} m : A + B \rrbracket &:= \llbracket \Gamma \rrbracket \xrightarrow{\llbracket m \rrbracket} \llbracket B \rrbracket \xrightarrow{\text{right}} \llbracket A \rrbracket + \llbracket B \rrbracket = \llbracket A + B \rrbracket \\
\llbracket \Phi, \Gamma, \Sigma \vdash \text{case } m \text{ of } \{ \text{left } x \rightarrow n \mid \text{right } y \rightarrow p \} : C \rrbracket &:= \llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \otimes \llbracket \Sigma \rrbracket \xrightarrow{\Delta \otimes \text{id}} \\
&\llbracket \Phi \rrbracket \otimes \llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \otimes \llbracket \Sigma \rrbracket \xrightarrow{\cong} \llbracket \Phi \rrbracket \otimes \llbracket \Sigma \rrbracket \otimes \llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \xrightarrow{\text{id} \otimes \llbracket m \rrbracket} \llbracket \Phi \rrbracket \otimes \llbracket \Sigma \rrbracket \otimes \llbracket A + B \rrbracket \xrightarrow{\cong} \\
&(\llbracket \Phi \rrbracket \otimes \llbracket \Sigma \rrbracket \otimes \llbracket A \rrbracket) + (\llbracket \Phi \rrbracket \otimes \llbracket \Sigma \rrbracket \otimes \llbracket B \rrbracket) \xrightarrow{\llbracket [n], [p] \rrbracket} \llbracket C \rrbracket \\
\llbracket \Phi, \Gamma, \Sigma \vdash \langle m, n \rangle : A \otimes B \rrbracket &:= \llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \otimes \llbracket \Sigma \rrbracket \xrightarrow{\Delta \otimes \text{id}} \llbracket \Phi \rrbracket \otimes \llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \otimes \llbracket \Sigma \rrbracket \xrightarrow{\cong} \\
&\llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \otimes \llbracket \Phi \rrbracket \otimes \llbracket \Sigma \rrbracket \xrightarrow{\llbracket m \rrbracket \otimes \llbracket n \rrbracket} \llbracket A \rrbracket \otimes \llbracket B \rrbracket = \llbracket A \otimes B \rrbracket \\
\llbracket \Phi, \Gamma, \Sigma \vdash \text{let } \langle x, y \rangle = m \text{ in } n : C \rrbracket &:= \llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \otimes \llbracket \Sigma \rrbracket \xrightarrow{\Delta \otimes \text{id}} \llbracket \Phi \rrbracket \otimes \llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \otimes \llbracket \Sigma \rrbracket \xrightarrow{\cong} \\
&\llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \otimes \llbracket \Phi \rrbracket \otimes \llbracket \Sigma \rrbracket \xrightarrow{\llbracket m \rrbracket \otimes \text{id}} \llbracket A \otimes B \rrbracket \otimes \llbracket \Phi \rrbracket \otimes \llbracket \Sigma \rrbracket \xrightarrow{\cong} \llbracket \Phi \rrbracket \otimes \llbracket \Sigma \rrbracket \otimes \llbracket A \rrbracket \otimes \llbracket B \rrbracket \xrightarrow{\llbracket n \rrbracket} \llbracket C \rrbracket \\
\llbracket \Gamma \vdash \lambda x^A. m : A \multimap B \rrbracket &:= \llbracket \Gamma \rrbracket = L[\Gamma]_{\mathbf{V}} \xrightarrow{L\text{curry}(\llbracket m \rrbracket)} L(\llbracket A \rrbracket_{\mathbf{V}} \multimap \llbracket B \rrbracket) = \llbracket A \multimap B \rrbracket \\
\llbracket \Phi, \Gamma, \Sigma \vdash mn : B \rrbracket &:= \llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \otimes \llbracket \Sigma \rrbracket \xrightarrow{\Delta \otimes \text{id}} \llbracket \Phi \rrbracket \otimes \llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \otimes \llbracket \Sigma \rrbracket \xrightarrow{\cong} \\
&\llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \otimes \llbracket \Phi \rrbracket \otimes \llbracket \Sigma \rrbracket \xrightarrow{\llbracket m \rrbracket \otimes \llbracket n \rrbracket} (\llbracket A \multimap B \rrbracket) \otimes \llbracket A \rrbracket = L((\llbracket A \rrbracket_{\mathbf{V}} \multimap \llbracket B \rrbracket) \otimes_{\mathbf{V}} \llbracket A \rrbracket_{\mathbf{V}}) \xrightarrow{\text{eval}} \llbracket B \rrbracket \\
\llbracket \Phi \vdash \text{lift } m : !A \rrbracket &:= \llbracket \Phi \rrbracket \xrightarrow{\square} !\llbracket \Phi \rrbracket \xrightarrow{!\llbracket m \rrbracket} !\llbracket A \rrbracket = \llbracket !A \rrbracket \\
\llbracket \Gamma \vdash \text{force } m : A \rrbracket &:= \llbracket \Gamma \rrbracket \xrightarrow{\llbracket m \rrbracket} !\llbracket A \rrbracket \xrightarrow{\varepsilon} \llbracket A \rrbracket \\
\llbracket \Phi \vdash \text{rec } x^!A. m : A \rrbracket &:= \llbracket \Phi \rrbracket \xrightarrow{\gamma_{\llbracket \Phi \rrbracket}} \Omega_{\llbracket \Phi \rrbracket} \xrightarrow{\sigma_{\llbracket m \rrbracket}} \llbracket A \rrbracket
\end{aligned}$$

Figure 6: Interpretation of  $\lambda_l$ -terms.

**Lemma 22.** *For every non-linear value  $\Phi \vdash v : P$ , we define an interpretation  $\llbracket \Phi \vdash v : P \rrbracket_{\mathbf{B}} : \llbracket \Phi \rrbracket_{\mathbf{B}} \rightarrow \llbracket P \rrbracket_{\mathbf{B}}$  within  $\mathbf{B}$  by induction on the derivation of  $\Phi \vdash v : P$  as follows:*

$$\begin{aligned} \llbracket \Phi, x : P \vdash x : P \rrbracket_{\mathbf{B}} &:= \llbracket \Phi \rrbracket_{\mathbf{B}} \times \llbracket P \rrbracket_{\mathbf{B}} \xrightarrow{\pi_2} \llbracket P \rrbracket_{\mathbf{B}} \\ \llbracket \Phi \vdash * : I \rrbracket_{\mathbf{B}} &:= \llbracket \Phi \rrbracket_{\mathbf{B}} \xrightarrow{1} 1 = \llbracket I \rrbracket_{\mathbf{B}} \\ \llbracket \Phi \vdash \text{left}_{P,R} v : P + R \rrbracket_{\mathbf{B}} &:= \llbracket \Phi \rrbracket_{\mathbf{B}} \xrightarrow{\llbracket v \rrbracket_{\mathbf{B}}} \llbracket P \rrbracket_{\mathbf{B}} \xrightarrow{\text{inl}} \llbracket P \rrbracket_{\mathbf{B}} \amalg \llbracket R \rrbracket_{\mathbf{B}} = \llbracket P + R \rrbracket_{\mathbf{B}} \\ \llbracket \Phi \vdash \text{right}_{P,R} v : P + R \rrbracket_{\mathbf{B}} &:= \llbracket \Phi \rrbracket_{\mathbf{B}} \xrightarrow{\llbracket v \rrbracket_{\mathbf{B}}} \llbracket R \rrbracket_{\mathbf{B}} \xrightarrow{\text{inr}} \llbracket P \rrbracket_{\mathbf{B}} \amalg \llbracket R \rrbracket_{\mathbf{B}} = \llbracket P + R \rrbracket_{\mathbf{B}} \\ \llbracket \Phi \vdash \langle v, w \rangle : P \otimes R \rrbracket_{\mathbf{B}} &:= \llbracket \Phi \rrbracket_{\mathbf{B}} \xrightarrow{\langle \llbracket v \rrbracket_{\mathbf{B}}, \llbracket w \rrbracket_{\mathbf{B}} \rangle} \llbracket P \rrbracket_{\mathbf{B}} \times \llbracket R \rrbracket_{\mathbf{B}} = \llbracket P \otimes R \rrbracket_{\mathbf{B}} \\ \llbracket \Phi \vdash \text{lift } m : !A \rrbracket_{\mathbf{B}} &:= \llbracket \Phi \rrbracket_{\mathbf{B}} \xrightarrow{\eta} GF \llbracket \Phi \rrbracket_{\mathbf{B}} = G \llbracket \Phi \rrbracket \xrightarrow{G \llbracket m \rrbracket} G \llbracket A \rrbracket = \llbracket !A \rrbracket_{\mathbf{B}} \end{aligned}$$

Then  $\llbracket v \rrbracket = F \llbracket v \rrbracket_{\mathbf{B}}$ .

Using the same idea, we can define for every value  $v$  an interpretation  $\llbracket v \rrbracket_{\mathbf{V}}$  in  $\mathbf{V}$ .

**Lemma 23.** *For every value  $\Gamma \vdash v : A$  of both  $\lambda_l$  and  $\lambda_a$  it is possible to define an interpretation  $\llbracket \Gamma \vdash v : A \rrbracket_{\mathbf{V}} : \llbracket \Gamma \rrbracket_{\mathbf{V}} \rightarrow \llbracket A \rrbracket_{\mathbf{V}}$  within  $\mathbf{V}$ , such that  $\llbracket v \rrbracket = L \llbracket v \rrbracket_{\mathbf{V}}$  (details omitted for lack of space).*

**Proposition 24.** *In any  $\lambda_l$ -model, for every non-linear value  $\Phi \vdash v : P$ , we have:*

$$\diamond_P \circ \llbracket v \rrbracket = \diamond_{\Phi} \quad \triangle_P \circ \llbracket v \rrbracket = (\llbracket v \rrbracket \otimes \llbracket v \rrbracket) \circ \triangle_{\Phi} \quad \square_P \circ \llbracket v \rrbracket = !\llbracket v \rrbracket \circ \square_{\Phi}.$$

Moreover, in any  $\lambda_a$ -model, for every value  $\Gamma \vdash v : A$ , we also have that  $\diamond_A \circ \llbracket v \rrbracket = \diamond_{\Gamma}$ .

With this place, soundness may now be proved in a straightforward way.

**Theorem 25** (Soundness). *If  $\Gamma \vdash m : A$  and  $m \Downarrow v$ , then  $\llbracket m \rrbracket = \llbracket v \rrbracket$  (in both  $\lambda_l$  and  $\lambda_a$ ).*

The above theorem shows that  $\lambda_l$  ( $\lambda_a$ ) can be soundly interpreted in any  $\lambda_l$ -model ( $\lambda_a$ -model). To prove computational adequacy, we need some additional assumptions (as Example 11 demonstrates).

**Theorem 26** (Adequacy). *Let  $\cdot \vdash p : I$  be a closed program of unit type in  $\lambda_l$  ( $\lambda_a$ ). Then in any computationally adequate  $\lambda_l$ -model ( $\lambda_a$ -model):*

$$\llbracket p \rrbracket \neq \perp \text{ iff } p \Downarrow.$$

*Proof.* This may be established using standard proof techniques for adequacy, e.g. [14, 16].  $\square$

## 5 Lambda Abstractions, Monoidal Closure and Adequacy

One of the stated goals of the present paper is to study how lambda abstractions may be interpreted for substructural lambda calculi and the effects this has on computational adequacy. We have shown that if one uses currying through our category  $\mathbf{V}$ , then both  $\lambda_l$  and  $\lambda_a$  can be interpreted in a sound and adequate way in §4. However, for linear lambda calculi, ones often sees lambda abstractions interpreted using the monoidal closed structure of the computational category. In this section, we will assume that our category  $\mathbf{C}$  is monoidal closed and update the semantics to interpret lambda abstractions through this structure and we then show that this does not cause problems for  $\lambda_l$ , but it does cause problems for  $\lambda_a$ .

**Assumption 27.** *Throughout the remainder of the section, we assume that the category  $\mathbf{C}$  of a  $\lambda_l$ -model is a symmetric monoidal closed category  $(\mathbf{C}, \otimes, \multimap, I)$ . The functor  $\multimap : \mathbf{C}^{\text{op}} \times \mathbf{C} \rightarrow \mathbf{C}$  now refers to the functor induced by the adjunction  $(-\otimes A) \dashv (A \multimap -)$  of the symmetric monoidal closed structure.*

As a special case of Proposition 8, by taking  $\mathbf{V} = \mathbf{C}$  and  $L = \text{Id} = R$ , we get a sound model of  $\lambda_l$ , where lambda abstractions are interpreted via the monoidal closed structure of  $\mathbf{C}$ . Under the additional assumptions of Definition 13, we also get computational adequacy, and it is not too difficult to find computationally adequate concrete models (Examples 14, 16, 17). Therefore, we see that interpreting lambda abstractions via the monoidal closed structure of the computational category is not a problem for *linear* lambda calculi.

Next, let us consider the situation for  $\lambda_a$ . We will first explain how to interpret  $\lambda_a$  using the newly assumed structure. The interpretation of types is updated by setting  $\llbracket A \multimap B \rrbracket = \llbracket A \rrbracket \multimap \llbracket B \rrbracket$ . The interpretation of lambda abstractions and application are updated as follows:

$$\begin{aligned} \llbracket \Gamma \vdash \lambda x^A. m : A \multimap B \rrbracket &:= \llbracket \Gamma \rrbracket \xrightarrow{\mathbf{curry}(\llbracket m \rrbracket)} (\llbracket A \rrbracket \multimap \llbracket B \rrbracket) = \llbracket A \multimap B \rrbracket \\ \llbracket \Phi, \Gamma, \Sigma \vdash mn : B \rrbracket &:= \llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \otimes \llbracket \Sigma \rrbracket \xrightarrow{\Delta \otimes \text{id}} \llbracket \Phi \rrbracket \otimes \llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \otimes \llbracket \Sigma \rrbracket \xrightarrow{\cong} \\ &\llbracket \Phi \rrbracket \otimes \llbracket \Gamma \rrbracket \otimes \llbracket \Phi \rrbracket \otimes \llbracket \Sigma \rrbracket \xrightarrow{\llbracket m \rrbracket \otimes \llbracket n \rrbracket} (\llbracket A \rrbracket \multimap \llbracket B \rrbracket) \otimes \llbracket A \rrbracket \xrightarrow{\text{eval}} \llbracket B \rrbracket \end{aligned}$$

We may now show the interpretation is provably inadequate and also completely degenerate.

**Proposition 28.** *Assume we are given a sound  $\lambda_a$ -model under Assumption 27. Then  $\mathbf{C} \simeq \mathbf{1}$  and so the  $\lambda_a$ -model is not computationally adequate (because every homset of  $\mathbf{C}$  has exactly one morphism).*

*Proof.* The monoidal closure of  $\mathbf{C}$  together with Definition 3.6 imply that  $\mathbf{C}$  is a pointed category (has a zero object) [8, Theorem 4.9] and so we shall write  $\perp_{A,B} : A \rightarrow B$  for its zero morphisms. The monoidal closure of  $\mathbf{C}$  implies that  $A \otimes 0 \cong 0$ , for every  $A \in \text{Ob}(\mathbf{C})$  and that  $f \otimes \perp_{C,D} = \perp_{A \otimes B, C \otimes D}$ , for any  $f : A \rightarrow B$ .

Let  $p = \text{rec } z^I$ . force  $z$ . Then,  $\cdot \vdash p : I$  and moreover  $p \not\Downarrow$  with  $\llbracket p \rrbracket = \perp$  [8, Theorem 4.9]. But then

$$\llbracket \cdot \vdash \lambda x^I. p : I \multimap I \rrbracket = \mathbf{curry}(\perp) = (I \multimap \perp) \circ \eta' = \perp \circ \eta' = \perp.$$

Next, consider the program  $t = (\lambda y^{I \multimap I}. *) (\lambda x^I. p)$ . This program is well-formed in  $\lambda_a$  with  $\cdot \vdash t : I$  (but it is not well-formed in  $\lambda_l$ ) and  $t \not\Downarrow *$ . By soundness,  $\llbracket t \rrbracket = \llbracket * \rrbracket = \text{id}_I$ . By definition of  $\llbracket - \rrbracket$ , we have

$$\llbracket t \rrbracket = \text{eval} \circ (\mathbf{curry}(\diamond) \otimes \perp) \circ \cong \circ (\Delta \otimes \text{id}) = \text{eval} \circ \perp \circ \cong \circ (\Delta \otimes \text{id}) = \perp.$$

This means  $\text{id}_I$  is a zero morphism and so  $I$  is a zero object. Then, every  $A \in \text{Ob}(\mathbf{C})$  is a zero object, because  $A \cong A \otimes I \cong A \otimes 0 \cong 0$  and therefore  $\mathbf{C} \simeq \mathbf{1}$ .  $\square$

**Remark 29.** *If we model recursion by assuming our model is order-enriched instead of compact in Definition 3, then one can also show that the model becomes degenerate using similar arguments.*

**Remark 30.** *If one assumes the full law of beta-equivalence is satisfied by the language, then the degeneration can be demonstrated for a wider class of models as well.*

**Acknowledgements.** I thank Bert Lindenhovius for discussions about  $W^*$ -algebras. I also thank the anonymous reviewers for their feedback and I gratefully acknowledge financial support from the French projects ANR-17-CE25-0009 SoftQPro and PIA-GDN/Quantex.

## References

- [1] P. N. Benton & Philip Wadler (1996): *Linear Logic, Monads and the Lambda Calculus*. In: *Logic in Computer Science*, doi:10.1109/LICS.1996.561458.
- [2] P.N. Benton (1995): *A mixed linear and non-linear logic: Proofs, terms and models*. In: *Computer Science Logic: 8th Workshop, CSL '94*, doi:10.1007/BFb0022251.
- [3] Kenta Cho & Abraham Westerbaan (2016): *Von Neumann Algebras form a Model for the Quantum Lambda Calculus*. Available at <http://arxiv.org/abs/1603.02133>. Manuscript.
- [4] Jeff Egger, Rasmus Ejlers Møgelberg & Alex Simpson (2014): *The enriched effect calculus: syntax and semantics*. *J. Log. Comput.* 24(3), pp. 615–654, doi:10.1093/logcom/exs025.
- [5] Jean-Yves Girard (1987): *Linear Logic*. *Theor. Comput. Sci.* 50, pp. 1–102, doi:10.1016/0304-3975(87)90045-4.
- [6] Andre Kornell, Bert Lindenhovius & Michael Mislove (2020): *Quantum CPOs*. Preprint.
- [7] Paul Blain Levy (2004): *Call-By-Push-Value: A Functional/Imperative Synthesis*. *Semantics Structures in Computation 2*, Springer.
- [8] Bert Lindenhovius, Michael Mislove & Vladimir Zamdzhiev (2018): *Enriching a Linear/Non-linear Lambda Calculus: A Programming Language for String Diagrams*. In: *LICS 2018*, ACM, doi:10.1145/3209108.3209196.
- [9] Bert Lindenhovius, Michael Mislove & Vladimir Zamdzhiev (2020): *Semantics for a Lambda Calculus for String Diagrams*. Preprint.
- [10] Bert Lindenhovius, Michael W. Mislove & Vladimir Zamdzhiev (2019): *Mixed linear and non-linear recursive types*. *Proc. ACM Program. Lang.* 3(ICFP), pp. 111:1–111:29, doi:10.1145/3341715.
- [11] Bert Lindenhovius, Michael W. Mislove & Vladimir Zamdzhiev (2020): *LNL-FPC: The Linear/Non-linear Fixpoint Calculus*. Available at <http://arxiv.org/abs/1906.09503>. Preprint.
- [12] Maria Emilia Maietti, Paola Maneggia, Valeria de Paiva & Eike Ritter (2005): *Relating Categorical Semantics for Intuitionistic Linear Logic*. *Applied Categorical Structures* 13(1), doi:10.1007/s10485-004-3134-z.
- [13] Eugenio Moggi (1991): *Notions of Computation and Monads*. *Inf. Comput.* 93(1), pp. 55–92, doi:10.1016/0890-5401(91)90052-4.
- [14] Michele Pagani, Peter Selinger & Benoît Valiron (2014): *Applying quantitative semantics to higher-order quantum computing*. In: *POPL*, doi:10.1145/2535838.2535879.
- [15] Romain Péchoux, Simon Perdrix, Mathys Rennela & Vladimir Zamdzhiev (2020): *Quantum Programming with Inductive Datatypes*. Preprint.
- [16] Romain Péchoux, Simon Perdrix, Mathys Rennela & Vladimir Zamdzhiev (2020): *Quantum Programming with Inductive Datatypes: Causality and Affine Type Theory*. In: *Foundations of Software Science and Computation Structures 2020*, pp. 562–581, doi:10.1007/978-3-030-45231-5\_29.
- [17] Francisco Rios & Peter Selinger (2017): *A categorical model for a quantum circuit description language*. In: *QPL 2017*, pp. 164–178, doi:10.4204/EPTCS.266.11.
- [18] Peter Selinger (2004): *Towards a quantum programming language*. *Mathematical Structures in Computer Science* 14(4), pp. 527–586, doi:10.1017/S0960129504004256.
- [19] Peter Selinger & Benoît Valiron (2008): *A Linear-non-Linear Model for a Computational Call-by-Value Lambda Calculus (Extended Abstract)*. In: *FOSSACS 2008*, doi:10.1007/978-3-540-78499-9\_7.
- [20] Peter Selinger & Benoît Valiron (2009): *Quantum Lambda Calculus*. doi:10.1017/CBO9781139193313.005. *Semantic Techniques in Quantum Computation*.
- [21] William Wootters & Wojciech Zurek (1982): *A single quantum cannot be cloned*. *Nature* 299(5886), doi:10.1038/299802a0.