Notes on Categorical Message Passing Language (Part 1)

Alexanna Little
University of Calgary

Robin Cockett [1]
Department of Computer Science, University of Calgary

Abstract

Categorical Message Passing Language (CaMPL) is a concurrent functional programming language in which the semantics of the programming features are defined using category theory. This document aims to introduce the categorical structures used to define the basic CaMPL programming features and to show the connection between the CaMPL programming language syntax and its categorical semantics. We begin with an introduction of the category theory background starting with the definition of a category and ending with the definition of a linear actegory. Then, we give the basic programming features in CaMPL and describe their categorical semantics, including concurrent processes and communication channels, sequential functions and messages, message-passing on communication channels, and user-defined channel-based session types called protocols and coprotocols. Each programming feature will be discussed from the three perspectives that exist in the Curry-Howard-Lambek correspondence: its categorical semantics, programming syntax, and its representation in proof theory using both a sequent calculus and circuit diagram.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases concurrency, message passing, category theory, non-determinism, progress

Acknowledgements This document is intended for an audience of people with an understanding of functional programming who are interested in learning about category theory and for people with an understanding of category theory to learn about an application of category theory to programming semantics. It is a compilation of reports from my undergraduate CaMPL research projects and includes references to project reports from other students who have also worked on the CaMPL project. The process of compiling these works was possible because Dr. Robin Cockett has dutifully maintained an archive of the work that has been done on CaMPL over many years. While working with Dr. Cockett, I have learned a lot about research and category theory, and I have been introduced to many friendly category theorists who have supported me along the way! Special thanks to Jason Parker and Priyaa Varshinee Srinivasan for their support on this document. Tikzit was used to generate circuit diagrams and quiver and tikzcd were used to generate commutative diagrams.

[1] Supervisor

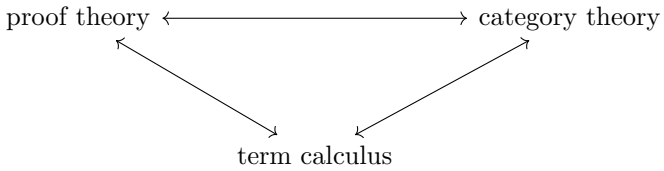
Contents

1 Introduction

Category theory is an area of mathematics in which mathematical structures are defined in terms of relationships that are generally called *maps*. A categorical semantics of functional programming can be obtained by defining programs as maps from their input types to their output types, as described by the Curry-Howard-Lambek correspondence [3, 6, 9]. Categorical Message Passing Language (CaMPL) is a functional-style concurrent programming language which implements concurrency by message passing. A categorical semantics of message passing is described by Cockett and Pastro in "The Logic of Message Passing" [1]. We discuss this and other related previous work on CaMPL in Section 2 to explain the context of the CaMPL project.

We begin the body of this document with an introductory explanation of some fundamental concepts from category theory that are important to this project, for example the italicized words on this page, in Section 3. This is intended to be a simple explanation of relevant concepts in category theory to provide the basics to new category theorists and establish the notation we use to experienced category theorists. It is not meant to be a comprehensive introduction to category theory.

In Section 4, we will explore the basic programming features of CaMPL. We will discuss each programming feature from the three perspectives that exist in the Curry-Howard-Lambek correspondence:



The category theory is the categorical semantics for the programming feature. The proof theory is a sequent calculus representation of a CaMPL program and a circuit diagram notation which gives an intuitive visualization for the semantics. The term calculus is the programming syntax, so this section is also inteded to be a resource for learning how to code using the basic programming features in CaMPL.

CaMPL programs are composed of multiple concurrent processes which communicate with each other by passing messages along communication channels. This is discussed in Section 4.1. Channels are typed, so processes must be plugged together along a channel with a consistent type which will be discussed in Section 4.3. The categorical semantics for how concurrent processes can be connected by communication channels is given by a *linearly distributive category*, the details of which will be discussed in Section 4.4. CaMPL programs have been shown to have an important property called *progress* which guarantees that they will never result in a deadlock or livelock [13]. This is guaranteed by basing the categorical semantics on a linearly distributive category.

The concurrent processes independently run their own sequential functional programs, and sending or receiving a message is an operation they can perform to send an output to or receive an input from another concurrent process. The categorical semantics for sequential computations is given by a *monoidal category*. This is discussed in Section 4.5.

The semantics for CaMPL programs, which includes the sequential messages, concurrent processes, and communication channels, is given by a *linear actegory* [1]. The semantics for passing a message along a channel is given by one of two *actions* in the linear actegory [1]. Which of the two actions should be used depends on the "direction" the message is being

passed on the channel, so if one action is used for receiving a message, the other is used for sending a message. This is discussed in Section 4.6.

A process can also initiate a session of message passing with another process according to a *protocol* or *coprotocol* on the communication channel between them. These are special concurrent channel types which can have recursive typed interactions between processes. In this way, protocols and coprotocols are related to session types. The difference between a protocol and a coprotocol is also the "direction" it is initiated because of the distinction between the directions messages are passed [8]. The semantics for protocols and coprotocols are given by *inductive and coinductive datatypes* [18]. This is discussed in Section 4.7.

Finally, we conclude and discuss in-progress and future work on the CaMPL project in Section 5.

2 Related Work

In this section, we mainly discuss direct contributions to CaMPL by authors who have previously worked on this project. Specifically, previous works that defined the semantics for message passing, showed that it guarantees CaMPL programs will not have deadlocks or livelocks, and defined the semantics for session types. We will also mention contributions to CaMPL beyond these basic programming features including non-determinism and higher order processes. However, these will not be the focus of this document.

The semantics of CaMPL was originally established in Cockett and Pastro's paper "The Logic of Message Passing" in which the primary motivation was to define a concurrent analogue of the Curry-Howard-Lambek correspondence [1]. This paper described a two-tiered logic consisting of a sequential part, called the message logic, that interacts with a concurrent part, called the message-passing logic, in which concurrent processes pass messages of sequential data back and forth along communication channels. That is, the two-tierd logic describes a concurrent system that uses message passing to exchange information between processes – as opposed to a concurrent system that uses shared memory to exchange information between processes. The categorical semantics of the message-passing logic is given by a linear actegory in which the category of messages and sequential functions *acts* on the category of communication channels and concurrent processes in two directions.

Cockett and Pastro's paper developed the logic of message passing based on the multiplicative fragment of linear logic. In Girard's paper "Linear Logic," it was proposed that linear logic could give the sematics of concurrent programming [5]. Additionally, the use of "proof nets" to visually express proofs in linear logic was introduced. In CaMPL, the semantics for how processes can be connected to each other using channels is based on multiplicative linear logic with linear distributivity – the categorical semantics of which is given by a linearly distributive category [2]. The linear distribution of the multiplicative logical connectives ensures that any valid configuration of processes connected by channels is acyclic, i.e. a graph of the processes that have direct connections in which a message can be sent or received is a tree at any given point of the computation. In "Progress for the Message Passing Logic," Lybbert used this property to show that CaMPL programs can never become deadlocked, i.e. every process is waiting to send or recieve a message from another process, or livelocked, i.e. every process is stuck in a purposeless infinite loop that prevents the intended computation from continuing [13].

The final basic programming feature that we will consider is channel-based session types which are called protocols and coprotocols in CaMPL. A protocol or coprotocol allows one to define a custom channel type with certain handles that each specify a kind of interaction

that the processes can have on that channel. Then, one process will decide which handle will be used, and the other process will respond based on the selected handle. These session types allow one to define fixed-point recursive channel types, so an interaction between processes can be arbitrarily long if the process choosing the handles continues to choose to recurse. In "Linear Functors and their Fixed Points," Yeasin showed that inductive and coinductive datatypes, given by categorical initial algebras and final coalgebras, provide the categorical semantics for protocols and coprotocols [18].

An implementation with the above basic programming features was developed by Kumar in "Implementation of Message Passing Language" [8]. This implementation included a message logic, a message-passing logic, protocols and coprotocols, and examples of their uses. Later, in "Implementation Status of CMPL" and "Redesigning the Abstract Machine for CaMPL," Pon re-implemented CaMPL to improve efficiency and add a new programming feature called *races*, [16, 17]. The programming syntax we use in this document and the default version of CaMPL available on the CaMPL website (`https://campl-ucalgary.github.io/`) is Pon's implementation.

We now breifly describe some additional programming features of CaMPL and their semantics which will not be discussed in the body of this document. As mentioned above, CaMPL has races. Races allow programs to be non-deterministic by "racing" channels they are receiving messages on and following different executions depending on the order they receive the messages. Defining the categorical semantics of non-deterministic processes in CaMPL is in-progress work, but the general idea is to enrich the category of channels and processes in sup-lattices as described by Little in "Semantics for Non-Determinism in the Categorical Message Passing Language" and "Formalizing Non-Determinism in the Categorical Message Passing Language" [10, 11]. The details to show that the linear actegory structure, and therefore the property of progress, are preserved by the sup-lattice enrichment is still in-progress work. For protocols and coprotocols, inductive and coinductive datatypes use products and coproducts which are not preserved by the sup-lattice enrichment. This means that non-deterministic protocols and coprotocols are slightly different than in the determinstic case. The general idea of this is also presented in the previous work by Little, and the details are still in-progress.

Another feature of CaMPL is the addition of higher-order processes described in "Higher-Order Message Passing in CaMPL" by Norouzbeygi [15]. This feature allows one to store the code of a process, pass it as a message, and use it as another process. At the time of writing, this feature is not in the default version of CaMPL on the website, but it should be soon in the future. Finally, a very early version of a feature that allows CaMPL programs to run on different devices and connect over a network was developed in "Building Foundations for Secure Networking in CaMPL" by Little [12]. This was a change to the implementation rather than the semantics, and it is also not yet added to the default version of CaMPL.

3 Category Theory Background

The semantics of programming features in CaMPL have been defined using different kinds of categories that have certain categorical structures. This section is a compilation of my notes from when I learned about category theory to understand the semantics. Ideally, it will provide value to the reader as an entry point into category theory or a reminder of the concepts, but it should not necessarily be used as a main source to learn about these categories.

To show a categorical structure is well-defined, one must show that the equations governing

it, called coherence conditions, are satisfied. This may include showing certain data or coherence maps always behave in a certain way and therefore have a "universal property." Coherence conditions are often expressed as diagrams which need to be recreated with the instance of the categorical structure that is being defined. This means that, to define the categorical semantics of a programming feature, we would show that it satisfies the coherence conditions of its corresponding categorical structure. The proofs that show each programming feature is defined by its categorical semantics can be found in the corresponding related work document in which they were originally presented. This document is intended to explain what the categorical semantics are to readers interested in the CaMPL project and point them in the direction of the rigorous resources. To provide context for the descriptions of the categorical semantics of CaMPL, definitions and explanations of the coherence conditions for the categorical structures that are used in the CaMPL semantics have been included in this section.

3.1 Categories

To define a category, it is necessary to define its objects, maps, identity maps, and composition of maps, and show that they satisfy certain coherence conditions. This means that a category $\mathbb{C}$ consists of the following data:

- A class of objects, $Obj(\mathbb{C})$. We can talk about objects $A, B, C, D, \ ... \in Obj(\mathbb{C})$.
- A class of maps between pairs of objects, $Mor(\mathbb{C})$. For $A, B \in Obj(\mathbb{C})$, we can talk about a map $f : A \to B \in Mor(\mathbb{C})$.
- For each object $X \in Obj(\mathbb{C})$, an identity map $1_X : X \to X \in Mor(\mathbb{C})$.
- For each pair of maps $h : X \to Y, \ k : Y \to Z \in Mor(\mathbb{C})$, there is a way to compose them into a single map $hk : X \to Z \in Mor(\mathbb{C})$.

We will discuss the coherence conditions shortly. In general, we will write objects with capital letters and maps with lower case letters. Using this convention, we will avoid ambiguity between objects and maps, so we can write $A \in \mathbb{C}$ to mean $A \in Obj(\mathbb{C})$ and $f : A \to B \in \mathbb{C}$ to mean $f \in Mor(\mathbb{C})$.

For any objects $A, B \in \mathbb{C}$, we write $\mathbb{C}(A, B)$ to denote the hom-set of A and B which is the set of all the maps from A to B. We can write $f \in \mathbb{C}(A, B)$ to indicate that it has the type $f : A \to B$. There may be no maps between A and B in which case the hom-set $\mathbb{C}(A, B) = \{\}$ is the empty set. In general, a coherence condition may require that there is only one map which satisfies it. In this case, it is specified that the map must be a "unique map." This does not mean it is the only map in the hom-set, but it is the only map that satisfies the condition.

A category must have an identity for every object which means that there must be a way to "do nothing" to an object, and the identity is unique. A category also must have composition defined for every pair of maps in the category, and the composition is unique. In other areas of mathematics, the composition fg might be written as $g \circ f$.

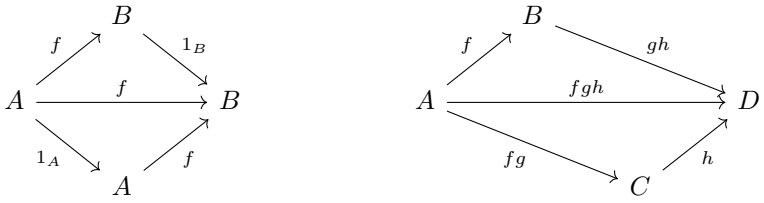
The maps in a category can also be called arrows or morphisms. A map that has another map which "reverses" it is called an isomorphism. The coherence conditions for an isomorphism are that there exists an inverse map and composing them in either direction is the identity. For example, if there was a map $f' : B \to A$ and $ff' = 1_A, f'f = 1_B$, then f is an isomorphism. The inverse of an isomorphism is also unique.

An example of a category is Set, the category of sets and set functions:

- Objects: sets $A, B, C, ... \in Obj(Set)$.
- Maps: set functions $f : A \to B; a \mapsto f(a), ... \in Mor(Set)$.
- Identity: $1_A : A \to A; a \mapsto a \in Mor(Set), \forall A \in Obj(Set)$.

- Composition: $fg : A \to C; a \mapsto g(f(a))$, for $f : A \to B, g : B \to C \in Mor(Set)$.

The coherence conditions for defining a category are that composition must satisfy unit laws (with respect to identities) and associativity. When we say identities and compositions are unique, we mean in terms of these conditions. Composition satifying unit laws means that composing with the identity doesn't change anything, i.e. for $f : A \to B$, $1_A : A \to A$, $1_B : B \to B \in \mathbb{C}$, it must be true that $1_A f = f$ and $f1_B = f$. Composition satisfying associativity means that the order you compose maps doesn't matter, i.e. for $f : A \to B$, $g : B \to C$, $h : C \to D \in \mathbb{C}$, it must be true that $fgh = (fg)h = f(gh)$. These equalities can also be interpreted to mean that the following diagrams must "commute" for all maps in the category:



Returning to our example of the category Set. We can show that the coherence conditions are satisfied.

- Composition is associative. We can show $fgh = (fg)h$ as follows:

$$\begin{aligned} fgh(a) &= h(g(f(a))) && \text{decompose } fgh \text{ by definition} \\ &= h(fg(a)) && \text{compose } f \text{ and } g \\ &= (fg)h(a) && \text{compose } fg \text{ and } h \end{aligned}$$

We can show $fgh = f(gh)$ as follows:

$$\begin{aligned} fgh(a) &= h(g(f(a))) && \text{decompose } fgh \text{ by definition} \\ &= gh(f(a)) && \text{compose } g \text{ and } h \\ &= f(gh)(a) && \text{compose } f \text{ and } gh \end{aligned}$$

Thus, we have shown $fgh = (fg)h = f(gh)$ for $f : A \to B, g : B \to C, h : C \to D \in Set$.

- Composition is unital. We can show $1_A f = f$ as follows:

$$\begin{aligned} 1_A f(a) &= f(1_A(a)) \\ &= f(a) \end{aligned}$$

We can show $f1_B = f$ as follows:

$$\begin{aligned} f1_B(a) &= 1_B(f(a)) \\ &= f(a) \end{aligned}$$

Thus, we have shown $1_A f = f$ and $f1_B = f$ for $f : A \to B, 1_A, 1_B \in Set$.

3.2 Functors

A functor $F : \mathbb{C} \to \mathbb{D}$ is a map between categories $\mathbb{C}$ and $\mathbb{D}$. We define how a functor acts on objects and arrows, and the coherence conditions are that a functor must preserve the

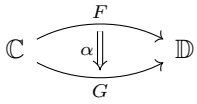
direction of arrows between objects, identities, and composition. An example of each of these conditions for F, with $\mathbb{C}$ as described above, are that the following must be true in $\mathbb{D}$:

$$F(f) : F(A) \to F(B)$$
$$F(1_A) = 1_{F(A)} : F(A) \to F(A)$$
$$F(fg) = F(f)F(g) : F(A) \to F(C)$$

A functor as described above is called a *covariant* functor. The category $\mathbb{C}^{op}$ is called the *opposite* category of $\mathbb{C}$ in which the objects, identities, and composition are the same, but the direction of arrows (maps) is reversed. That is, for an arrow $f : A \to B \in \mathbb{C}$, there is an arrow $f : B \to A \in \mathbb{C}^{op}$. A covariant functor $F' : \mathbb{C}^{op} \to \mathbb{D}$ from the opposite category of $\mathbb{C}$ is called a *contravariant* functor from $\mathbb{C}$ to $\mathbb{D}$. An endofunctor is a functor that maps a category to itself, so $E : \mathbb{C} \to \mathbb{C}$ is an endofunctor on $\mathbb{C}$. Another example of a category is **Cat**, the category of categories, in which the objects are categories and the morphisms are functors.

3.3 Natural Transformations

A natural transformation $\alpha : F \Rightarrow G$ is a map between functors $F : \mathbb{C} \to \mathbb{D}$ and $G : \mathbb{C} \to \mathbb{D}$:



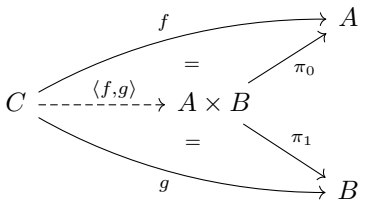
We define α as a family of component maps $\alpha_X,\ \forall X \in \mathbb{C}$, such that the following naturality square diagram commutes for all $X, Y,\ f : X \to Y \in \mathbb{C}$:

$$\begin{array}{ccc} F(X) & \xrightarrow{\alpha_X} & G(X) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ F(Y) & \xrightarrow[\alpha_Y]{} & G(Y) \end{array}$$

In general, lower case greek letters are often used for natural transformations. In this document, we will discuss specific natural transformations for a given categorical structure, and will likely use lower case latin letters instead.

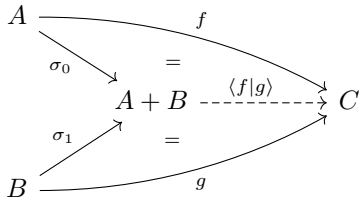
3.4 Products and coproducts

A product is a structure in a category $\mathbb{C}$ on two objects, $A, B, \in \mathbb{C}$ which consists of the product object $A \times B$ of A and B and projection maps $\pi_0 : A \times B \to A$ and $\pi_1 : A \times B \to B$. For any pair of maps $f : C \to A, g : C \to B$, products have a unique product map $\langle f, g \rangle$ satisfying the universal property that $\langle f, g \rangle \pi_0 = f$ and $\langle f, g \rangle \pi_1 = g$ so the following diagram commutes:



In *Set*, this is the cartesian product.

A coproduct is a structure on two objects, $A, B, \in \mathbb{C}$ which consists of the coproduct object $A + B$ of A and B and injection maps $\sigma_0 : A \to A + B$ and $\sigma_1 : B \to A + B$. For any pair of maps $f : A \to C, g : B \to C$, coproducts have a unique coproduct map $\langle f \mid g \rangle$ satisfying the couniversal property that $\sigma_0\langle f \mid g \rangle = f$ and $\sigma_1\langle f \mid g \rangle = g$ so the following diagram commutes:



In Set, this is the disjoint union.

3.5 Monoidal Categories

A monoidal structure, or tensor product structure, $(\otimes, \top, a, u^L, u^R)$ on a category $\mathbb{X}$ is a functor $\otimes$, called the tensor, with unit object $\top$, and natural isomorphisms a, called the associator, u^L called the left unitor, and u^R called the right unitor. These data have the following types with $X, Y, Z \in \mathbb{X}$:

$$\begin{aligned} \otimes &: \mathbb{X} \times \mathbb{X} \to \mathbb{X} \\ \top &\in \mathbb{X} \\ a_{X,Y,Z} &: (X \otimes Y) \otimes Z \to X \otimes (Y \otimes Z) \\ u^L_X &: \top \otimes X \to X \\ u^R_X &: X \otimes \top \to X \end{aligned}$$

The natural isomorphisms must satisfy the MacLane pentagon (1) and Kelly triangle (2) coherence diagrams with $W, X, Y, Z \in \mathbb{X}$:

$$\begin{array}{ccc} ((W \otimes X) \otimes Y) \otimes Z & \xrightarrow{a_{W,X,Y} \otimes 1_Z} & (W \otimes (X \otimes Y)) \otimes Z \\ \downarrow{\scriptstyle a_{W \otimes X, Y, Z}} & & \downarrow{\scriptstyle a_{W, X \otimes Y, Z}} \\ (W \otimes X) \otimes (Y \otimes Z) & & W \otimes ((X \otimes Y) \otimes Z) \\ & {\scriptstyle a_{W,X,Y \otimes Z}} \searrow \quad \swarrow {\scriptstyle 1_W \otimes a_{X,Y,Z}} & \\ & W \otimes (X \otimes (Y \otimes Z)) & \end{array} \tag{1}$$

$$\begin{array}{ccc} (X \otimes \top) \otimes Y & \xrightarrow{a_{X,\top,Y}} & X \otimes (\top \otimes Y) \\ & {\scriptstyle u^R_X \otimes 1_Y} \searrow \quad \swarrow {\scriptstyle 1_X \otimes u^L_Y} & \\ & X \otimes Y & \end{array} \tag{2}$$

We can summarize these conditions by saying that the tensor $\otimes$ is associative and follows unit laws [7]. If a category $\mathbb{X}$ has a tensor with a unit such that these coherence conditions are satisfied, we say $(\mathbb{X}, \otimes, \top, a, u^L, u^R)$ is a monoidal category.

A symmetric monoidal category is a monoidal category with an additional natural isomorphism c called a commutator, commutativity isomorphism, or symmetry map. In a symmetric monoidal category, the tensor is commutative. The commutator has the type

$c_{X,Y} : X \otimes Y \to Y \otimes X$ such that $c_{X,Y} c_{Y,X} = 1_{X \otimes Y} : X \otimes Y \to X \otimes Y$, and it must satisfy the following hexagon coherence diagram (3), with $X, Y, Z \in \mathbb{X}$ [14]:

$$\begin{array}{ccccc} X \otimes (Y \otimes Z) & \xrightarrow{a_{X,Y,Z}} & (X \otimes Y) \otimes Z & \xrightarrow{c_{X \otimes Y, Z}} & Z \otimes (X \otimes Y) \\ {\scriptstyle 1_X \otimes c_{Y,Z}}\downarrow & & & & \downarrow{\scriptstyle a_{Z,X,Y}} \\ X \otimes (Z \otimes Y) & \xrightarrow[a_{X,Z,Y}]{} & (X \otimes Z) \otimes Y & \xrightarrow[c_{X,Z} \otimes 1_Y]{} & (Z \otimes X) \otimes Y \end{array} \tag{3}$$

A distributive monoidal category is a monoidal category with coproducts over which the tensor distributes. This means that there exists a natural isomorphism $\langle 1_A \otimes \sigma_0 \mid 1_A \otimes \sigma_1 \rangle$ for $A, X, Y \in \mathbb{X}$ [1]:

$$\langle 1_A \otimes \sigma_0 \mid 1_A \otimes \sigma_1 \rangle : A \otimes X + A \otimes Y \to A \otimes (X + Y) \tag{4}$$

3.6 Linearly Distributive Categories

A linearly distributive category $\mathbb{X}$, originally called a weakly distributive category in [2], has two monoidal structures: tensor $(\otimes, \top, a_\otimes, u^L_\otimes, u^R_\otimes)$ and par $(\oplus, \bot, a_\oplus, u^L_\oplus, u^R_\oplus)$. These monoidal structures must each satisfy the pentagon diagram in (1) and triangle diagram in (2). Additionally, it has natural transformations δ^L and δ^R, called the left and right distributors, which have the following types:

$$\delta^L_{X,Y,Z} : X \otimes (Y \oplus Z) \to (X \otimes Y) \oplus Z \qquad \delta^R_{X,Y,Z} : (X \oplus Y) \otimes Z \to X \oplus (Y \otimes Z)$$

This is a linear distribution where linear means that X, Y, Z cannot be duplicated, changed, or deleted. The distributors are not isomorphisms, so they are not reversible.

There are numerous coherence conditions for a linearly distributive category which have been dutifully presented in [2]. We will simply summarize the kinds of coherence conditions related to interactions with the distributors. In general, interactions with the distributors should ensure the natural transformations δ^L and δ^R distribute the tensor $\otimes$ linearly over the par $\oplus$. The [op'] and [$\odot$'] symmetries allow us to describe the coherence conditions using a representative diagram for each kind of interaction. The [op'] symmetry reverses arrows, swaps the $\otimes$ and $\oplus$, and swaps the units $\top$ and $\bot$, giving the following assignments:

$$\begin{array}{llll} \delta^L \leftrightarrow \delta^R & a_\otimes \mapsto a_\oplus^{-1} & a_\oplus \mapsto a_\otimes^{-1} & \\ u^L_\otimes \mapsto (u^L_\oplus)^{-1} & u^L_\oplus \mapsto (u^L_\otimes)^{-1} & u^R_\otimes \mapsto (u^R_\oplus)^{-1} & u^R_\oplus \mapsto (u^R_\otimes)^{-1} \end{array}$$

The [$\odot$'] symmetry reverses the $\otimes$ and $\oplus$, giving the following assignments:

$$\delta^L \leftrightarrow \delta^R \qquad a_\otimes \mapsto a_\otimes^{-1} \qquad a_\oplus \mapsto a_\oplus^{-1} \qquad u^L_\otimes \leftrightarrow u^R_\otimes \qquad u^L_\oplus \leftrightarrow u^R_\oplus$$

The following representative coherence diagrams, and all variations given by the symmetries must be satisfied.

Interactions between a distributor and a unitor are represented by (5). Eliminating or introducing a unit must be the same before and after a distribution.

$$\begin{array}{ccc} \top \otimes (X \oplus Y) & & \\ {\scriptstyle \delta^L}\downarrow & \searrow^{u^L_\otimes} & \\ (\top \otimes X) \oplus Y & \xrightarrow[u^L_\otimes \oplus 1_Y]{} & X \oplus Y \end{array} \tag{5}$$

Interactions between a distributor and an associator are represented by (6). Different associations of tensors and pars allow distributions over multiple layers to be done as one distribution or individually.

$$\begin{array}{ccc} (W \otimes X) \otimes (Y \oplus Z) & \xrightarrow{a_\otimes} & W \otimes (X \otimes (Y \oplus Z)) \\ & & \downarrow{\scriptstyle 1_W \otimes \delta^L} \\ \downarrow{\scriptstyle \delta^L} & & W \otimes ((X \otimes Y) \oplus Z) \\ & & \downarrow{\scriptstyle \delta^L} \\ ((W \otimes X) \otimes Y) \oplus Z & \xrightarrow[a_\otimes \oplus 1_Z]{} & (W \otimes (X \otimes Y)) \oplus Z \end{array} \tag{6}$$

Interactions between the left and right distributors are represented by (7). Left and right distributions can be done in either order depending on the association.

$$\begin{array}{ccc} & (W \oplus X) \otimes (Y \oplus Z) & \\ {\scriptstyle \delta^L}\swarrow & & \searrow{\scriptstyle \delta^R} \\ ((W \oplus X) \otimes Y) \oplus Z & & W \oplus (X \otimes (Y \oplus Z)) \\ \downarrow{\scriptstyle \delta^R \oplus 1_Z} & & \downarrow{\scriptstyle 1_W \oplus \delta^L} \\ (W \oplus (X \otimes Y)) \oplus Z & \xrightarrow[a_\oplus]{} & W \oplus ((X \otimes Y) \oplus Z) \end{array} \tag{7}$$

A symmetric linearly distributive category is the case where tensor and par are symmetric with commutators $c_\otimes$ and $c_\oplus$. Each symmetric monoidal structure must satisfy the hexagon diagram in (3). In a symmetric linearly distributive category, we can obtain permuting distributors δ'^L and δ'^R, which have the following types:

$$\delta'^L_{X,Y,Z} : X \otimes (Y \oplus Z) \to Y \oplus (X \otimes Z) \qquad \delta'^R_{X,Y,Z} : (X \oplus Y) \otimes Z \to (X \otimes Z) \oplus Y$$

Each square in the following coherence diagram (8) for the permuting distributors must commute:

$$\begin{array}{ccccccc} X \otimes (Y \oplus Z) & \xrightarrow{1_X \otimes c_\oplus} & X \otimes (Z \oplus Y) & \xrightarrow{c_\otimes} & (Z \oplus Y) \otimes X & \xrightarrow{c_\oplus \otimes 1_X} & (Y \oplus Z) \otimes X \\ \downarrow{\scriptstyle \delta^L} & & \downarrow{\scriptstyle \delta'^L} & & \downarrow{\scriptstyle \delta^R} & & \downarrow{\scriptstyle \delta'^R} \\ (X \otimes Y) \oplus Z & \xrightarrow[c_\oplus]{} & Z \oplus (X \otimes Y) & \xrightarrow[1_Z \oplus c_\otimes]{} & Z \oplus (Y \otimes X) & \xrightarrow[c_\oplus]{} & (Y \otimes X) \oplus Z \end{array} \tag{8}$$

In the symmetric case, there are additional assignments for the permuting distributors in the previous [op'] and [$\odot$'] symmetries and two additional symmetries, [$\otimes$'] that only reverses the tensor and [$\oplus$'] that only reverses the par, which generate more forms of the previous diagrams. The details for the symmetries can be found in [2]. Additionally, the following representative coherence diagram and all of it variations given by the symmetries must be satisfied. Interactions between a distributor and a permuting distributor are represented by (9). A distribution and a permuting distribution can be done in either order depending on the association.

$$\begin{array}{ccc} W \otimes ((X \oplus Y) \oplus Z) & \xrightarrow{1_W \otimes a_\oplus} & W \otimes (X \oplus (Y \oplus Z)) \\ \downarrow{\scriptstyle \delta^L} & & \downarrow{\scriptstyle \delta'^L} \\ (W \otimes (X \oplus Y)) \oplus Z & & X \oplus (W \otimes (Y \oplus Z)) \\ \downarrow{\scriptstyle \delta'^L \oplus 1_Z} & & \downarrow{\scriptstyle 1_X \oplus \delta^L} \\ (X \oplus (W \otimes Y)) \oplus Z & \xrightarrow[a_\oplus]{} & X \oplus ((W \otimes Y) \oplus Z) \end{array} \tag{9}$$

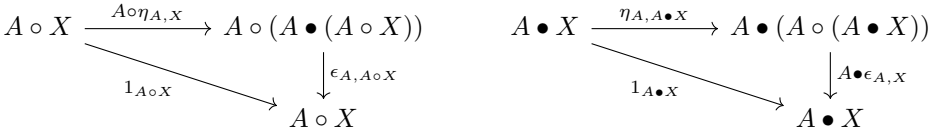
3.7 Linear Actegories

A symmetric linear $\mathbb{A}$-actegory consists of a symmetric monoidal category $(\mathbb{A}, *, I, a_*, u_*^L, u_*^R, c_*)$; a symmetric linearly distributive category $(\mathbb{X}, \otimes, \top, a_\otimes, u_\otimes^L, u_\otimes^R, c_\otimes, \oplus, \bot, a_\oplus, u_\oplus^L, u_\oplus^R, c_\oplus, \delta^L, \delta^R)$; and two action functors $\circ : \mathbb{A} \times \mathbb{X} \to \mathbb{X}$ and $\bullet : \mathbb{A}^{op} \times \mathbb{X} \to \mathbb{X}$ which allow the monoidal category to act on the linearly distributive category in two directions. The action functors form a left parameterized adjunction $A \circ - \dashv A \bullet - : \mathbb{X} \to \mathbb{X}$ with unit and counit, $\forall A \in \mathbb{A}, \forall X \in \mathbb{X}$ [1]:

$$\eta_{A,X} : X \to A \bullet (A \circ X) \qquad \epsilon_{A,X} : A \circ (A \bullet X) \to X$$

The triangle equalities for the parameterized adjunction are:

$$A \circ \eta_{A,X} \; \epsilon_{A,A\circ X} = 1_{A\circ X} : A \circ X \to A \circ X \qquad \eta_{A,A\bullet X} \; A \bullet \epsilon_{A,X} = 1_{A\bullet X} : A \bullet X \to A \bullet X$$



There are also a number of unitor and associator natural isomorphisms, distributive natural morphisms, and symmetry morphisms and isomorphisms which must satisfy several coherence conditions. The details of these data and coherence conditions are dutifuly presented in [1].

An additive linear actegory is a linear $\mathbb{A}$-actegory that is $\mathbb{A}$-additive. This means the monoidal category $\mathbb{A}$ is a distributive monoidal category which has coproducts that the tensor distributes over according to the map described in (4), and the covariant action $\circ$ preserves coproducts and the contravariant action $\bullet$ turns them into products [1]. If we have a coproduct in $\mathbb{A}$ which is shown in Figure 1, the action $\circ : \mathbb{A} \times \mathbb{X} \to \mathbb{X}$ gives a coproduct in $\mathbb{X}$ which is shown in Figure 2. If we have the coproduct in $\mathbb{A}$, we have a product in $\mathbb{A}^{op}$

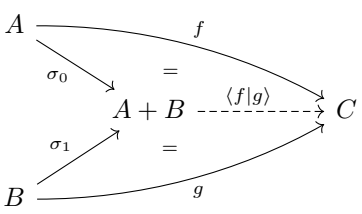


Figure 1 Coproduct in $\mathbb{A}$

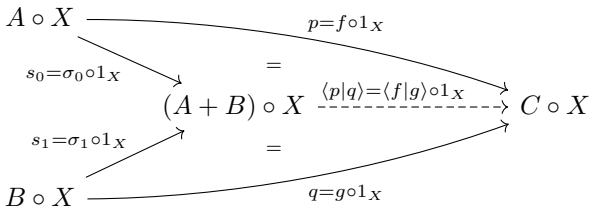


Figure 2 Coproduct in $\mathbb{X}$

which is shown in Figure 3 and the action $\bullet : \mathbb{A}^{op} \times \mathbb{X} \to \mathbb{X}$ gives a product in $\mathbb{X}$ which is shown in Figure 4.

3.8 Initial and Final Algebras

Inductive and coinductive datatypes are datatypes that can be constructed inductively with a fold on an initial datatype or coinductively with an unfold on a final codatatype [18]. Examples of initial datatypes include natural numbers, lists, and trees, and examples of final datatypes include infinite lists and conatural numbers (which count down from infinity).

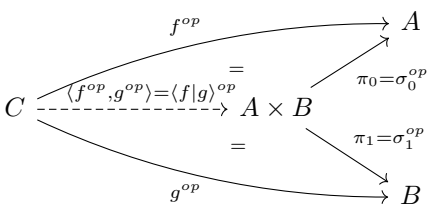


Figure 3 Product in $\mathbb{A}^{op}$

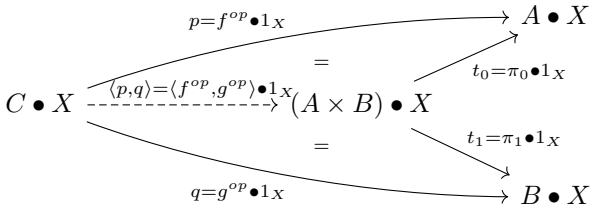


Figure 4 Product in $\mathbb{X}$

Inductive and coinductive datatypes can be formalized as F-algebras and F-coalgebras for a functor $F : \mathbb{X} \to \mathbb{X}$ [4].

An F-algebra for a functor $F : \mathbb{X} \to \mathbb{X}$ is a pair of an object and a map $(A, f : F(A) \to A)$ in $\mathbb{X}$. An initial algebra for F is a pair $(F^{@}, cons : F(F^{@}) \to F^{@})$ which satisfies the universal property that for every F-algebra there exists a unique map $f' : F^{@} \to A$ which makes the diagram below commute. In the context of inductive data, the unique map from the initial object $F^{@}$ to an inductive datatype A is a fold over the initial datatype which replaces its constructor with the constructor for A, so $f' := fold(f)$.

$$\begin{array}{ccc} F(F^{@}) & \xrightarrow{cons} & F^{@} \\ {\scriptstyle F(f')}\downarrow & = & \downarrow{\scriptstyle f'=fold(f)} \\ F(A) & \xrightarrow[f]{} & A \end{array}$$

An F-coalgebra for a functor $F : \mathbb{X} \to \mathbb{X}$ is a pair of an object and a map $(A, g : A \to F(A))$ in $\mathbb{X}$. A final coalgebra for F is a pair $(F_{@}, dest : F_{@} \to F(F_{@}))$ which satisfies the couniversal property that that for every F-coalgebra there exists a unique map $g' : A \to F_{@}$ which makes the diagram below commute. In the context of coinductive data, the unique map from A to the final object $F_{@}$ is an unfold over a coinductive datatype A which replaces its destructor with the destructor for the final object, so $g' := unfold(g)$.

$$\begin{array}{ccc} A & \xrightarrow{g} & F(A) \\ {\scriptstyle unfold(g)=g'}\downarrow & = & \downarrow{\scriptstyle F(g')} \\ F_{@} & \xrightarrow[dest]{} & F(F_{@}) \end{array}$$

We will consider an example of an initial datatype for the functor $F(X) = X + 1$ which is the pair of the natural numbers object and its constructor, the successor or zero function, $(\mathbb{N}, \langle succ \mid zero\rangle)$. An arbitrary F-algebra is a pair of an inductive datatype and its constructor $(A, \langle f \mid a\rangle)$. The unique map from the natural numbers object to A is a fold over the natural numbers which replaces the constructors, so $f' := fold(\langle f \mid a\rangle)$. A fold over the natural number 2, expressed as $succ(succ(zero))$, would look like $fold(\langle f \mid a\rangle) : succ(succ(zero)) \mapsto f(f(a))$. This means that any iterative induction of this form can be expressed as a fold over the natural numbers. The diagram for this initial algebra must commute $\langle succ \mid zero\rangle\ fold(\langle f \mid a\rangle) = (fold(\langle f \mid a\rangle) + 1)\ \langle f \mid a\rangle$ as follows:

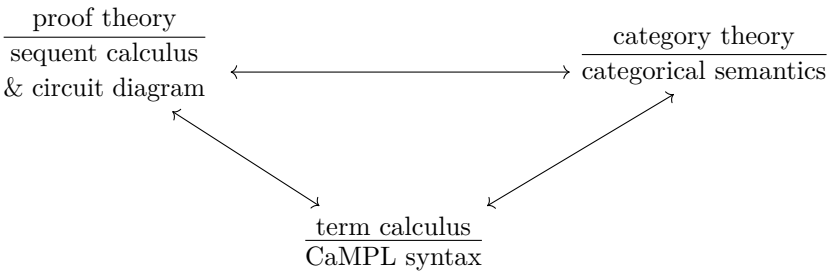
$$\begin{array}{ccc} \mathbb{N}+1 & \xrightarrow{\langle succ|zero\rangle} & \mathbb{N} \\ {\scriptstyle fold(\langle f|a\rangle)+1}\downarrow & = & \downarrow{\scriptstyle fold(\langle f|a\rangle)} \\ A+1 & \xrightarrow[\langle f|a\rangle]{} & A \end{array}$$

4 Semantics of Programming Features

CaMPL is a functional-style concurrent programming language which implements concurrency by message passing. A categorical semantics of functional programming can be obtained by defining programs as maps from their input types to their output types. CaMPL programs have sequential functions, which are maps from sequential input types to sequential output types, and concurrent processes which are maps from input polarity channel types to output polarity channel types. Processes can receive messages on a channel and use the information from the message as input for a function. Processes can also send the output from a function as a message along a channel for another process to use. In this section, we will explore the basic programming features in CaMPL which define the interactions of these processes, channels, and messages.

The categorical semantics of the sequential computations that the concurrent processes perform is given by a distributive symmetric monoidal category that we will call $\mathbb{S}$. The categorical semantics for concurrent processes and communication channels is given by a symmetric linearly distributive category that we will call $\mathbb{C}$. The categorical semantics of message passing is given by an additive linear actegory which consists of the distributive symmetric monoidal category $\mathbb{S}$, the symmetric linearly distributive category $\mathbb{C}$, and two actions $\circ : \mathbb{S} \times \mathbb{C} \to \mathbb{C}$ and $\bullet : \mathbb{S}^{op} \times \mathbb{C} \to \mathbb{C}$ that allow $\mathbb{S}$ to act on $\mathbb{C}$ in two directions. The action functors $\circ$ and $\bullet$ allow values from the sequential computations, called *messages*, to act on the channels and processes which gives the semantics for message passing between processes along a channel [1]. Please see Section 3.5 for details about monoidal categories, Section 3.6 for linearly distributive categories, and Section 3.7 for linear actegories.

We will explore the categorical semantics and programming syntax for the programming features related to each of these main parts of the programming language. Additionally, the programming features in CaMPL can be described equivalently by the programming syntax, circuit diagrams, and sequent calculus. These different representations of the categorical semantics constitute a Curry-Howard-Lambek correspondence in which the circuit diagrams and sequent calculus are representations of the proof theory, and the programming language syntax is the term calculus.



We will not be proving this Curry-Howard-Lambek correspondence as this has already been done very thoroughly in [1]. Instead, we will use the Curry-Howard-Lambek correspondence to develop a comprehensive understanding of each programming feature. We will discuss the category theory and programming syntax, and the proof theory representations will provide an intuitive visualization for how they are connected.

4.1 Processes and Channels

We use the term *process* to mean specifically CaMPL processes rather than the concept of processes in general. These are computations running concurrently that can be connected to

each other using *communication channels* (or just *channels*). These channels are typed with concurrent datatypes, and the type of a channel determines the kind of interaction that the processes will have along it.

The categorical semantics of processes and channels defines the valid arrangements of how processes can be connected along channels. We will call $\mathbb{C}$ the **category of processes and channels**. In $\mathbb{C}$, objects are concurrent channel types and morphisms are processes that use channels of certain concurrent types. The identity morphisms are just channels of a certain type, and composition of morphisms is given by plugging processes together along a communication channel. $\mathbb{C}$ is a symmetric linearly distributive category which has two monoidal structures given by $\otimes$ ('tensor') and $\oplus$ ('par'). The $\otimes$ and $\oplus$ functors are used to type channels between more than processes two processes while ensuring the overall arrangement processes connected by channels remains acyclic. This guarantees deadlock and livelock freedom for CaMPL [13]. The units $\top$ ('top') of $\otimes$ and $\bot$ ('bot') of $\oplus$ are actually treated as one unit 'top-bot' in the CaMPL syntax, and it indicates when an interaction on a channel is complete and the processes can close the channel.

A process can be defined in the programming syntax, shown in Figure 6, as a circuit diagram, shown in Figure 5, and in the sequent calculus, shown in Figure 7. In the

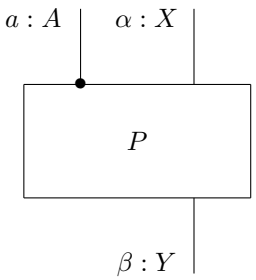


Figure 5 Circuit diagram.

```
proc p :: A | X => Y =
    a | alpha => beta ->
    ...
```

Figure 6 Programming syntax.

$$A \mid X \Vdash Y$$

Figure 7 Sequent calculus.

programming syntax, a concurrent process is defined using the keyword `proc`. The type of the process can be optionally declared after a double colon. The **sequential context** that contains the sequential types is given first. Sequential types will typically be denoted using letters near the beginning of the alphabet such as `A, B, C`. Then, after the bar (`|`), the concurrent channel types are given with the **input polarity** channels separated by `=>` from the **output polarity** channels. Channel types will typically be letters near the end of the alphabet such as `X, Y, Z`. After the optional type declaration and an equals sign, variable names are given to the sequential data and channels that will be used in the process body which is written after the `->` arrow. The process body consists of process commands which will be discussed in the following sections. If a type declaration was included, variable names will be bound to the messages and channels in the same order that they appeared in the type declaration. A process `p` with a message `a` of sequential type `A`, an input polarity channel `alpha` of concurrent type `X`, and an output polarity channel `beta` of concurrent type `Y` is displayed in Figure 6 [8].

In the circuit diagram representation, a concurrent process is depicted as a box which is labeled P in Figure 5. Process P has a message a of sequential type A, an input polarity channel α of concurrent type X, and an output polarity channel β of concurrent type Y. A message is depicted as a line labeled with its type and attached with a $\bullet$ on the top left of the process box. A channel is depicted as a line attached to the process box labeled with the name and type of the channel it represents. Channels attached to the top of a process have

input polarity, and channels attached to the bottom of a process have output polarity.

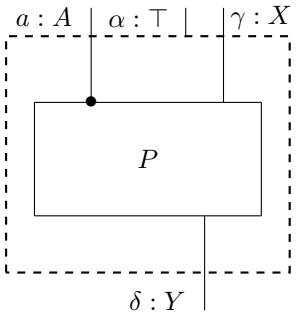
In the sequent calculus, a concurrent process is represented by a sequent which is displayed in Figure 7. The types of the messages in the sequential context are listed on the left of the bar in the sequent. Channels are represented by formulas with the names of their types. Input polarity channels are represented by the antecedent formulas (left of the $\Vdash$ turnstile), and output polarity channels are represented by the succedent formulas (right of the $\Vdash$ turnstile).

The sequent calculus can also be annotated with variable names to specify a certain message or channel of each type which gives additional details that describe how the process operates. An annotated version of the process P using the same variable names would look like this: $a : A \mid \alpha : X \Vdash \beta : Y \;\; :: P$. The annotated sequent calculus forms a term calculus for CaMPL which performs the same role as the programming syntax in the Curry-Howard-Lambek correspondence, and the construction of this term calculus was one of the main results of "The Logic of Message Passing" [1].

4.2 Closing and Halting

To be able to close a channel, the type of the channel must be `TopBot`. To be able to halt a channel, all other channels must already be closed, i.e. the channel being halted must be the last channel left in scope, and it must be of type `TopBot`.

The categorical semantics of this is given by the units $\top$ ('top') of $\otimes$ and $\bot$ ('bot') of $\oplus$. In the CaMPL syntax, we treat them as a single channel type `TopBot`. In Figure 8, we have given an example of closing an input channel `alpha` and then continuing a computation.




```
proc p :: A | TopBot, X => Y =
    a | alpha, gamma => delta ->
          close alpha
          ...
```

$$\frac{A \mid X \Vdash Y}{A \mid \top, X \Vdash Y}\ \top_L$$

Figure 8 Closing a channel.

4.3 Plugging

CaMPL is a concurrent programming language which means we can write programs with multiple processes running at the same time. These processes can communicate with each other while they are running if they are connected by a channel. In $\mathbb{C}$, the objects are channel types and morphisms between channel types are processes that use those channel types. Composition of morphisms gives the categorical semantics for the `plug` process command, which is used to connect processes, and the *cut* rule. An example of the programming syntax to `plug` processes `p` and `q` together along a channel `alpha` is shown in Figure 10. A diagram of processes P and Q plugged together over a channel α of type Y is shown in Figure 9, and the *cut* rule of the sequent calculus is shown in Figure 11.

We can only plug processes together along a channel if they are not already connected by another channel and if they both have an available channel of the same type and opposite

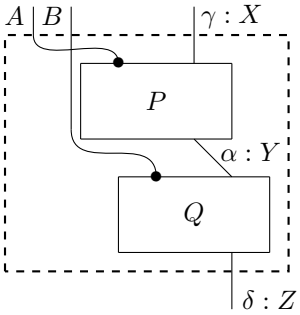


Figure 9 Plugging in a diagram.

```
proc pq :: A, B | X => Z =
    a, b | gamma => delta -> do
        plug
            p(a | gamma => alpha)
            q(b | alpha => delta)
```

Figure 10 The `plug` process command.

$$\frac{A \mid X \Vdash Y \quad B \mid Y \Vdash Z}{A, B \mid X \Vdash Z}\ cut$$

Figure 11 The *cut* rule.

polarity [8]. That is, an available input polarity channel can connect to an available output polarity channel if they have the same type. The type of a channel determines the interaction that processes will have along that channel, for example, the channel type can represent the direction of communication. A channel that plugs processes together must have a consistent type because the processes must be in agreement on the interaction they are having. The channels must be of opposite polarity because channel polarity determines what a process does based on the channel type. After processes agree on the direction of communication on the channel type, the opposite polarity ensures that one process is sending and the other process is receiving. Two processes can only be plugged together over a single channel, but a process can be plugged on multiple channels with different processes at the same time.

4.4 Splitting and Forking Channels

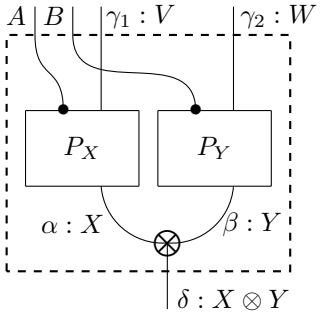
The categorical semantics for plugging multiple processes together are given by the **tensor** $\otimes$ and **par** $\oplus$ functors. These are the functors for the monoidal structures from the linearly distributive category which was discussed in Section 3.6. The sequent calculus rules $\otimes_L, \otimes_R, \oplus_L, \oplus_R$ related to process and channel arrangements are given by the multiplicative fragments of linear logic. Together with the semantics for connecting processes – discussed in the previous section – we can ensure that process arrangements remain acyclic and prevent deadlocks and livelocks from occurring.

The process commands used to manipulate channels and connect multiple processes together are `split` and `fork`. When a channel is split, a process can use both of the channels that are produced to communicate with different processes. When a channel is forked, the channels that are produced must be connected to two different processes.

A channel that has a $\otimes$ type can be split if it has input polarity and forked into two processes if it has output polarity. Dually, a channel that has a $\oplus$ type can be split if it has output polarity and forked into two processes if it has input polarity. This means that a channel cannot be split or forked on both ends because the type of the channel would not allow that. Furthermore, the semantics of plugging processes together ensures channels with different types or the same polarity will not be connected. Together, the semantics ensure that a channel can only be split on one side and forked into two different processes on the other, and this ensures an acyclic arrangement of connected processes is maintained.

In the programming syntax, a channel of type $X \otimes Y$ is written as `X (*) Y`. A process with an output channel of type `X (*) Y` can `fork` it into output channels with types `X` and

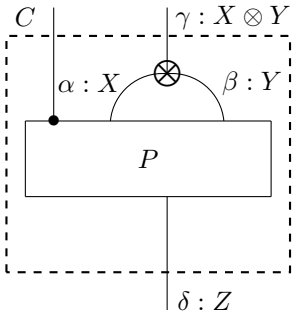
`Y`. These two channels must be used by two different processes in place of the original process as shown in Figure 12. A process with an input channel of type `X (*) Y` can `split` it into two input channels of types `X` and `Y` and continue to use them as shown in Figure 13 [8].




```
proc p' :: A, B | V, W => X (*) Y =
    a, b | gamma_1, gamma_2 => delta -> do
        fork delta as
            alpha -> px(a | gamma_1 => alpha)
            beta -> py(b | gamma_2 => beta)
```

$$\frac{A \mid V \Vdash X \quad B \mid W \Vdash Y}{A, B \mid V, W \Vdash X \otimes Y}\ \otimes_R$$

Figure 12 Fork $\otimes$ output channel.

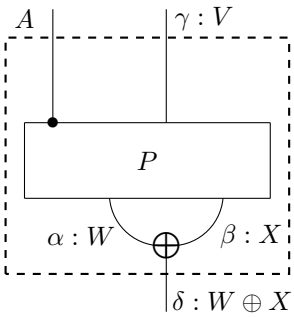



```
proc p' :: C | X (*) Y => Z =
    c | gamma => delta -> do
        split gamma into alpha, beta
        p(c | alpha, beta => delta)
```

$$\frac{C \mid X, Y \Vdash Z}{C \mid X \otimes Y \Vdash Z}\ \otimes_L$$

Figure 13 Split $\otimes$ input channel.

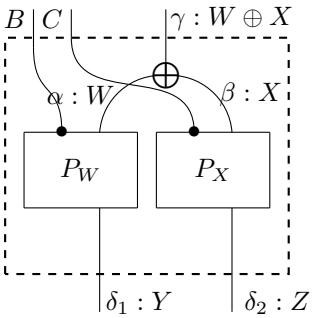
In the programming syntax, a process with an output channel of type $X \oplus Y$ is written as `X (+) Y` and it can be `split` into two output channels with types X and Y as shown in Figure 14. A process with an input channel of type $X \oplus Y$ can `fork` it into input channels with types X and Y which must be used by two different processes as shown in Figure 15.




```
proc p' :: A | V => W (+) X =
    a | gamma => delta -> do
        split delta into alpha, beta
        p(a | gamma => alpha, beta)
```

$$\frac{A \mid V \Vdash W, X}{A \mid V \Vdash W \oplus X}\ \oplus_R$$

Figure 14 Split $\oplus$ output channel.




```
proc p' :: B, C | W (+) X => Y, Z =
    b, c | gamma => delta_1, delta_2 -> do
        fork gamma as
            alpha -> pw(b | alpha => delta_1)
            beta -> px(c | beta => delta_2)
```

$$\frac{B \mid W \Vdash Y \quad C \mid X \Vdash Z}{B, C \mid W \oplus X \Vdash Y, Z} \oplus_R$$

Figure 15 Fork $\oplus$ input channel.

4.5 Messages

Messages are the sequential data used by processes in their internal sequential computations. They can also be passed on channels between processes. The sequential computations processes perform are written with functions and substitutions of message values into those functions.

The categorical semantics defines valid sequential types that messages can have and how they can be used to write functions. We will call $\mathbb{S}$ the **category of messages**. In $\mathbb{S}$, objects are sequential types and morphisms are functions from an input type to an output type. The identity morphism is an identity function, and composition of morphisms is function composition.

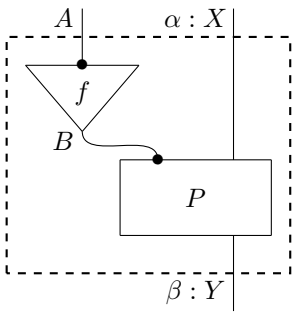
The logic of messages was intentionally kept simple in [1]. It showed that the minimum requirement for $\mathbb{S}$ is a symmetric distributive monoidal category which gives all the functionality that we describe here. However, additional structure can be added, and the sequential part could be any form of sequential functional programming like the typed λ-calculus or Haskell.

The sequential tensor $*$ allows sequential types to be packaged and unpackaged, so functions can take multiple inputs and messages can contain more than one value. The coproduct $+$ allows for sequential datatypes to be constructed from initial datatypes and final codatatypes and for input to make a function execute differently or give different types of output. This gives the semantics for programming features like `if`, `switch`, or `case` statements, and it gives us the ability to write functions that use the actual values of messages to perform computations [1]. Basic or built-in functions, like those included the Prelude, are represented as *non-logical axioms* in the logic of messages. An example of this in the current implementation of CaMPL would be mathematical operators like `+`.

A process can compute a message which is called a **substitution** by using messages in its sequential context as inputs for a function. In the programming syntax, this can be achieved by using a built in function or by defining a function and then calling it with the messages as input which is shown with the diagram representation for this substitution and the sequent calculus *subs* rule in Figure 16.

The rule for function composition is also called *subs*, as shown in Figure 17, but this is a case of passing the output of a function into another function.

The sequential tensor, represented by $*$, allows us to package and unpackage sequential types. This is often done implicitly, and we have already done this in the previous examples of sequential functions that had more than one input value. It is also useful to pass a single message with multiple values rather than multiple single valued messages. In Figure 18, we see that the tensor of sequential types can be unpackaged so both of the types exist in the

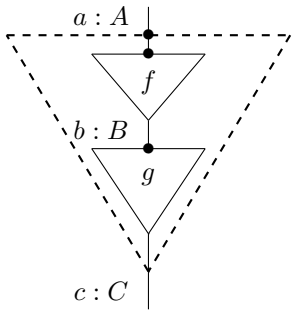



```
fun f :: A -> B
     a -> b

proc p' :: A | X => Y =
     a | alpha => beta -> do
          p(f(a) | alpha => beta)
```

$$\frac{A \vdash B \quad B \mid X \Vdash Y}{A \mid X \Vdash Y}\ subs$$

Figure 16 Computing a message by substitution.




```
fun fg :: A -> C =
    a -> g(f(a))
         ...
```

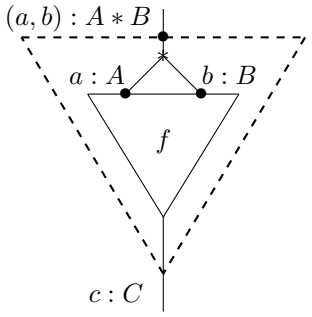
$$\frac{A \vdash B \quad B \vdash C}{A \vdash C}\ subs$$

Figure 17 Function composition substitution.

sequential context and can be used by the process.

The coproduct + is necessary for defining datatypes because a datatype is the coproduct of its constructors. Inductive and coinductive datatypes can be constructed inductively with a fold on an initial datatype or coinductively with an unfold on a final codatatype [18]. This means that their constructors use fixed point recursion in which the fixed point is each application of a constructor. Examples of initial datatypes include natural numbers, lists, and trees, and examples of final datatypes include infinite lists and conatural numbers (which count down from infinity). Inductive and coinductive datatypes can be formalized as F-algebras and F-coalgebras for a functor $F : \mathbb{X} \to \mathbb{X}$ which is discussed in Section 3.8.

The coproduct + also allows a message to change how a process executes and gives the semantics for `if`, `switch`, or `case` statements. In Figure 19, we emphasize how datatypes are the coproduct of their constructors and show how a `case` statement can be used to change what the process does depending on which type is present.




```
fun f :: A, B -> C =
    a, b -> c
        ...
```

$$\frac{A, B \vdash C}{A * B \vdash C} *_L$$

Figure 18 The $*$ rule.

```
data Co(A, B) -> C =
    CoA :: A -> C
    CoB :: B -> C

proc p' :: Co(A, B) | X => Y =
    coprod | alpha => beta -> do
        case coprod of
            CoA(a) -> ...
            CoB(b) -> ...
```

$$\frac{A \mid X \Vdash Y \quad B \mid X \Vdash Y}{A + B \mid X \Vdash Y} +_L$$

Figure 19 The $+$ rule.

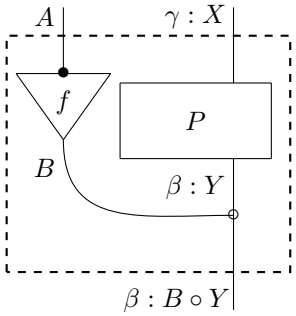
4.6 Message Passing

As mentioned previously, messages can be passed between processes along channels. Messages can be sent and received on both input (polarity) channels and output (polarity) channels, but the types of the channels will be different.

The categorical semantics defines the channel type that is used to pass a message, and it is given by an additive linear actegory that we will call the **message passing actegory**. In the message passing actegory, the distributive monoidal category $\mathbb{S}$ gives the semantics for computing the messages, the linearly distributive category $\mathbb{C}$ gives the semantics for the concurrent processes, and the two actions $\circ : \mathbb{S} \times \mathbb{C} \to \mathbb{C}$ and $\bullet : \mathbb{S}^{op} \times \mathbb{C} \to \mathbb{C}$ give the semantics for passing messages in both directions along channels [1].

As discussed in Section 4.3, the type of a channel defines the interaction processes will have on it, and processes must be connected along a channel of a consistent type with opposite polarities. In the case of $\circ$ ('circ') and $\bullet$ ('bullet') typed channels, the type of the channel determines the direction the messages are travelling. The opposing polarities ensure that when one process is sending, the process on the other end of the channel is receiving. Processes are connected by $\circ$ channels if messages are travelling from output polarity to input polarity (the covariant action) and by $\bullet$ channels if messages are travelling from input polarity to output polarity (the contravariant action).

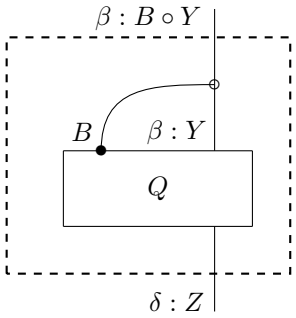
In the programming syntax, a $\circ$ channel (from output to input) has a `Put` type, and a $\bullet$ channel (from input to output) has a `Get` type. The following are examples of sending and receiving messages on both input and output channels in the programming syntax, diagram representation, and sequent calculus.




```
proc p' :: A | X => Put(B|Y) =
    a | gamma => beta -> do
        put f(a) on beta
        p( | gamma => beta)
```

$$\frac{A \vdash B \quad | \; X \Vdash Y}{A \mid X \Vdash B \circ Y} \circ_R$$

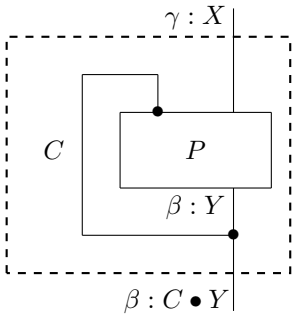
Figure 20 Sending on output channel with a ∘/`Put` type.




```
proc q' :: | Put(B|Y) => Z =
    | beta => delta -> do
        get b on beta
        q(b | beta => delta)
```

$$\frac{B \mid Y \Vdash Z}{\mid B \circ Y \Vdash Z} \circ_L$$

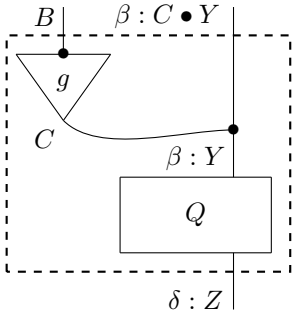
Figure 21 Receiving on input channel with a ∘/`Put` type.




```
proc p' :: | X => Get(C|Y) =
    | gamma => beta -> do
        get c on beta
        p(c | gamma => beta)
```

$$\frac{C \mid X \Vdash Y}{\mid X \Vdash C \bullet Y} \bullet_R$$

Figure 22 Receiving on output channel with a •/`Get` type.




```
proc q' :: B | Get(C|Y) => Z =
    b | beta => delta -> do
        put g(b) on beta
        q( | beta => delta)
```

$$\frac{B \vdash C \quad C \mid Y \Vdash Z}{B \mid C \bullet Y \Vdash Z} \bullet_L$$

Figure 23 Sending on input channel with a •/`Get` type.

The semantics given by the action functors ensure that message passing is defined in a way that behaves logically. The left parameterized adjunction $A \circ - \dashv A \bullet -$ has unit $\eta_{A,X} : X \to A \bullet (A \circ X)$ and counit $\epsilon_{A,X} : A \circ (A \bullet X) \to X$. The unit can be interpreted as defining how a process receiving a message of type A and then immediately sending that message of type A back on the same output channel cancels out. The counit can be interpreted as defining the same idea but occurring on an input channel.

We can express the triangle equalities for the adjunction as a circuit diagram, and we can show that it is satisfied by making rewritings of the circuit diagram. For example, the proof of the equality $A \circ \eta_{A,X} \; \epsilon_{A,A\circ X} = 1_{A\circ X}$ is shown as a circuit diagram rewriting in Figure 24 below.

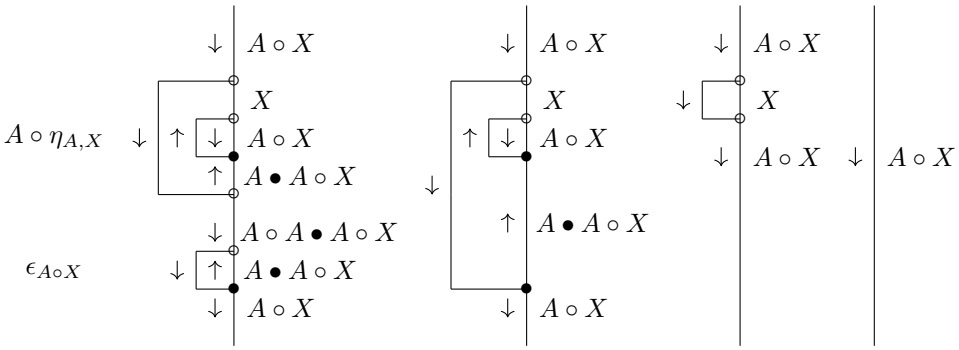


Figure 24 Triangle equality proof.

4.7 Protocols and Coprotocols

In CaMPL, session types are given by special channel types called **protocols** and **coprotocols**. They allow processes to follow predetermined sequences of typed interactions and sustain communication along a channel for an arbitrary number of interactions. When processes are connected by a protocol or coprotocol channel, one process makes an internal choice about which specific typed interaction the channel will carry by sending a *handle* on the channel with the `hput` process command. The other process experiences this as an external choice and uses the `hcase` process command to determine what it should do for each handle that it could be sent. Protocols are initiated by a process sending a handle over an output polarity channel, in the same direction as $\circ$ or `Put` channels, and coprotocols are initiated by a process sending a handle over an input polarity channel, in the same direction as $\bullet$ or `Get` channels. Examples of protocols and coprotocols are included below.

A simple example of a recursive protocol for a channel with type `X` – in which a process with an output polarity channel of type `X` can initiate the protocol by passing the handle `H`, which is communicating that it wants to `put` something of type `A`, or terminate communication by passing the handle `K` – can be defined as follows:

```
protocol
    X( | ) => Z =
        H :: Put(A|Z) => Z
        K :: TopBot => Z
```

Processes that use protocol `X` can be written like this:

```
proc p :: A | Gamma_1 => X, Delta_1 =
```

```
    phi | gamma_1 => alpha, delta_1 -> do
        hput H on alpha
        put f(phi) on alpha
        ...
        hput K on alpha
        halt alpha

proc q :: A | X, Gamma_2 => Delta_2 =
    psi | gamma_2 => alpha, delta_2 -> do hcase alpha of
        H -> do
            get a on alpha
            ...
            q(psi' | gamma_2 => alpha, delta_2)
        K -> do
            halt alpha
```

As shown by this example, process `p` sending a handle on the channel changes the type of the channel to the type of the handle, allows the interaction of that type to proceed, and then allows `p` to change the type of the channel again until `p` terminates the protocol and halts. Process `q` must have a strategy ready for each of the handles it can be sent, and it automatically applies each strategy based on the handle it receives.

A simple example of a recursive coprotocol for a channel with type `Y` – in which a process with an input polarity channel of type `Y` can initiate the coprotocol by passing the handle `H`, which is communicating that it wants to `get` something of type `A`, or terminate communication by passing the handle `K` – can be defined as follows:

```
coprotocol
    Z => Y( | ) =
        H :: Z => Put(A|Z)
        K :: Z => TopBot
```

Processes that use coprotocol `Y` can be written like this:

```
proc p :: A | Gamma_1 => Y, Delta_1 =
    phi | gamma_1 => beta, delta_1 -> do hcase beta of
        H -> do
            put f(phi) on beta
            ...
            q(phi' | gamma_2 => beta, delta_2)
        K -> do
            halt beta

proc q :: A | Y, Gamma_2 => Delta_2 =
    psi | gamma_2 => beta, delta_2 -> do
        hput H on beta
        get a on beta
        ...
        hput K on beta
        halt beta
```

As shown by this example, process `q` sending a handle on the channel changes the type of the channel to the type of the handle, allows the interaction of that type to proceed, and

then allows `q` to change the type of the channel again until `q` terminates the coprotocol and halts. Process `p` must have a strategy ready for each of the handles it can be sent, and it automatically applies each strategy based on the handle it receives.

Recall that the semantics of sequential datatypes is given by inductive and coinductive datatypes, as described in Section 4.5. Protocols and coprotocols are user defined channel types which means that they are concurrent datatypes. Thus, their semantics is given by inductive and coinductive concurrent datatypes [18]. Recursive protocols and coprotocols that allow for an arbitrarily long interaction, like the examples above, are coinductive datatypes, similar to an infinite list, in which we unfold one layer of the interaction at a time until the halt handle is sent and the interaction terminates.

Since protocols and coprotocols are concurrent inductive and coinductive datatypes, the semantics of their handles – the functor of the respective algebra – is given by the coproduct. Recall that coproducts can be used to change how processes behave depending on their value using a `case` statement. This is why the process receiving the handles determines what they should do in response to the handle using an `hcase` statement.

5 Conclusion

We have described the basic programming features in CaMPL and the categorical semantics that define their behaviour. The benefit of using this categorical semantics to define a concurrent message-passing system is that progress is guranteed. This means that the system will never become deadlocked, i.e. every process is waiting to send or recieve a message from another process, or livelocked, i.e. every process is stuck in a purposeless infinite loop that prevents the intended computation from continuing. Since channels are typed according to the categorical semantics, a successful type-check at compile time guarantees the compiled program has progress. The ability to check whether a concurrent system will have progress at compile time is a significant feature of CaMPL, and it has been shown to exist in the basic version of CaMPL we have described. This means that, for any other programming features added to CaMPL, we must develop a categorical semantics and confirm that the property of progess is preserved.

We are currently working on developing a categorical semantics for a version of CaMPL in which non-deterministic processes can be defined. These are processes that use a `race` process command on channels they are waiting to receive messages on and execute differently based on the order in which the messages are received [16]. We are considering a categorical semantics enriched in sup-lattices. This means that, instead of having a set of processes that can be defined between input and output channel types, we would have the power set of the set of deterministic processes and each non-deterministic process would be defined as the subset containing all of its possible executions. Then, to show that non-deterministic CaMPL also has progress, we will define a change of base functor between the deterministic set-enriched semantics and the non-deterministic sup-lattice-enriched semantics and show that the linear actegory structure is preserved.

We also discussed how products and coproducts can be used to define inductive and coinductive datatypes using initial algebras and final coalgebras. Inductive and coinductive datatypes are used to represent protocols and coprotocols, which are very useful features of CaMPL [18]. Products and coproducts are not preserved by a change of enrichment from sets to sup-lattices, which is how we are defining the semantics of non-deterministic CaMPL. Therefore, we also need to define a categorical semantics for protocols and coprotocols in non-determinsitic CaMPL . Future work in this area includes developing a better understanding of

non-deterministic protocols and coprotocols by creating more examples of non-deterministic datatypes and examining what happens to them. This is also related to the development of syntax to write non-determinsitic processes that `race` protocol handles as well as `get` commands [16].

Another version of CaMPL allows one to define higher-order processes in which a process can be stored, passed as a message, and then used by another process. The categorical semantics of higher-order CaMPL is given by a categorical semantics enriched in the category of messages [15]. Future work in this direction might consider how the categorical semantics of non-deterministic processes and higher-order processes are related.

References

1 J. R. B. Cockett and Craig Pastro. The Logic of Message Passing. *Science of Computer Programming*, 74(8):498–533, 2009.

2 J. R. B. Cockett and R. A. G. Seely. Weakly Distributive Categories. *Journal of Pure and Applied Algebra*, 114(2):133–173, 1997.

3 H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584–590, 9 1934. doi:10.1073/pnas.20.11.584.

4 Herman Geuvers. Inductive and coinductive types with iteration and recursion. In *BRA-LF meeting, Edinburgh, Scotland, May 1991*, pages 1–25, 1992.

5 J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

6 William A. Howard. *The Formulae-as-Types Notion of Construction*, pages 479–490. Academic Press, 1980. Original paper manuscript from 1969.

7 G. M. Kelly. Tensor products in categories. *Journal of Algebra*, 2(1):15–37, 1965.

8 Prashant Kumar. Implementation of Message Passing Language. Master's thesis, University of Calgary, Calgary, Alberta, February 2018. URL: https://cspages.ucalgary.ca/~robin/Theses/PrashantKumar.pdf.

9 Joachim Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, Cambridge, New York, Port Chester, 1989.

10 Alexanna Little. Semantics for Non-Determinism in the Categorical Message Passing Language. PURE Final Assignment: Research Findings and Synthesis, University of Calgary, September 2022. URL: https://github.com/campl-ucalgary/campl/blob/main/resources/PURE2022_ResearchFindingsSynthesis_Little.pdf.

11 Alexanna Little. Formalizing Non-Determinism in the Categorical Message Passing Language. Undergraduate thesis, University of Calgary, April 2023. URL: https://github.com/campl-ucalgary/campl/blob/main/resources/CPSC502F22W23_FinalReport_Little.pdf.

12 Alexanna Little. Building Foundations for Secure Networking in CaMPL. Graduate course project report, University of Calgary, February 2025. URL: https://github.com/campl-ucalgary/campl/blob/main/resources/CPSC626_Final_Paper_Little.pdf.

13 Reginald Lybbert. Progress for the Message Passing Logic. Undergraduate thesis, University of Calgary, April 2018. URL: https://github.com/campl-ucalgary/campl/blob/main/resources/ProgressForMPL.pdf.

14 Saunders MacLane. Natural associativity and commutativity. *Rice Institute Pamphlet - Rice University Studies*, 49(4):28–46, 1963.

15 Melika Norouzbeygi. Higher-Order Message Passing in CaMPL. Master's thesis, University of Calgary, Calgary, Alberta, September 2025. URL: https://cspages.ucalgary.ca/~robin/Theses/Melika.pdf.

16 Jared Pon. Implementation Status of CMPL. Undergraduate thesis Interim Report, University of Calgary, December 2021. URL: https://github.com/campl-ucalgary/campl/blob/main/resources/JaredPon_final_report.pdf.

17 Jared Pon. Redesigning the Abstract Machine for CaMPL. Undergraduate thesis, University of Calgary, April 2022. URL: https://github.com/campl-ucalgary/campl/blob/main/resources/JaredPon_final_report.pdf.

18 Masuka Yeasin. Linear Functors and their Fixed Points. Master's thesis, University of Calgary, Calgary, Alberta, December 2012. URL: https://cspages.ucalgary.ca/~robin/Theses/masuka_thesis.pdf.