Semantics for Non-Determinism in the Categorical Message Passing Language

Alexanna Little

August 2022

Pure 2022 Final Assignment: Research Findings and Synthesis
Research Supervisor: Dr. Robin Cockett

Abstract. Non-determinism is introduced in the Categorical Message Passing Language (CaMPL) in the form of races between processes to pass a message to another waiting process. A waiting process internally chooses to proceed with its computation using the first message it receives. The process of creating a categorical model is described as well as the behaviors of CaMPL that we are trying to model. The category of the existing model which comes from Dr. Robin Cockett's previous work, a linear additive actegory, is explained using circuit diagrams and a sequent calculus. Then, the category we are using to model non-determinism in CaMPL is described; a linear actegory enriched over the category of sup-lattices.

1 Introduction

Categorical Message Passing Language (CaMPL) is a concurrent programming language which uses message passing. Category theory was used to prove that programs written in previous versions of CaMPL have a valuable property called "progress" [Lyb18]. In this project, structures in category theory were explored to model non-determinism which has been implemented in the current version of CaMPL [Pon21]. To introduce this project, concurrent programming, progress, message passing, and non-determinism will be briefly discussed.

The applications of concurrent programming are everywhere, so improving the theoretical understanding and tools available to programmers is important. Concurrent Programming is programming where multiple processes perform computations at the same time and might communicate with each other or access common resources. Most useful programs today require some amount of concurrency. An example is a program for a movie theatre to sell tickets for seats to customers. The movie theatre has a process which communicates with each customer's process. A database keeps track of which seats have been assigned, and the movie theatre's process and customer processes can access this information.

Concurrent programs have extra problems which must be avoided: they can deadlock, livelock, or starve a process. A program is deadlocked if every process is waiting for other processes to complete a computation before they can complete their own computation. A program is livelocked if the processes are stuck in a loop where computations which have no external effect are being performed. Starvation occurs in a program when a process is consistently denied resources by the actions of other processes. A program has "progress" when it avoids deadlocks and livelocks, and a motivation for creating CaMPL was to design a language where the concurrent programs will always have progress. Starvation may be avoided by carefully writing programs that schedule when processes access resources, but the semantics of the language developed below is deliberately schedule independent.

Message passing is a way for processes to communicate directly with each other when performing computations by passing messages over channels. For a previous version of CaMPL, it was shown that the key to ensuring a CaMPL program will never deadlock or livelock is in the arrangement of the channels that connect the processes [Lyb18]. Processes must be able to communicate with each other without actually creating cycles. Valid arrangements of channels between processes were modelled by a linearly distributive category where the tensors control when a process can connect channels to other processes.

The focus of this research project was to explore how to model non-determinism in CaMPL. In general, non-determinism is not knowing in advanced how a computation will proceed. In CaMPL, non-determinism is introduced by racing processes: a process waits as multiple other processes race to pass it a message, and it chooses to use the first message it receives when it proceeds with its computations [Pon21]. In the movie tickets example, the theatre does not know in advanced which customer will be assigned a specific seat, so it waits for requests from the customers' processes and it gives a seat to the first process that passes it a message to request the seat.

In the future, we want to explore the effect non-determinism has on the progress of CaMPL programs. Likely, programs will still avoid deadlocks and livelocks however the possibility of starvation will not be avoidable. If a process consistently "loses a race" against other processes, the process will starve. This research project completed the first step of defining the categorical semantics of non-determinism in CaMPL.

2 Process/Methods

The categorical model of CaMPL was created by representing different features of the language using corresponding categorical structures. The existing model of CaMPL is a linear additive actegory where the messages, a distributive monoidal category, act on the processes and channels, a linearly distributive category, when they are passed [CP09]. The action of a message being passed is a functor in the actegory of CaMPL programs. In the category of the messages, sequential types are the objects, computations performed on those sequential types are the morphisms, substitution in the computations is the composition, and new sequential types and datatypes can be built using functors on existing sequential types in the category. In the category of the processes and channels, concurrent types for channels are the objects and processes using channels of those concurrent types are the morphisms. Processes can be plugged together to form new processes which is the composition. New concurrent types and protocols can be built using functors on existing concurrent types in the category [Yea12].

The objective of this project was to find a categorical semantics for non-determinism in CaMPL. Non-determinism arises in CaMPL because a program can have races [Pon21]. Recall the example of selling movie tickets; when customers want the same seat, who gets the seat? A straightforward solution is the customer who requests it first. This is non-determinism because the movie theatre does not know who is going to ask for which seat or when they will ask for it relative to other customers. Customer processes are raced against each other, and the first process that passes a seat request message to the theatre's process wins. The winning customer process is assigned the requested seat, provided it is available, and the race may continue for the customer processes.

We can express a program as a diagram with processes represented as arrows from their input type to their output type. For example, a program X that races processes f and f' with input type A and output type B is represented in Figure 1 below.

We can also represent a program that has a race with one arrow that contains all the options for outcomes of the race. The options for program X are f or f' which are denoted by $f \vee f'$ in Figure 2 below.

A program can also plug processes together over communication channels. The type of the channel on which the processes are plugged together must match with an input type of one process and an output type of the other

$$\begin{tikzcd} A \arrow[r, shift left, "f"] \arrow[r, shift right, "f'"'] & B \end{tikzcd}$$

Figure 1: Program X.

$$\begin{tikzcd} A \arrow[r, "f \vee f'"] & B \end{tikzcd}$$

Figure 2: Program X where the race is denoted by $f \vee f'$.

process. For example, a program Y plugs two processes, g and h, into the process $f \vee f'$ from program X. Process g has input type C and output type A and is plugged into $f \vee f'$ over a channel of type A, and process h has input type B and output type D and is plugged into $f \vee f'$ over a channel of type B. This is represented in Figure 3 below.

$$\begin{tikzcd} C \arrow[r, "g"] & A \arrow[r, "f \vee f'"] & B \arrow[r, "h"] & D \end{tikzcd}$$

Figure 3: Program Y which plugs processes g and h into program X.

We can write the composition of processes in program Y as $g\ (f \vee f')\ h$. And again, we can represent a program that has a race with one arrow that contains all the options given by possible outcomes of the race. There are still only the two options from program X, so the options for program Y are $(gfh)\ \vee\ (gf'h)$. This is represented in Figure 4 below.

The process $g\ (f \vee f')\ h$ must be the same process as $(gfh)\ \vee\ (gf'h)$. The important observation here is that if an interpretation correctly describes these programs, they must be equivalent. Therefore the categorical semantics we seek must represent the composition and options of a non-deterministic program and the equivalence of these different interpretations. Categorically, this is captured by enriching the linear actegory in the category of sup-lattices. The programs are represented by sup-lattices where the non-deterministic options are joins in the sup-lattices and the composition of two

$$\begin{tikzcd} C \arrow[r, "(gfh) \vee (gf'h)"] & D \end{tikzcd}$$

Figure 4: Program Y with options $(gfh) \vee (gf'h)$.

programs is given by a map out of the tensor of two sup-lattices.[1]

[1] Part of how I learned about category theory and the useful properties different categories can have was by attending a conference for Foundational Methods in Computer Science hosted by Dr. Cockett. I was able to learn about category theory from the other participants of the conference who presented their research, and I was able to present the progress I had made so far on this research project. I made many friends with brilliant people from across the world. It was an excellent experience to be immersed in what I am studying, and it advanced my knowledge by leaps and bounds.

3 Findings/Results

The new semantics for CaMPL is built using the existing semantics of deterministic concurrency and adding non-determinism by sup-lattice enriching. First, we will describe the semantics for deterministic concurrency. Second, the semantics for non-deterministic concurrency. This will involve describing sup-lattices and enrichment, and finally, how they are used to model non-determinism.

3.1 Deterministic Concurrency

The interactions of processes and channels and messages in CaMPL can be described by circuit diagrams and a sequent calculus [CP09]. A process in a diagram is depicted as a box which is labeled P in Figure 5 below. It is also represented by a sequent in the sequent calculus which is seen in Figure 6 below.

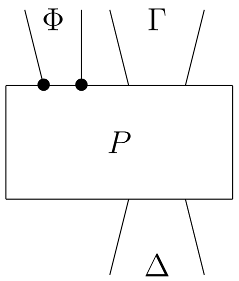


Figure 5: Circuit diagram.

$$\Phi \mid \Gamma \Vdash \Delta$$

Figure 6: Sequent calculus.

A **channel** - or an arbitrary number of channels - is depicted as a line attached to a process box labeled with the type of the channel it represents; channels attached to the top of a process have **input polarity**, in Figure 5 an arbitrary number of input channels are labeled with a Γ, and channels attached to the bottom of a process have **output polarity**, in Figure 5 an arbitrary number of output channels are labeled with a Δ. In the sequent calculus, input polarity channels are represented by the antecedent formulas (left of the $\Vdash$ turnstile in Figure 6) with the names of their types, and output polarity channels are represented by the succedent formulas (right of the $\Vdash$ turnstile in Figure 6) with the names of their types.

A message - or an arbitrary number of messages that constitute the **sequential context** of a process - is depicted as a line labeled with its type, attached with a $\bullet$ on the top left of a process box, in Figure 5 an arbitrary number of messages in the sequential context are labeled with a Φ. In the sequent calculus, the types in the sequential context are listed on the left of the stoup in the concurrent sequent (left of | in Figure 6).

The sequent calculus can be annotated with variable names that specify certain messages or channels of the types to form a term calculus for the language. This is shown in in Figure 8 Diagrams can also be annotated to show which specific messages and channels interact. We will first change the arbitrary amounts of channel and message types to be single ones and then annotate those types. The annotated version of P, with message a that has sequential type A and channels α and β that have concurrent types X and Y, respectively, is shown in Figure 7.

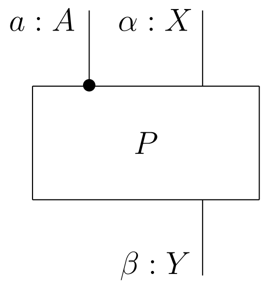


Figure 7: Annotated diagram.

$$a : A \mid \alpha : X \Vdash \beta : Y \ :: P$$

Figure 8: Term calculus.

3.1.1 Plugging Processes together over a Channel

To connect processes so they can communicate, we **plug** them together with a channel which must be the type of an input of one process and an output of the other. Two processes can only be plugged together over a single channel. However, a process can be plugged on multiple channels at once with different processes. A diagram of processes P and Q plugged together over a channel of type X is seen in Figure 9, and the *plug* rule[2] in the sequent calculus is seen in Figure 10 [CS].

[2]This is also called the *cut* rule.

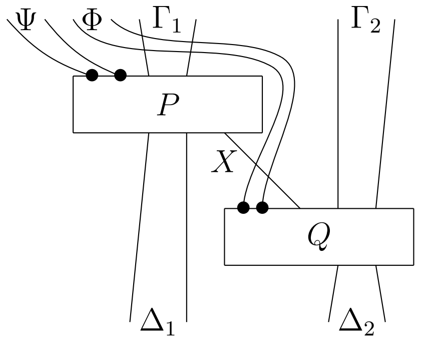


Figure 9: Plugging in a diagram.

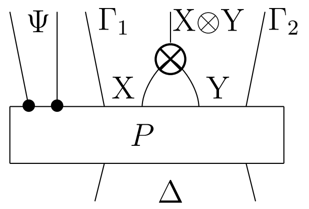
$$\frac{\Phi \mid \Gamma_1 \Vdash \Delta_1, X \quad \Psi \mid X, \Gamma_2 \Vdash \Delta_2}{\Phi, \Psi \mid \Gamma_1, \Gamma_2 \Vdash \Delta_1, \Delta_2}\ \textit{plug}$$

Figure 10: The *plug* rule.

3.1.2 Splitting and Forking Channels

The arrangement of channels connecting processes is constrained by "tensor" $\otimes$ and "par" $\oplus$. The interaction of $\otimes$ and $\oplus$ on the processes and channels form a linearly distributive category [CP09]. The sequent calculus rules for processes and channels are based on linear logic rules [Gir87]. Rules for using $\otimes$ and $\oplus$ in the diagrams and sequent calculus are included below.

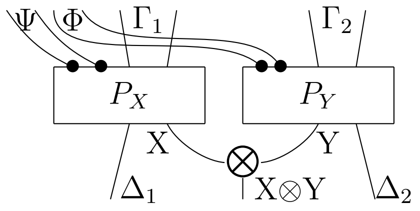
First, the rules for $\otimes$. A process with an input channel of type $X \otimes Y$ can be split into two input channels X and Y which is seen in Figure 11 and Figure 12. A process with an output channel of type $X \otimes Y$ can be forked into output channels X and Y for two different processes which is seen in Figure 13 and Figure 14. The rules for $\oplus$ are similar. A process with an output channel of type $X \oplus Y$ can be split into two output channels X and Y which is seen in Figure 15 and Figure 16. A process with an input channel of type $X \oplus Y$ can be forked into input channels X and Y for two different processes which is seen in Figure 17 and Figure 18. In general, splitting a channel uses $\otimes$ on the input channels and $\oplus$ on the output channels, and forking a channel into two processes uses $\oplus$ on the input channels and $\otimes$ on the output channels.



$$\frac{\Phi \mid \Gamma_1, X, Y, \Gamma_2 \Vdash \Delta}{\Phi \mid \Gamma_1, X \otimes Y, \Gamma_2 \Vdash \Delta} \otimes_L$$

Figure 11: Split input channel.

Figure 12: $\otimes$ on input channels.



$$\frac{\Phi \mid \Gamma_1 \Vdash \Delta_1, X \quad \Psi \mid \Gamma_2 \Vdash Y, \Delta_2}{\Phi, \Psi \mid \Gamma_1, \Gamma_2 \Vdash \Delta_1, X \otimes Y, \Delta_2} \otimes_R$$

Figure 13: Fork output channel.

Figure 14: $\otimes$ on output channels.

3.1.3 Passing messages

Messages are the sequential computations performed by processes to be sent to other processes. The logic of messages could be any programmable language such as the typed lambda calculus. In CaMPL, the logic of the messages is intentionally simple, and the rules form a distributive monoidal category [CP09].

Sequential computations are built with functions and substitutions into those functions. Functions are represented as **non-logical axioms**. The rules for *axiom* are depicted in diagrams as seen in Figure 19 and in the sequent calculus as in Figure 20. Outputs from functions can be used as inputs for other functions which is called a **substitution**. The rules for substitutions are depicted in diagrams as seen in Figure 21 and in the sequent calculus as in Figure 22.

The logic of the messages also include a tensor represented by "star" $*$ and coproduct with $+$. These structures allow messages to influence the processes and channels. In Figure 23 and Figure 24, we see that the product of sequential types can be split into the two types in the sequential context.

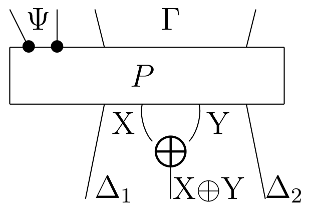


Figure 15: Split output channel.

$$\frac{\Phi \mid \Gamma \Vdash \Delta_1, X, Y, \Delta_2}{\Phi \mid \Gamma \Vdash \Delta_1, X \oplus Y, \Delta_2}\ \oplus_R$$

Figure 16: $\oplus$ on output channels.

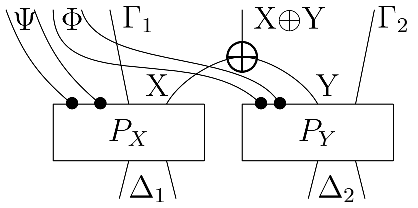


Figure 17: Fork input channel.

$$\frac{\Phi \mid \Gamma_1, X \Vdash \Delta_1 \quad \Psi \mid Y, \Gamma_2 \Vdash \Delta_2}{\Phi, \Psi \mid \Gamma_1, X \oplus Y, \Gamma_2 \Vdash \Delta_1, \Delta_2}\ \oplus_R$$

Figure 18: $\oplus$ on input channels.

In Figure 25 and Figure 26, we see that the coproduct of sequential types will change the process so that it has only one of the types in the sequential context.

Messages are sent or received by processes on channels using the action functors of "white bullet" $\circ$ and "black bullet" $\bullet$. The actions of $\circ$ and $\bullet$ with messages on process channels form a linearly actegory [CP09]. Processes can be connected by $\circ$ channels if messages are travelling from output to input and $\bullet$ channels if messages are travelling from input to output. Rules for using $\circ$ and $\bullet$ in the diagrams and sequent calculus are included below.

First, the $\circ$ rules. The action of receiving a message on an input channel given by $\circ$ is seen in Figure 27 and Figure 28. The action of sending a message on an output channel given by $\circ$ is seen in Figure 29 and Figure 30. The $\bullet$ rules are similar. The action of sending a message on an input channel given by $\bullet$ is seen in Figure 31 and Figure 32. The action of receiving a message on an output channel given by $\bullet$ is seen in Figure 33 and Figure 34. In general, recieving a message uses $\circ$ on an input channel and $\bullet$ on an output channel, and sending a message uses $\circ$ on an output channel and $\bullet$ on an input channel.

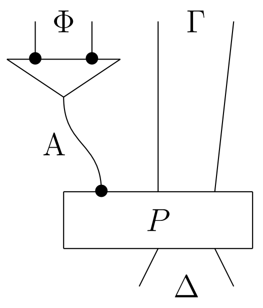


Figure 19: Computing a message.

$$\frac{}{\Phi \vdash A \mid \Gamma \Vdash \Delta}\ axiom$$

Figure 20: A non-logical axiom.

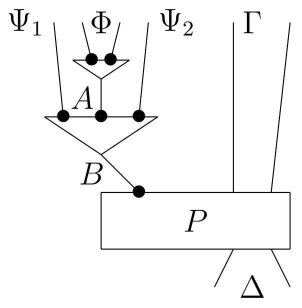


Figure 21: Substitution in a function.

$$\frac{\Phi \vdash A \quad \Psi_1, A, \Psi_2 \vdash B \mid \Gamma \Vdash \Delta}{\Psi_1, \Phi, \Psi_2 \vdash B \mid \Gamma \Vdash \Delta}\ subs$$

Figure 22: The substitution rule.

3.1.4 Session Types

Session types are a predetermined set of interactions on a channel [Pfe18]. In CaMPL, they are called protocols and coprotocols, and they allow for two processes to sustain an indefinite connection using certain channel types for interactions during the session. Protocols are sent by a process over an output polarity channel, and coprotocols are sent by a process over an input polarity channel [Yea12]. For the movie theatre example, the movie theatre process may send a protocol to a customer process that allows the customer to respond to request available seats, book a certain seat, or end the session.

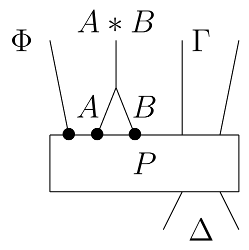


Figure 23: Product puts both in context.

$$\frac{\Phi, A, B \mid \Gamma \Vdash \Delta}{\Phi, A * B \mid \Gamma \Vdash \Delta} *$$

Figure 24: The $*$ rule.

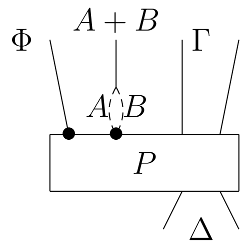


Figure 25: Coproduct puts one of them in context.

$$\frac{\Phi A \mid \Gamma \Vdash \Delta \quad \Phi, B \mid \Gamma \Vdash \Delta}{\Phi, A + B \mid \Gamma \Vdash \Delta} +$$

Figure 26: The $+$ rule.

3.2 Non-deterministic Concurrency

We add non-determinism to the CaMPL semantics by enriching the linear actegory in the category of sup-lattices.

3.2.1 Sup-lattices

A sup-lattice can either be defined in terms of a set of elements and a join operation on the elements, or as a partially ordered set and a supremum operation on the elements. Both definitions will be discussed and then they will be connected and expanded to show how the functionality we want can be expressed.

First, the definition with a join operation. A sup-lattice, $\mathcal{L}$, can be defined with a carrier L which is a set, a bottom element $\perp \in L$, and a join operation $\vee$, as follows:

$$\mathcal{L} = (L, \perp, \vee)$$

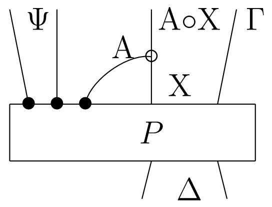


Figure 27: Receiving on input channel.

$$\frac{\Phi, A \mid X, \Gamma \Vdash \Delta}{\Phi \mid A \circ X, \Gamma \Vdash \Delta} \circ_L$$

Figure 28: The $\circ_L$ rule.

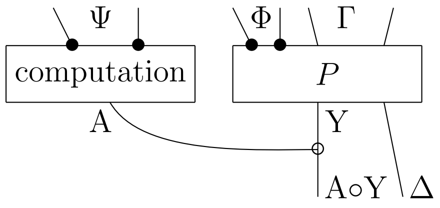


Figure 29: Sending on output channel.

$$\frac{\Psi \vdash A \quad \Phi \mid \Gamma \Vdash Y, \Delta}{\Psi, \Phi \mid \Gamma \Vdash A \circ Y, \Delta} \circ_R$$

Figure 30: The $\circ_R$ rule.

The join operation, $\vee$, is a map from elements in L to an element in L. For two elements in L, the type of $\vee$ can be expressed as:

$$\vee : (L \times L) \to L$$

A sup-lattice must satisfy the following identities, with $\bot, x, y, z \in L$:

unit	$x \vee \bot = x = \bot \vee x$
commutative	$x \vee y = y \vee x$
associative	$(x \vee y) \vee z = x \vee (y \vee z)$
idempotent	$x \vee x = x$

Second, the definition with a poset (partially ordered set) and a supremum operation. A poset, P, is given by a relation $\leq$ that compares elements in P:

$$\leq \quad \subseteq P \times P$$

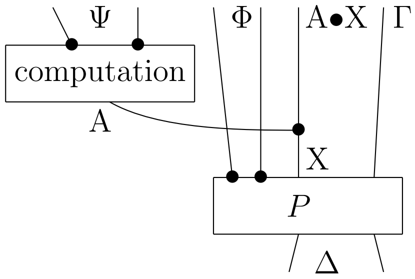


Figure 31: Sending on input channel.

$$\frac{\Psi \vdash A \quad \Phi \mid X, \Gamma \Vdash \Delta}{\Psi, \Phi \mid A \bullet X, \Gamma \Vdash \Delta} \bullet_L$$

Figure 32: The $\bullet_L$ rule.

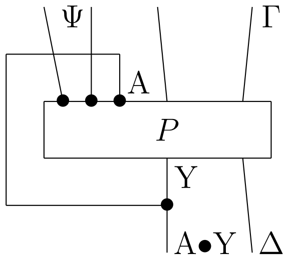


Figure 33: Receiving on output channel.

$$\frac{\Phi, A \mid \Gamma \Vdash Y, \Delta}{\Phi \mid \Gamma \Vdash A \circ Y, \Delta} \bullet_R$$

Figure 34: The $\bullet_R$ rule.

With $x, y, z \in P$, $\leq$ is:

reflexive	$x \leq x$
transitive	$x \leq y,\ y \leq z \implies x \leq z$
anti-symmetric	$x \leq y,\ y \leq x \implies x = y$

The supremum operation, *sup*, is a unique map from elements in a poset P to an element in P. For two elements $x, y \in P$, the supremum element of x and y, $sup(x, y)$, if it exists, is defined such that:

$$x \leq sup(x, y),\ y \leq sup(x, y)$$
$$\text{and } z \in P,\ x \leq z,\ y \leq z \implies sup(x, y) \leq z$$

A sup-lattice can be defined as a poset P given by a relation $\leq$ where:

$$x \leq y \iff x \vee y = y$$

And if there exists a bottom element $\perp \in P$ which, for $x \in P$, satisfies:

$$\perp \leq x,\ \forall x$$

And there exists a sup (supremum element) given by the *sup* operation for any finite set of elements from P. So, a sup-lattice is a poset with sups and a bottom element where the join of a finite set of elements is the sup of those elements.

We have only discussed sup-lattices with joins or sups of finite sets of elements. It is useful to expand this to the notion of arbitrary joins using the above definition of the *sup* operation. For a sup-lattice with carrier L, the join of an arbitrary amount I of elements $x_{i \in I} \in L$ can be defined as:

$$x_i \leq \bigvee_{i \in I} x_i$$
$$\text{and } z \in P,\ \forall x_i \leq z \implies \bigvee_{i \in I} x_i \leq z$$

As with the finite join, an arbitrary join is a map from elements in L to an element in L. We can express the type of an arbitrary join as follows:

$$\vee : \prod L \to L$$

From here, the terms join and sup will be used interchangeably depending on the context.

3.2.2 Tensor of sup-lattices

Sup is the category of sup-lattices, so the objects are sup-lattices and the morphisms are sup-preserving maps on sup-lattices. A sup-preserving map ensures the mapping of a sup for a set of elements is the sup for the mappings of those elements. This is formalized in the equation below which shows a sup-preserving map f applied to the sup of a set of I elements is equivalent to the sup of f applied to each of the I elements:

$$f(\bigvee_{i \in I} x_i) = \bigvee_{i \in I} f(x_i)$$

Sup is a monoidal category, which means it has a unit object and a tensor product with unitors, associators, and commutators. The unit object is a sup-lattice where the carrier is $\{\top, \perp\}$.

The tensor of two sup-lattices, $\mathcal{L}_1 = (L_1, \bot, \vee)$ and $\mathcal{L}_2 = (L_2, \bot, \vee)$ is defined as:

$$\mathcal{L}_1 \otimes \mathcal{L}_2 = (\{ \bigvee (x_1 \otimes x_2) \mid x_1 \in L_1, x_2 \in L_2 \}, \bot, \vee);$$
$$\bot \otimes x = \bot = x \otimes \bot$$

In Figure 35 below, the maps m' out of the product of a pair of sup-lattices $(\mathcal{L}_1, \mathcal{L}_2)$ are bi-sup-preserving. This means that the sups from the two component sup-lattices are preserved but not the sups that exist in their product. These maps can also be obtained from sup-preserving maps m out of their tensor because any bi-sup-preserving map from the product of sup-lattices as posets[3] can be uniquely presented as a map from the tensor. So, $(\mathcal{L}_1, \mathcal{L}_2)$ can be tensored to form a single sup-lattice $\mathcal{L}_1 \otimes \mathcal{L}_2$ to represent bi-sup-preserving maps m' out of their product.

$$\begin{tikzcd}
\mathcal{L}_1 \times \mathcal{L}_2 \arrow[r, "m'"] \arrow[d, "\varphi"'] & \mathcal{X} \\
\mathcal{L}_1 \otimes \mathcal{L}_2 \arrow[ur, "m"']
\end{tikzcd}$$

Figure 35: $\mathcal{L}_1 \times \mathcal{L}_2$ is treated as a poset and not a sup-lattice, so m' and φ are bi-sup-preserving but not sup-preserving. $\mathcal{L}_1 \otimes \mathcal{L}_2$ and $\mathcal{X}$ are sup-lattices, so m is a sup map and therefore sup-preserving.

The behavior of the bi-sup-preserving maps m' and φ from Figure 35 is formalized in the equations below.

$$m'(\bigvee_{i \in I} x_i, y) = \bigvee_{i \in I} m'(x_i, y) \qquad m'(x, \bigvee_{j \in J} y_j) = \bigvee_{j \in J} m'(x, y_j)$$

$$\varphi(\bigvee_{i \in I} x_i, y) = (\bigvee_{i \in I} x_i) \otimes y \qquad \varphi(x, \bigvee_{j \in J} y_j) = x \otimes (\bigvee_{j \in J} y_j)$$

$$\varphi(\bigvee_{i \in I} x_i, y) = \bigvee_{i \in I} (x_i \otimes y) \qquad \varphi(x, \bigvee_{j \in J} y_j) = \bigvee_{j \in J} (x \otimes y_j)$$

[3]A poset is a partially ordered set. Above, we showed how a sup-lattice is a poset with sups and a bottom element. Here, we are treating the sup-lattices as posets by ignoring the sups.

From these equations we can infer that the tensor of a join and a fixed element is equal to the join of each element tensored with the fixed element. This is formalized in Equation 1 below which is the reason the unique m maps from Figure 35 exist and can be used to compose sup-lattices.

$$(\bigvee_{i\in I} x_i) \otimes y = \bigvee_{i\in I}(x_i \otimes y) \qquad x \otimes (\bigvee_{j\in J} y_j) = \bigvee_{j\in J}(x \otimes y_j) \tag{1}$$

3.2.3 Enrichment

The purpose of enrichment is to represent the morphisms between two objects as an object of a monoidal category. These objects of the monoidal category are called **hom-objects**. An identity morphism is given by a map e out of the unit object of the monoidal category, and the composition of morphisms is given by a map m out of the tensor of two hom-objects.

Given a monoidal category $\mathcal{X}$, an $\mathcal{X}$-Enriched Category $\mathcal{A}$ has:

a set of objects:	$\mathcal{A}_0$
a set of hom-objects:	$\forall A_1, A_2 \in \mathcal{A}_0,$
	$\exists\ \mathcal{A}(A_1, A_2) \in \mathcal{X}$
maps for unit assignment:	$\forall A \in \mathcal{A}_0,$
	$\exists\ e_A : \mathcal{I} \to \mathcal{A}(A, A)$
maps for composition:	$\forall A_1, A_2, A_3 \in \mathcal{A}_0,$
	$\exists\ m_{A_1 A_2 A_3} : \mathcal{A}(A_1, A_2) \otimes \mathcal{A}(A_2, A_3) \to \mathcal{A}(A_1, A_3)$

The maps for unit assignment and composition must satisfy unit laws and the associative law [CS].

3.2.4 Sup-enrichment

A sup-enriched category is the specific case of enrichment where the monoidal category $\mathcal{X}$ is the category of sup-lattices. The hom-objects are sup-lattices, and the unit object and tensor of sup-lattices are as previously described. A sup-enriched category can be built from any set of objects by using the power set[4] of the morphisms between objects as the carrier for a sup-lattice.

[4]The power set is the set of all subsets.

The union of the subsets is used for the join operation, and the composition of morphisms is used for the tensor of the elements in a sup-lattice when tensoring sup-lattices.

Using $\mathcal{P}$ to denote taking a power set, we can build a sup-enriched category $\mathcal{A}$ from an arbitrary set-based category $\mathcal{C}$:

$$\begin{aligned}
&\mathcal{A}_0 = \mathcal{C}_0\\
&\mathcal{A}(X,Y) := \mathcal{P}(\mathcal{C}(X,Y))\\
e_X :\ &\mathcal{I} \rightarrow \mathcal{A}(X,X) := \mathcal{P}(\mathcal{C}(X,X));\\
&\top \mapsto \{1_X\}, \bot \mapsto \emptyset\\
m_{XYZ} :\ &\mathcal{A}(X,Y) \otimes \mathcal{A}(Y,Z) \rightarrow \mathcal{A}(X,Z) :=\\
&\mathcal{P}(\mathcal{C}(X,Y)) \otimes \mathcal{P}(\mathcal{C}(Y,Z)) \rightarrow \mathcal{P}(\mathcal{C}(X,Z));\\
&S_{XY} \otimes S_{YZ} \mapsto \{fg \mid f \in S_{XY}, g \in S_{YZ}\}
\end{aligned}$$

3.2.5 Modelling non-determinism

The non-deterministic semantics for CaMPL is built by sup-lattice enriching the semantics for deterministic concurrency. We are creating sup-lattice representations of CaMPL programs with a sup-enriched category where the objects and morphisms are taken from our existing semantics and the hom-objects are sup-lattices that represent non-deterministic programs. As discussed above, a sup-enriched categeory can be obtained by taking the power set of the morphisms between objects. So, we take the subsets of the processes in a program. When a deterministic program has more than one process option, the non-deterministic program will have a join of those options. For example, the sup-lattice for the far above example Program Y, which has options $(gfh)\ \vee\ (gf'h)$, would be created as follows:

$$\begin{tikzcd}[column sep=small]
& (gfh) \vee (gf'h) & \\
gfh \arrow[ur, dash] & & gf'h \arrow[ul, dash] \\
& \bot \arrow[ul, dash] \arrow[ur, dash] &
\end{tikzcd}$$

When a non-deterministic program is a composition of programs, we can obtain the whole program with a map out of the tensor of the component programs. For example, the sup-lattice formed from composing process g with Program X, which would give $g\ (f \vee f')$, would be created as follows:

$$\begin{tikzcd}
g \arrow[dd, dash] & & & f \vee f' \arrow[dl, dash] \arrow[dr, dash] & & & g(f \vee f') \arrow[dl, dash] \arrow[dr, dash] & \\
& \otimes & f \arrow[dr, dash] & & f' \arrow[dl, dash] \arrow[r, "m"] & gf \arrow[dr, dash] & & fg' \arrow[dl, dash] \\
\perp & & & \perp & & & \perp &
\end{tikzcd}$$

Similarly, we compose Program Y with the above program and process h, which would give $g\ (f \vee f')\ h$, as follows:

$$\begin{tikzcd}
& g(f \vee f') \arrow[dl, dash] \arrow[dr, dash] & & & h \arrow[dd, dash] & & g(f \vee f')h \arrow[dl, dash] \arrow[dr, dash] & \\
gf \arrow[dr, dash] & & fg' \arrow[dl, dash] & \otimes & \arrow[r, "m"] & gfh \arrow[dr, dash] & & gf'h \arrow[dl, dash] \\
& \perp & & & \perp & & \perp &
\end{tikzcd}$$

Using Equation 1, we can show that the Program Y we have composed with processes g, Program X, and h is equivalent to the Program Y we created above:

$$\begin{tikzcd}
& g(f \vee f')h \arrow[dl, dash] \arrow[dr, dash] & & & (gfh) \vee (gf'h) \arrow[dl, dash] \arrow[dr, dash] & \\
gfh \arrow[dr, dash] & & gf'h \arrow[dl, dash] & = \; gfh \arrow[dr, dash] & & gf'h \arrow[dl, dash] \\
& \perp & & & \perp &
\end{tikzcd}$$

Sup-lattice enriching the deterministic semantics gives the behaviour we seek in a non-deterministic semantics for CaMPL because the interaction between joins and tensors models the interaction between options and composition in non-deterministic programs.

We have represented this addition as a circuit diagram in Figure 36 and in the sequent calculus in Figure 37.

Progress of CaMPL programs will likely be unaffected by this addition. Deterministic programs in CaMPL have been shown to avoid deadlocks and livelocks [Lyb18]. Intuitively, If all deterministic programs for each of the process options can be shown to have progress, then a non-deterministic program that makes choices using those options should also have progress.

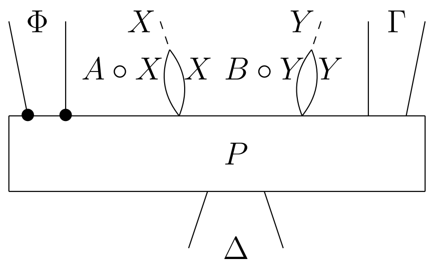


Figure 36: Racing channels X and Y.

$$\frac{\Phi, A \mid A \circ X, Y, \Gamma \Vdash \Delta \quad \Phi, B \mid X, B \circ Y, \Gamma \Vdash \Delta}{\Phi \mid X \vee Y, \Gamma \Vdash \Delta} \vee$$

Figure 37: The $\vee$ rule.

4 Conclusion

This project explored sup-enrichment as a way to model non-determinism in the categorical semantics for CaMPL. The deterministic semantics for CaMPL is a linear additive actegory, and we have shown that a linear actegory enriched in sup-lattices provides the semantics for non-determinism. The focus for modelling non-determinism was capturing the interaction between process options and composition in non-deterministic programs. Sup-lattices have an interaction between their joins and tensors which behaves in the desired way.

The plan for this model is to further investigate how non-determinism in CaMPL can be formalized. Specifically, we want to look at the products, coproducts, and session types of channels and formally show how progress is preserved after introducing non-determinism.

5 References

[CP09] J. R. B. Cockett and Craig Pastro. “The Logic of Message Passing”. In: *Science of Computer Programming* 74.8 (2009), pp. 498–533. DOI: `https://doi.org/10.48550/arXiv.math/0703713`.

[CS] J. R. B. Cockett and R. A. G. Seely. *Proof Theory of the Cut Rule.*

[Gir87] J.-Y. Girard. “Linear logic”. In: *Theoretical Computer Science* 50.1 (1987), pp. 1–102.

[Lyb18] Reginald Lybbert. *Progress for the Message Passing Logic.* Undergraduate Thesis, University of Calgary. 2018.

[Pfe18] Frank Pfenning. *Lecture Notes on Session Types.* Lecture 22, Types and Programming Languages, Carnegie Mellon University. 2018.

[Pon21] Jared Pon. *Implementation Status of CMPL.* Undergraduate Thesis Interim Report, University of Calgary. 2021.

[Yea12] Masuka Yeasin. “Linear Functors and their Fixed Points”. MA thesis. Calgary, AB: University of Calgary, Dec. 2012.