PROGRESS FOR THE MESSAGE PASSING LOGIC

REGINALD LYBBERT

ABSTRACT. One way of modelling parallel, or concurrent, systems is through message passing. Such a system consists of distinct processes running in parallel that communicate directly with one another. Here, the formal system for modelling message passing, MPL, from work previously done by Cockett and Pastro is reviewed. The system is then extended using protocols and co-protocols. In the second section of the paper, some properties of concurrent systems relating to progress are reviewed and adapted this setting, and it is shown that MPL is deadlock-free, livelock-free, but not starvation-free.

1. INTRODUCTION

Cockett and Pastro have described a logic for a programming language, MPL, that deals with the passing of messages between processes running in parallel [4]. MPL is a statically-typed concurrent programming language in which the concurrency is performed via a message passing mechanism. However, the logic described in [4] is minimalistic. It provides the basic operations necessary for a message passing system, but it is lacking functionality such as inductive and co-inductive data types in both the sequential and concurrent sides of the language. Thus, it does not provide any mechanism for recursion, or looping of any kind.

As such, MPL has been extended [6] to include inductive and co-inductive datatypes, as well as protocols and co-protocols, which are the concurrent analogues of these structures. We are now permitted to recurse using folds and unfolds, and the concurrent analogue, `drive`. In this report, I will provide the typing and term formation rules for the protocols and co-protocols, in the same manner that they are described in [4].

However, adding this extra functionality raises some questions regarding the progress properties of this system. Intuitively, we desire that an MPL program will always provide the user with appropriate output, once it has recieved sufficient user input. However, in concurrent systems, deadlock, livelock and even starvation can prevent this from happending. The main goal of this project is to analyze the progress properties of MPL, when the functionality of protocols and co-protocols is added, and determine what progess guarantees can be made.

In this report, we will first define the logical structure of MPL, and relate it to the programming syntax used to create MPL programs. This will involve the inclusion of protocols and co-protocols. Typing and term formation rules will be provided. Following this, a rewriting system that simulates the evaluation of an MPL program will be described. In the final section, we will look at the progress properties that an MPL program must satisfy.

1.1. **The Message Logic.**

In order to describe the logic of message passing, we must first look at the structure of the messages being passed. This will give us the

$$\dfrac{\text{axiom } f}{x_1 : A_1, \ldots, x_n : A_n \vdash f(x_1, \ldots, x_n) : B}\ \text{axiom} \qquad \dfrac{\Phi \vdash f : A \quad \Psi_1, w : A, \Psi_2 \vdash g : B}{\Psi_1, \Phi, \Psi_2 \vdash (w \mapsto g)f : B}\ \text{subs}$$

$$\dfrac{\Phi, x : A, y : B \vdash f : C}{\Phi, (x, y) : A * B \vdash f : C}\ *_\ell \qquad \dfrac{\Phi \vdash f : A \quad \Psi \vdash g : B}{\Phi, \Psi \vdash (f, g) : A * B}\ *_r$$

$$\dfrac{\Phi \vdash f : A}{\Phi, () : I \vdash f : A}\ I_\ell \qquad \dfrac{}{\vdash () : I}\ I_r$$

$$\dfrac{\Phi, x : A \vdash f : C \quad \Phi, y : B \vdash g : C}{\Phi, z : A + B \vdash \left\{ \begin{array}{c} \sigma_1(x) \mapsto f \\ \sigma_2(y) \mapsto g \end{array} \right\} z : C}\ \text{coprod} \qquad \dfrac{\Phi \vdash f : A}{\Phi \vdash \sigma_1(f) : A + B}\ \text{inj}_\ell \qquad \dfrac{\Phi \vdash f : B}{\Phi \vdash \sigma_2(f) : A + B}\ \text{inj}_r$$

$$\dfrac{}{\Phi, z : 0 \vdash \{\} z : A}\ \mathbf{0}$$

FIGURE 1. Term formation rules for Message Logic

sequential side of MPL. The logic of messages, as described in by Cockett [4] uses the term formation rules described in the figure below.

The axiom rule, allows us to define and use functions $f : A_1 \times \cdots \times A_n \to B$. The product rules allow us to pair data together, and the coproduct rules allow for branching. Thus we can use this logic to make computations on the sequential data in the MPL language.

The sequential side of the Message Passing Logic is based off of the typed λ-calculus. This is then augmented by adding in some basic types, such and integers and characters, along with some built-in operations, such as addition, multiplication or comparison. A complete description of this portion of the language can be found in [6].

However, there is one important aspect of this language that we will discuss here, that is the implementation of data and co-data types. It is based on of the categorical semantics described in [3, 5].

An inductive datatype for a functor $F : \mathbb{X} \to \mathbb{X}$ is an object $\mu x.F(x)$ and a map cons: $F(\mu x.F(x)) \to \mu x.F(x)$, such that for every object A and map $F(A) \to A$, a unique map, called fold, exists, for which the following diagram commutes:

$$\begin{tikzcd}
F(\mu x.F(x)) \arrow[r, "cons"] \arrow[d, "F(fold)"] & \mu x.F(x) \arrow[d, "fold"] \\
F(A) \arrow[r, "f"] & A
\end{tikzcd}$$

This gives rise to the constructor, case and fold constructs for algebraic datatypes in this language. Let F be an endofunctor on sequential types in MPL. We can then write $F(x)$ as $\sum_{i=1}^{n} F_i(x)$. Each of the F_i correspond to a constructor for this type, the sum of all the F_i is the *cons* map on the above diagram. This corresponds to the following term formation rule in sequent calculus:

$$\dfrac{}{x_1 : A_1, \ldots, x_n : A_n \vdash F_i(x_1, \ldots, a_n) : \mu x.F(x)}\ \text{cons}$$

Note that some of the A_i may be $\mu x.F(x)$, and in fact will be, for every x that appears in $F_i(x)$.

The case construct allows us to break a datatype into its component types, depending on the constructor used. This corresponds to the inclusion maps defined in the universal property of the coproduct of all the F_i. This corresponds to the following term formation rule in the sequent calculus:

$$\frac{\Phi, x_1 : F_1(\mu x.F(x)) \vdash P_1 : C \ \ldots \ \Phi, X_1 : F_n(\mu x.F(x)) \vdash P_n : C}{\Phi, z : \mu x.F(x) \vdash \text{case } z \text{ of } \begin{cases} F_1 \to P_1(x_1) \\ \vdots \\ F_n \to P_n(x_n) \end{cases} : C} \text{case}$$

Finally the fold construct corresponds to the $fold : \mu x.F(x) \to A$ map in the diagram previously mentioned. Note that such a map exists for every map $F(A) \to A$. This corresponds to the following term formation rule:

$$\frac{\Phi, x_1 : F_1(A) \vdash P_1 : A \ \ldots \ \Phi, X_1 : F_n(A) \vdash P_n : A}{\Phi, z : \mu x.F(x) \vdash \text{fold}(z, P_1, \ldots, P_n) : A} \text{fold}$$

Dually, A co-inductive datatype for a functor $G : \mathbb{X} \to \mathbb{X}$ is an object $\nu y.G(y)$ and map $dest : \nu y.G(y) \to G(\nu y.G(y))$, such that for every object A and map $A \to G(A)$, a unique map, called unfold, exists, for which the following diagram commutes:

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "unfold"] & G(A) \arrow[d, "G(unfold)"] \\ \nu y.G(y) \arrow[r, "dest"] & G(\nu y.G(y)) \end{tikzcd}$$

This also gives rise to destructor, record, and unfold constructs for codata-types. These are exactly dual to the constructor, case, and fold constucts for datatypes respectively. As none of the examples in this paper will use codata, the details are omitted from this report. The details for the codata constructs can be found in [6].

1.2. **The Message Passing Logic.**

Now that we have messages, we must discuss the kinds of channels that can pass such a message. Before the addition of protocols and co-protocols to the language, we can type our channels as follows. Let A represent a message type, and X and Y represent channel type. Then a channel must be one of the following types:

$$\top, \bot, A \circ X, A \bullet X, X \otimes Y, X \oplus Y$$

Note that since channels connect two different processes, and the behaviour of the channel differs on each end, we must have the notion of polarity for a channel. For example, a channel of type $A \circ X$ will take a value from the negative end, and give it to the process on the positive end, and continue as a channel of type X, whereas a channel of type $A \bullet X$ will take a value from the positive end, and give it to the process on the negative end. Thus, when representing processes, we must denote the polarity of every channel attached to the process. We do this by representing process types in the form $\Phi \mid \Gamma \Vdash \Delta$.

Here, the Φ represents the sequential context of the process, that is all messages (with their types), currently available to the process. Γ, and Δ represent the channels connected to the process. The channels for which their positive end is connected to the process are represented as part of Γ, to the left of $\Vdash$, while the channels of negative polarity are represented in Δ, on the right side of $\Vdash$. Sometimes, when appropriate, we may omit the sequential context from the type of the process, and simply write $P : \Gamma \Vdash \Delta$.

Now we can describe the message passing logic, as described in [4]. Here we will define rules for the formation of processes $P : \Phi \mid \Gamma \Vdash \Delta$. The programming syntax for this language is simply replaces each ; with a new line, so that we read the terms from the bottom to the top of the proof tree. For example, the term (get n on α);(put n on β);(close α);(halt β), could be of type $\emptyset \mid \alpha : A \circ \top \Vdash A \circ \top$, where n is of type A. This corresponds to the proof tree, (using typing rules)

$$\dfrac{\dfrac{\dfrac{}{A \vdash A}\ axiom \qquad \dfrac{\dfrac{}{\emptyset \mid \ \Vdash \top}\ \top_r}{\emptyset \mid \top \Vdash \top}\ \top_\ell}{A \mid \top \Vdash A \circ \top}\ \circ_r}{\emptyset \mid A \circ \top \Vdash A \circ \top}\ \circ_\ell$$

As a program (replacing the name α with ch1, and β with ch2), this process would be written:

```
get n on ch1
put n on ch2
close ch1
halt ch2
```

We can also write channel types in the programming syntax. We give the translation rules below:

$$\begin{aligned}
\texttt{Top} &:= \top \\
\texttt{Bot} &:= \bot \\
\texttt{Get(A|X)} &:= A \circ X \\
\texttt{Put(A|X)} &:= A \bullet X \\
\texttt{X (*) Y} &:= X \otimes Y \\
\texttt{X (+) Y} &:= X \oplus Y
\end{aligned}$$

1.3. **Protocols and Co-protocols.** The design of protocols and co-protocols in MPL is analogous to that of inductive and co-inductive datatypes in functional programming languages.

We use these to define our protocols and co-protocols. A protocol is an inductive datatype in the category of concurrent MPL programs. Dually, a co-protocol is a co-inductive datatype in the category of concurrent MPL programs. We call the fold associated to the protocol and the unfold associated to the co-protocol, `drive`.

Within the language, we can write a protocol or co-protocol in terms of a set of handles. In terms of syntax, if we want a protocol with handles $F_1, F_2, \ldots, F_n$, we

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

$$\frac{}{\emptyset \mid X \Vdash X}\ \text{atom id} \qquad \frac{}{\Phi \mid \Gamma \Vdash \Delta}\ \text{axiom}$$

$$\frac{\Phi \mid \Gamma, X, Y \Vdash \Delta}{\Phi \mid \Gamma, X \otimes Y \Vdash \Delta}\ \otimes_\ell \qquad \frac{\Phi \mid \Gamma_1 \Vdash \Delta_1, X \quad \Psi \mid \Gamma_2 \Vdash Y, \Delta_2}{\Phi, \Psi \mid \Gamma_1, \Gamma_2 \Vdash \Delta_1, X \otimes Y, \Delta_2}\ \otimes_r$$

$$\frac{\Phi \mid \Gamma_1, X \Vdash \Delta_1 \quad \Psi \mid Y, \Gamma_2 \Vdash \Delta_2}{\Phi, \Psi \mid \Gamma_1, X \oplus Y, \Gamma_2 \Vdash \Delta_1, \Delta_2}\ \oplus_\ell \qquad \frac{\Phi \mid \Gamma \Vdash X, Y, \Delta}{\Phi \mid \Gamma \Vdash X \oplus Y \Delta}\ \oplus_r$$

$$\frac{\Phi \mid \Gamma \Vdash \Delta}{\Phi \mid \Gamma, \top \Vdash \Delta}\ \top_\ell \qquad \frac{}{\emptyset \mid\ \Vdash \top}\ \top_r$$

$$\frac{}{\emptyset \mid \bot \Vdash}\ \bot_\ell \qquad \frac{\Phi \mid \Gamma \Vdash \Delta}{\Phi \mid \Gamma \Vdash \bot, \Delta}\ \bot_r$$

$$\frac{\Phi, A \mid \Gamma, X \Vdash \Delta}{\Phi \mid \Gamma, A \circ X \Vdash \Delta}\ \circ_\ell \qquad \frac{\Phi \vdash A \quad \Psi \mid \Gamma \Vdash X, \Delta}{\Phi, \Psi \mid \Gamma \Vdash A \circ X, \Delta}\ \circ_r$$

$$\frac{\Phi \vdash A \quad \Psi \mid \Gamma, X \Vdash \Delta}{\Phi, \Psi \mid \Gamma, A \bullet X \Vdash \Delta}\ \bullet_\ell \qquad \frac{\Phi, A \mid \Gamma \Vdash X, \Delta}{\Phi \mid \Gamma \Vdash A \bullet X, \Delta}\ \bullet_r$$

$$\frac{\Phi, A, B \mid \Gamma \Vdash \Delta}{\Phi, A * B \mid \Gamma \Vdash \Delta}\ *_\ell \qquad \frac{\Phi \mid \Gamma \Vdash \Delta}{\Phi, I \mid \Gamma \Vdash \Delta}\ I$$

$$\frac{\Phi, A \mid \Gamma \Vdash \Delta \quad \Phi, B \mid \Gamma \Vdash \Delta}{\Phi, A + B \mid \Gamma \Vdash \Delta}\ \text{coprod} \qquad \frac{}{\Phi, 0 \mid \Gamma \Vdash \Delta}\ \mathbf{0}$$

$$\frac{\Psi \vdash A \quad \Phi, A \mid \Gamma \Vdash \Delta}{\Phi, \Psi \mid \Gamma \Vdash \Delta}\ \text{subs}$$

FIGURE 2. Typing Rules for the Message Passing Logic

will use the functor $F(x) = F_1(x) + F_2(x) + \cdots + F_n(x)$, and within the program we will denote this functor as follows:

```
protocol F => P =
   F_1   :: F_1(P) => P
   F_2   :: F_2(P) => P
   ...
   F_n   :: F_n(P) => P
```

Note that the handles are each related to one of the summands of the functor.

As a concrete example, if we have a functor $\mathsf{Terminal}_\mathsf{A}(\mathsf{X}) = \mathsf{A} \circ \mathsf{X} + \mathsf{A} \bullet \mathsf{X} + \top$, and we wanted to use the protocol $\mu x.\mathsf{Terminal}_\mathsf{A}(\mathsf{x})$ in our program, we would define the protocol as follows:

```
protocol Terminal (A) => P =
```

$$\frac{P_1 :: \Phi \mid \Gamma_1 \Vdash \Delta_1, \alpha : X \quad P_2 :: \Psi \mid \beta : X, \Gamma_2 \Vdash \Delta_2}{\text{plug } \{P_1(\alpha), P_2(\beta)\} :: \Phi, \Psi \mid \Gamma_1, \Gamma_2 \Vdash \Delta_1, \Delta_2}\ \text{cut}$$

$$\frac{}{((\alpha \mid = \mid \beta) :: \emptyset \mid \alpha : X \Vdash \beta : X}\ \text{atom id}$$

$$\frac{P :: \Phi \mid \Gamma \Vdash \Delta}{(\text{close } \alpha); P :: \Phi \mid \Gamma, \alpha : \top \Vdash \Delta}\ \top_\ell$$

$$\frac{}{(\text{halt } \alpha) :: \emptyset \mid \alpha : \bot \Vdash}\ \bot_\ell$$

$$\frac{}{(\text{halt } \alpha) :: \emptyset \mid \ \Vdash \alpha : \top}\ \top_r$$

$$\frac{P :: \Phi \mid \Gamma \Vdash \Delta}{(\text{close } \alpha); P :: \Phi \mid \Gamma \Vdash \alpha : \bot, \Delta}\ \bot_r$$

$$\frac{P :: \Phi, n : A \mid \Gamma, \alpha : X \Vdash \Delta}{(\text{get } n \text{ on } \alpha); P :: \Phi \mid \Gamma, \alpha : A \circ X \Vdash \Delta}\ \circ_\ell$$

$$\frac{\Phi \vdash n : A \quad P :: \Psi \mid \Gamma, \alpha : X \Vdash \Delta}{(\text{put } n \text{ on } \alpha); P :: \Phi, \Psi \mid \Gamma, \alpha : A \bullet X \Vdash \Delta}\ \bullet_\ell$$

$$\frac{\Phi \vdash n : A \quad P :: \Psi \mid \Gamma \Vdash \alpha : X, \Delta}{(\text{put } n \text{ on } \alpha); P :: \Phi, \Psi \mid \Gamma \Vdash \alpha : A \circ X, \Delta}\ \circ_r$$

$$\frac{P :: \Phi, n : A \mid \Gamma \Vdash \alpha : X, \Delta}{(\text{get } n \text{ on } \alpha); P :: \Phi \mid \Gamma \Vdash \alpha : A \bullet X, \Delta}\ \bullet_r$$

$$\frac{P_1 :: \Phi \mid \Gamma_1 \Vdash \Delta_1, \alpha_1 : X \quad P_2 :: \Psi \mid \Gamma_2 \Vdash \alpha_2 : Y, \Delta_2}{\text{fork } \alpha \text{ as } \left\{\begin{array}{l} \alpha_1 \to P_1 \\ \alpha_2 \to P_2 \end{array}\right\} :: \Phi, \Psi \mid \Gamma_1, \Gamma_2 \Vdash \Delta_1, \alpha : X \otimes Y, \Delta_2}\ \otimes_r$$

$$\frac{P :: \Phi \mid \Gamma \Vdash \alpha_1 : X, \alpha_2 : Y, \Delta}{(\text{split } \alpha \text{ into } \alpha_1, \alpha_2); P :: \Phi \mid \Gamma \Vdash \alpha : X \oplus Y \Delta}\ \oplus_r$$

$$\frac{P :: \Phi \mid \Gamma, \alpha_1 : X, \alpha_2 : Y \Vdash \Delta}{(\text{split } \alpha \text{ into } \alpha_1, \alpha_2); P :: \Phi \mid \Gamma, \alpha : X \otimes Y \Vdash \Delta}\ \otimes_\ell$$

$$\frac{P_1 :: \Phi \mid \Gamma_1, \alpha_1 : X \Vdash \Delta_1 \quad P_2 :: \Psi \mid \alpha_2 : Y, \Gamma_2 \Vdash \Delta_2}{\text{fork } \alpha \text{ as } \left\{\begin{array}{l} \alpha_1 \to P_1 \\ \alpha_2 \to P_2 \end{array}\right\} :: \Phi, \Psi \mid \Gamma_1, \alpha : X \oplus Y, \Gamma_2 \Vdash \Delta_1, \Delta_2}\ \oplus_\ell$$

$$\frac{P :: \Phi, x : A, y : B \mid \Gamma \Vdash \Delta}{(z = (x, y)); P :: \Phi, z : A * B \mid \Gamma \Vdash \Delta}\ *_\ell$$

$$\frac{P_1 :: \Phi, x_1 : A \mid \Gamma \Vdash \Delta \quad P_2 :: \Phi, x_2 : B \mid \Gamma \Vdash \Delta}{\text{case } z \text{ of } \left\{\begin{array}{l} x_1 \to P_1 \\ x_2 \to P_2 \end{array}\right\} :: \Phi, z : A + B \mid \Gamma \Vdash \Delta}\ \text{coprod}$$

$$\frac{P :: \Phi \mid \Gamma \Vdash \Delta}{P :: \Phi, () : I \mid \Gamma \Vdash \Delta}\ I$$

$$\frac{f :: \Psi \vdash A \quad P :: \Phi, y : A \mid \Gamma \Vdash \Delta}{(y = f(x)); P :: \Phi, x : \Psi \mid \Gamma \Vdash \Delta}\ \text{subs}$$

FIGURE 3. Term Formation Rules for the Message Passing Logic

```
Getter    :: Get (A|P) => P
Putter    :: Put (A|P) => P
Close     :: Top    => P
```

Likewise, if we want to define a co-protocol, we look at the functor for which each of the handles is a factor. So, if we want handles $G_1, G_2, \ldots, G_m$, we can use the functor $G(y) = G_1(y) \times G_2(y) \times ... \times G_m(y)$. The programming syntax for such a co-protocol definition is:

$$\frac{\Phi \mid \Gamma \Vdash F_i(\mu x.F(x)), \Delta}{\Phi \mid \Gamma \Vdash \mu x.F(x), \Delta}\ \mathrm{hput}_r$$

$$\frac{\Phi \mid \Gamma, F_1(\mu x.F(x)) \Vdash \Delta \quad \ldots \quad \Phi \mid \Gamma, F_n(\mu x.F(x)) \Vdash \Delta}{\Phi \mid \Gamma, \mu x.F(x) \Vdash \Delta}\ \mathrm{hcase}_\ell$$

$$\frac{\begin{array}{|c}\hline \forall x\ \Phi \mid \Gamma, x \Vdash \Delta \\ \hline \\ \vdots \\ \hline \Phi \mid \Gamma, F_1(x) \Vdash \Delta\ \ldots\ \Phi \mid \Gamma F_n(x) \Vdash \Delta \end{array}}{\Phi \mid \Gamma, \mu x.F(x) \Vdash \Delta}\ \mathrm{drive}_\ell$$

$$\frac{\Phi \mid \Gamma, G(\nu x.G(x)) \Vdash \Delta}{\Phi \mid \Gamma, \nu x.G_i(x) \Vdash \Delta}\ \mathrm{hput}_\ell$$

$$\frac{\Phi \mid \Gamma \Vdash G_1(\nu x.G(x)), \Delta \quad \ldots \quad \Phi \mid \Gamma \Vdash G_m(\nu x.G(x)), \Delta}{\Phi \mid \Gamma \Vdash \nu x.G(x), \Delta}\ \mathrm{hcase}_r$$

$$\frac{\begin{array}{c|}\hline \forall x\ \Phi \mid \Gamma \Vdash x, \Delta \\ \hline \\ \vdots \\ \hline \Phi \mid \Gamma \Vdash G_1(x), \Delta\ \ldots\ \Phi \mid \Gamma \Vdash G_m(x), \Delta \end{array}}{\Phi \mid \Gamma \Vdash \nu x.G(x), \Delta}\ \mathrm{drive}_r$$

FIGURE 4. Typing rules for Protocols and Co-protocols

```
coprotocol CP => G =
   G_1   :: CP => G_1(CP)
   G_2   :: CP => G_2(CP)
   ....
   G_n   :: CP => G_n(CP)
```

As a concrete example, if we want a co-protocol "Console" with three handles of types $A \circ Y, A \bullet Y, \perp$ respectively, then we will write that as:

```
coprotocol CP => Console (A) =
   GetterC :: CP => Get(A|CP)
   PutterC :: CP => Put(A|CP)
   CloseC :: CP => Bot
```

Note the usage of a type variable A in both of these examples. We allow type variables of message type, as in this example, or of channel type. From a categorical viewpoint, these are parameters over which we define the functor.

Now that we have described their construction, we will discuss the usage of protocols and co-protocols in defining processes. We use the typing and term formation rules given in Figures 5 and 6 to produce them. In these rules, we are looking at a functor $F(x) = F_1(x) + F_2(X) + \cdots + F_n(x)$, and a functor $G(y) = G_1(y) \times G_2(y) \times \cdots \times G_m(y)$.

$$\frac{P :: \Phi \mid \Gamma \Vdash \alpha : F_i(\mu x.F(x)), \Delta}{(\text{hput } F_i \text{ on } \alpha); P :: \Phi \mid \Gamma \Vdash \alpha : \mu x.F(x), \Delta}\ \text{hput}_r$$

$$\frac{P_1 :: \Gamma, \alpha : F_1(\mu x.F(x)) \Vdash \Delta \quad \dots \quad P_n :: \Gamma, \alpha : F_n(\mu x.F(x)) \Vdash \Delta}{\text{hcase } \alpha \text{ of } \left\{ \begin{array}{c} F_1 \to P_1 \\ \vdots \\ F_n \to P_n \end{array} \right\} :: \Phi \mid \Gamma, \alpha : \mu x.F(x) \Vdash \Delta}\ \text{hcase}_\ell$$

$$\frac{\begin{array}{|c} \hline \forall \alpha : X\ \ \Phi \mid \Gamma, \alpha : X \Vdash \Delta \\ \hline \\ \dfrac{\dfrac{\vdots}{P_1 :: \Phi \mid \Gamma, \alpha : F_1(x) \Vdash \Delta} \quad \dots \quad \dfrac{\vdots}{P_n :: \Phi \mid \Gamma, \alpha : F_n(x) \Vdash \Delta}}{X : \Phi \mid \Gamma, F(x) \Vdash \Delta} \end{array}}{\text{drive } f(X|\Gamma \Rightarrow \Delta) \text{ with } T \text{ by } \alpha \text{ where } \left\{ \begin{array}{c} F_1 \to P_1 \\ \vdots \\ F_n \to P_n \end{array} \right\} :: T : \Phi \mid \Gamma, \alpha : \mu x.F(x) \Vdash \Delta}$$

$$\frac{P :: \Phi \mid \Gamma, \alpha : G_i(\nu x.G(x)) \Vdash \Delta}{(\text{hput } G_i \text{ on } \alpha); P :: \Phi \mid \Gamma, \alpha : \nu x.G(x) \Vdash \Delta}\ \text{hput}_\ell$$

$$\frac{P_1 :: \Gamma \Vdash \alpha : G_1(\nu x.G(x)), \Delta \quad \dots \quad P_n :: \Gamma \Vdash \alpha : G_n(\nu x.G(x)), \Delta}{\text{hcase } \alpha \text{ of } \left\{ \begin{array}{c} G_1 \to P_1 \\ \vdots \\ G_n \to P_n \end{array} \right\} :: \Phi \mid \Gamma \Vdash \alpha : \nu x.G(x), \Delta}\ \text{hcase}_r$$

$$\frac{\begin{array}{c|} \hline \forall \alpha : X\ \ \Phi \mid \Gamma \Vdash \alpha : X, \Delta \\ \hline \\ \dfrac{\dfrac{\vdots}{P_1 :: \Phi \mid \Gamma \Vdash \alpha : G_1(x), \Delta} \quad \dots \quad \dfrac{\vdots}{P_n :: \Phi \mid \Gamma \Vdash \alpha : G_n(x), \Delta}}{X : \Phi \mid \Gamma \Vdash G(x), \Delta} \end{array}}{\text{drive } f(X|\Gamma \Rightarrow \Delta) \text{ with } T \text{ by } \alpha \text{ where } \left\{ \begin{array}{c} G_1 \to P_1 \\ \vdots \\ G_n \to P_n \end{array} \right\} :: T : \Phi \mid \Gamma \Vdash \alpha : \nu x.G(x), \Delta}$$

FIGURE 5. Term Formation rules for Protocols and Co-protocols

1.4. **Executing Processes in MPL.** Now that we have defined the structure of a process, we will now describe their execution. Let $P : \Gamma \Vdash \Delta$. We will denote all channels within Γ or Δ as **external** channels of P. These are the open channels that connect P to the outside world. Now, if P is not of the form $P_1 ;_X P_2$, then it has no internal channels. However, if P is of that form, then we denote X as an internal channel, and all internal channels of P_1 and P_2 as internal channels of P.

Table 1. Strategies for external channels

	$X \in \Gamma$	$X \in \Delta$
$\top$	close channel.	close channel.
$\bot$	close channel.	close channel.
$A \circ X$	provide a value of type A and continue with a strategy on X.	receive some value of type A that determines a strategy on X.
$A \bullet X$	receive some value of type A that determines a strategy on X.	provide a value of type A and continue with a strategy on X.
$X \otimes Y$	divide into some strategy for X and Y.	divide into some strategy for X and Y.
$X \oplus Y$	divide into some strategy for X and Y.	divide into some strategy for X and Y.
$\mu x.F(x)$	provide a handle for protocol F and continue with a strategy with the same type.	receive a handle for protocol F and continue with a strategy of that type.
$\nu x.G(x)$	receive a handle for coprotocol G and continue with a strategy of that type.	provide a handle for coprotocol G and continue with a strategy with the same type.

Now, we will develop a rewriting system for P that evaluates the process. Since P is meant to communicate with the outside world, evaluation may require input or output. Thus, we first define a strategy for each external channel of P, upon which the rewriting system will depend. Each strategy is informally described in table 1.

To formally define a strategy, let S be a term formed by the rules in either Figure 6 or Figure 7. If the rules in Figure 6 are used, it is a positive strategy, whereas if the rules in Figure 7 are used, it is negative. Note that we will allow infinite strategies, just so long as they agree with the type of channel to which they are associated.

Now, we can assign strategies to external channels. Suppose α is a channel of type X is P with positive polarity, then we can attach a positive strategy of type X to α. Likewise if β is a channel of type Y in P with negative polarity, then we can attach a negative strategy of type Y to β. We will use the following notation.

Suppose that $P : \Gamma \Vdash \Delta$ is a process, and S_X is the strategy for some channel α of type $X \in \Gamma$. Then, we can denote the process P with the strategy S_X on α as:

$$S_X \ {}_{\alpha}{}^{\triangledown} \ P$$

Likewise, if we have a strategy S_Y for some channel Y for $Y \in \Delta$, then we can denote it as:

$$P \ {}^{\triangledown}{}_{\beta} \ S_Y$$

Note that we can add these onto each other. For example if $P : \alpha : X_1, \beta : X_2 \Vdash \gamma : Y_1, \delta : Y_2$, and we have strategies on each of these channels, we could denote it:

$$\frac{}{[\,] : \top}\ \top$$

$$\frac{}{\langle\rangle : \bot}\ \bot$$

$$\frac{\forall i : A, s_i : X}{?\{s_i\}_{i:A} : A \bullet X}\ \text{get}$$

$$\frac{n : A \quad s : X}{!n.s : A \circ X}\ \text{put}$$

$$\frac{s : X \quad t : Y}{s;t : X \otimes Y}\ \text{split}$$

$$\frac{s : X \quad t : Y}{s;t : X \oplus Y}\ \text{fork}$$

$$\frac{s_i : G_i(\nu x.G(x)))\forall \text{ factors } G_i(x) \text{ of } G(x)}{?_c\{s_i\}_{i\in\text{factors of } G(x)} : \nu x.G(x)}\ \text{hget}$$

$$\frac{h : \text{ summand } F_i(x) \text{ of } F(x) \quad s : F_i(\mu x.F(x))}{!_c h.s : \mu x.F(x)}\ \text{hput}$$

FIGURE 6. Formation Rules for Positive Strategies

$$\frac{}{\langle\rangle : \top}\ \top$$

$$\frac{}{[\,] : \bot}\ \bot$$

$$\frac{n : A, s : X}{!n.s : A \bullet X}\ \text{get}$$

$$\frac{\forall i : A, s_i : X}{?\{s_i\}_{i:A} : A \circ X}\ \text{put}$$

$$\frac{s : X \quad t : Y}{s;t : X \otimes Y}\ \text{split}$$

$$\frac{s : X \quad t : Y}{s;t : X \oplus Y}\ \text{fork}$$

$$\frac{h : \text{factor } G_i(x) of G(x) \quad s : G_i(\nu x.G(x))}{!_c h.s : \nu x.G(x)}\ \text{hget}$$

$$\frac{s_i : F_i(\mu x.F(x))\forall \text{ summands } F_i(x) \text{ of } F(x)}{?_c\{s_i\}_{i\in\text{ summands of } F(x)} : \mu x.F(x)}\ \text{hput}$$

FIGURE 7. Formation Rules for Negative Strategies

$$(S_1 \,{}^{\nabla}{}_{\alpha}\, (S_2 \,{}^{\nabla}{}_{\beta}\, (P \,{}_{\gamma}{}^{\nabla}\, S_3)))\, {}_{\delta}{}^{\nabla}\, S_4$$

We will now define a rewriting system that simulates the execution of an MPL program. The idea for this system is for a rewrite to correspond to the passing of a single message in the system. As such, we must look at both messages passed internally and externally. However, it is possible for a process to divide into two disjoint processes during execution, thus we will consider rewrites from lists of processes to lists of processes.

$$\boxed{\begin{array}{c}
(\text{close } \alpha); P \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; (\text{halt } \beta) \to P \\
(\text{halt } \alpha) \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; (\text{close } \beta); P \to P \\
(\alpha \mid = \mid \beta) \;{}_{\beta}{}^{\nabla}{}_{\gamma}\; P \to P[\gamma/\alpha] \\
P \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; (\beta \mid = \mid \gamma) \to P[\alpha/\gamma] \\
(\text{get } n \text{ on } \alpha); P \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; (\text{put } m \text{ on } \beta); Q \to P[n/m] \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; Q \\
(\text{put } m \text{ on } \alpha); P \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; (\text{get } n \text{ on } \beta); Q \to P \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; Q[n/m] \\
(\text{split } \alpha \text{ into } \alpha_1, \alpha_2); P \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; \text{fork } \beta \text{ as } \left\{ \begin{array}{c} \beta_1 \to P_1 \\ \beta_2 \to P_2 \end{array} \right\} \to (P \;{}_{\alpha_1}{}^{\nabla}{}_{\beta_1}\; P_1) \;{}_{\alpha_2}{}^{\nabla}{}_{\beta_2}\; P_2 \\
\text{fork } \alpha \text{ as } \left\{ \begin{array}{c} \alpha_1 \to P_1 \\ \alpha_2 \to P_2 \end{array} \right\} \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; (\text{split } \beta \text{ into } \beta_1, \beta_2); P \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; \to P_1 \;{}_{\alpha_1}{}^{\nabla}{}_{\beta_1}\; (P_2 \;{}_{\alpha_2}{}^{\nabla}{}_{\beta_2}\; P) \\
(\text{hput } G_i \text{ on } \alpha); P \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; \text{hcase } \beta \text{ of } \left\{ \begin{array}{c} G_1 \to P_1 \\ \vdots \\ G_n \to P_n \end{array} \right\} \to P \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; P_i \\
(\text{hput } G_i \text{ on } \alpha); P \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; \text{drive } \beta \text{ of } \left\{ \begin{array}{c} G_1 \to P_1 \\ \vdots \\ G_n \to P_n \end{array} \right\} \to P \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; \tilde{P}_i \\
\text{hcase } \alpha \text{ of } \left\{ \begin{array}{c} F_1 \to P_1 \\ \vdots \\ F_n \to P_n \end{array} \right\} \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; (\text{hput } F_i \text{ on } \beta); P \to P_i \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; P \\
\text{drive } \alpha \text{ of } \left\{ \begin{array}{c} F_1 \to P_1 \\ \vdots \\ F_n \to P_n \end{array} \right\} \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; (\text{hput } F_i \text{ on } \beta); P \to \tilde{P}_i \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; P
\end{array}}$$

FIGURE 8. Basic Internal Rewrites

Note that as a shortcut of notation, we will write plug $\{P_1(\alpha), P_2(\beta)\}$ as $P_1 \;{}_{\alpha}{}^{\nabla}{}_{\beta}\; P_2$. Note that this is similar notation to strategies, but differs in that the notation for a plug has two channels as subscripts, whereas strategies have but one.

Figure 8 contains the basic internal rewrites

For the rules involving drive, notice the change of P_i to $\tilde{P}_i$. Recall from the term formation rule that $P_i :: \Phi \mid \Gamma, F_i(x) \Vdash \Delta$, wherein an axiom $\Phi \mid \Gamma, X \Vdash \Delta$, may have been used in the term formation. Then $\tilde{P}_i$ is exactly the same process, but with X replaced by $\mu x.F(x)$, and the axiom replaced with the original drive term. This is what allows a form of recursion in this language.

Now, we can generate all of the internal rewrites using these basic ones. We use the following rules.

If $P;_X Q \to P';_X Q'$ is a rewrite, then so are

$$
\begin{gathered}
(\text{close } \alpha); P \;{}_{\alpha}^{\nabla}\; [] \to P \\
(\text{halt } \alpha) \;{}_{\alpha}^{\nabla}\; \langle\rangle \to \emptyset \\
((\alpha \mid = \mid \beta) \;{}_{\alpha}^{\nabla}\; s) \;{}_{\beta}^{\nabla}\; t \to \emptyset \\
(\text{get } n \text{ on } \alpha); P, \;{}_{\alpha}^{\nabla}\; !m.s \to P[m/n] \;{}_{\alpha}^{\nabla}\; s \\
(\text{put } m \text{ on } X); P \;{}_{\alpha}^{\nabla}\; ?\{s_i\} \to P \;{}_{\alpha}^{\nabla}\; s_m \\
(\text{split } \alpha \text{ into } \alpha_1, \alpha_2); P \;{}_{\alpha}^{\nabla}\; s; t \to (P \;{}_{\alpha_1}^{\nabla}\; s) \;{}_{\alpha_2}^{\nabla}\; t \\
\text{fork } \alpha \text{ as } \left\{ \begin{array}{c} \alpha_1 \to P_1 \\ \alpha_2 \to P_2 \end{array} \right\} \;{}_{\alpha}^{\nabla}\; s; t \to (P_1 \;{}_{\alpha_1}^{\nabla}\; s), (P_2 \;{}_{\alpha_2}^{\nabla}\; t) \\
(\text{hput } G_i \text{ on } \alpha); P, s(\alpha) = ?_c\{s_i\} \to P \;{}_{\alpha}^{\nabla}\; s_{G_i} \\
\text{hcase } \alpha \text{ of } \left\{ \begin{array}{c} F_1 \to P_1 \\ \vdots \\ F_n \to P_n \end{array} \right\} \;{}_{\alpha}^{\nabla}\; !_c F_i.s \to P_i \;{}_{\alpha}^{\nabla}\; s \\
\text{drive } \alpha \text{ of } \left\{ \begin{array}{c} F_1 \to P_1 \\ \vdots \\ F_n \to P_n \end{array} \right\} \;{}_{\alpha}^{\nabla}\; !_c F_i.s \to \tilde{P}_i \;{}_{\alpha}^{\nabla}\; s
\end{gathered}
$$

FIGURE 9. Basic External Rewrites

$$
\begin{gathered}
R, (P;_X Q), S \to R, (P';_X Q'), S \\
P;_X (R;_Y Q) \to P';_X (R;_Y Q') \\
R;_Y (P;_X Q) \to R;_Y (P'_X Q')
\end{gathered}
$$

The external rewrites are dependent on the strategies for the external channels. The basic external rewrites (for negative polarity) are given in Figure 9. Then, if $P \;{}_{\alpha}^{\nabla}\; S \to P' \;{}_{\alpha}^{\nabla}\; S'$ is a rewrite of negative polarity, then $S \;{}^{\nabla}_{\alpha}\; P \to S' \;{}^{\nabla}_{\alpha}\; P'$ is a rewrite of positive polarity.

Once again, we can use these basic rewrites to generate all external rewrites, using the following rules. If $P \to P'$ is a rewrite, then so are

$$
\begin{gathered}
R, P, S \to R, P', S \\
P;_Y Q \to P';_Y Q \\
Q;_Y P \to Q;_Y P'
\end{gathered}
$$

As rewrites go from lists of processes to lists of processes, we will generically refer to them as $\mathcal{P} \to \mathcal{P}'$, where $\mathcal{P}, \mathcal{P}'$ are lists of processes.

Note that there exists an internal and an external rewrite for every term formation rule except the cut rule. However, due to the structure of these rewrites, such a rewrite is unnecessary, and can be done solely by replacing the parameters in the processes being cut together, with the arguments in the plug call. For example, consider the process:

```
proc p1(|ch1 => ch2)
```

```
   get n on ch1
   plug{
       p2(n | ch1 => ch3)
       p3(| ch3 => ch2)
   }

proc p2(x|chA => chB)
   put x on chB
   close chA
   halt chB

proc p3(|chC => chD)
   get n on chC
   put n on chD
   close chC
   halt chD
```

To use the rewrites, we can rewrite this using the ${}_{\alpha}{}^{\nabla}{}_{\beta}$ notation for cuts, and use renaming on the subprocesses $p2$ and $p3$ and we will get:

$$(\mathsf{get\ n\ on\ ch1}); \big((\mathsf{put\ x\ on\ chB}); (\mathsf{close\ chA}); (\mathsf{halt\ chB})\ {}_{chB}{}^{\nabla}{}_{chC}\ (\mathsf{get\ n\ on\ chC});$$
$$(\mathsf{put\ n\ on\ chD}); (\mathsf{close\ chC}); (\mathsf{halt\ chD})\big)\,[x/n, chA/ch1, chD/ch2]$$

After performing the substitutions we get:

$$(\mathsf{get\ n\ on\ ch1}); \big((\mathsf{put\ n\ on\ chB}); (\mathsf{close\ ch1}); (\mathsf{halt\ chB})\ {}_{chB}{}^{\nabla}{}_{chC}\ (\mathsf{get\ n\ on\ chC});$$
$$(\mathsf{put\ n\ on\ ch2}); (\mathsf{close\ chC}); (\mathsf{halt\ ch2})\big)$$

To be able to perform rewrites on this process, we need to plug each external channel into an appropriate strategy. This gives us:

$$!3.[]\ {}^{\nabla}{}_{\mathsf{ch1}}\ ((\mathsf{get\ n\ on\ ch1}); ((\mathsf{put\ n\ on\ chB}); (\mathsf{close\ ch1}); (\mathsf{halt\ chB})))\ {}_{chB}{}^{\nabla}{}_{chC}\ ((\mathsf{get\ n\ on\ chC});$$
$$(\mathsf{put\ n\ on\ ch2}); (\mathsf{close\ chC}); (\mathsf{halt\ ch2}))\ {}_{\mathsf{ch2}}{}^{\nabla}\ ?\{[]\}$$

Now, notice that none of the rewrite generating rules allow us to work with anything that is after an ;, unless it is also after some ${}_{\alpha}{}^{\nabla}{}_{\beta}$ Thus, at this point there is a single rewrite we can perform, which is the external get rewrite over ch1. This gives us:

$$[]\ {}^{\nabla}{}_{\mathsf{ch1}}\ (\mathsf{put\ 3\ on\ chB}); (\mathsf{close\ ch1}); (\mathsf{halt\ chB}))\ {}_{chB}{}^{\nabla}{}_{chC}\ ((\mathsf{get\ n\ on\ chC});$$
$$(\mathsf{put\ n\ on\ ch2}); (\mathsf{close\ chC}); (\mathsf{halt\ ch2}))\ {}_{\mathsf{ch2}}{}^{\nabla}\ ?\{[]\}$$

After this, the system is already set up such that the plug has happened immediately, as the ${}_{chB}{}^{\nabla}{}_{chC}$ is brought to the top level. Thus, we next perform an internal put rewrite, over that cut. That give us:

$$[]\ {}^{\nabla}{}_{\mathsf{ch1}}\ (\mathsf{close\ ch1}); (\mathsf{halt\ chB}))\ {}_{chB}{}^{\nabla}{}_{chC}\ ((\mathsf{put\ 3\ on\ ch2}); (\mathsf{close\ chC}); (\mathsf{halt\ ch2}))\ {}_{\mathsf{ch2}}{}^{\nabla}\ ?\{[]\}$$

following by an external put on ch2 and an external close on ch1, which give us respectively:

$$[] \ {}^{\nabla}\mathsf{ch1} \ \ (\mathsf{close\ ch1}); (\mathsf{halt\ chB})) \ {}_{chB}{}^{\nabla}{}_{chC} \ \ ((\mathsf{close\ chC}); (\mathsf{halt\ ch2})) \ {}_{\mathsf{ch2}}{}^{\nabla} \ []$$

$$(\mathsf{halt\ chB}) \ {}_{chB}{}^{\nabla}{}_{chC} \ (\mathsf{close\ chC}); (\mathsf{halt\ ch2}) \ {}_{\mathsf{ch2}}{}^{\nabla} \ []$$

Then, we have an internal halt rewrite, leaving us with:

$$(\mathsf{halt\ ch2}) \ {}_{\mathsf{ch2}}{}^{\nabla} \ []$$

Now we have only the external halt rewrite left, which leaves nothing left, and thus ends the execution of the program.

The point of this is to provide an example of how the rewrite rules we have given model execution, and even take into account the plug command, despite not having its own rule.

2. Progress Properties

There are other formal systems in the literature that model concurrent systems based on message passing. For example, there has been work done using session types, an enhanced version of the π-calculus, to study message passing [2, 13]. In some of this work [2, 9], qualities, such as deadlock-freedom, that relate to the progress of a system have been studied. In this section of the report, we will consider some properties we would like a system to satisfy, with respect to making progess, and then prove whether they are necessarily satisfied by a well-formed program in MPL. Most interestingly, we will prove that MPL satisfies live-lock freedom, which as far as we can tell, has not been studied before within formal message passing systems.

In order to clearly define the different notions of progress that a process may satisfy, we will treat the evaluation of this program as a series of rewrites. First we consider the internal rewrites. These embody all the messages that are passed along internal channels within the process. This is adapted from the cut-elimination rewrites used in [4].

First, we must have a strategy for each external channel, these give the rewrites due to interacting with the external world. Furthermore, communication along any outside channel can occur without the process being required to wait. This simulates the progress requirement that the user provide sufficient input for progress to be made.

2.1. Deadlocks.

Now that we have established the setting, we will adapt definitions of conditions commonly used to describe progress, as in [7, 8, 12]. The first definition we will give is a weaker notion of progress, which we will prove MPL satisfies later in this report.

Definition 2.1. *Let $P : \Gamma \Rightarrow \Delta$ be a process. Then P is in **deadlock** if there do not exist any possible evaluation steps $P \to P'$. P is **deadlock-free** if for every possible strategy on the external channels there does not exist a (possibly empty) sequence of evaluations steps such that P evolves into some P', where P' is in deadlock.*

However, this definition is a very weak property, and we will show that all well-formed MPL programs are deadlock-free.

The following definition will be useful in some of the following proofs:

Definition 2.2. *Let $P : \Gamma \Vdash \Delta$ be a process. Then P is an atomic process if the last term formation rule used was not 'cut'.*

Notice that if P is an atomic process, than it is not in the appropriate form for any internal rewrite to occur. Also, if strategies are connected to the appropriate external channels, it will have exactly one possible external rewrite, depending on the last term formation rule used.

Proposition 2.1. *Let $P : \Gamma \Vdash \Delta$ be a process in MPL, with strategies attached to every external channel. Then there exists a connectivity tree T for which each node etiher represents an atomic process within P or a strategy on an external channel, and each edge represents a channel connecting the proof trees of two processes, or an external edge associated to a process and strategy. Also, every strategy is associated to a leaf node.*

Proof: We will prove this by induction on the number of cut rules used at the end of the term formation for P. First, suppose that P is atomic. That is, there were no cut rules used at the end of the term formation rule for P. Then its connectivity tree has a single node representing it, and an edge and node representing each external channel along with its strategy. Since every edge in this graph is connected to the node representing P, we see that it is a star graph, which is a form of tree. Also note that all nodes in a star graph are leaf nodes, with the exception of the center. Thus all nodes corresponding to strategies are leaf nodes. Thus, the proposition holds in this case.

Now let $k \geq 1$, and suppose that we know there is a connectivity tree for every process which ends at most $k-1$ cut rules in the term formation. Let P be a process for which k cut rules were used at the end of its formation. We see that P is of the form $s\ {}_{\alpha}\!\nabla_{\beta}\ t$. Since P had k cut rules in its formation, we see that both s and t must have had less, thus by our inductive hypothesis, both s and t have connectivity trees. Now, note that α is an external channel for s, and β is an external channel for t. Thus, in the tree for s, there is a leaf node and edge corresponding to α and a strategy on α. Likewise, there is a leaf node and edge on the tree for t corresponding to β and its strategy. Thus, to construct the connectivity tree for P, we delete nodes and edges corresponding to α and β and their strategies, and add an edge between the nodes these had been connected to. Note that removing a leaf node from a tree produces a new tree, and adding an edge between disjoint trees also produces a tree. Thus, we have produced a connectivity tree for P.

Thus, we have seen by induction that a connectivity tree exists for any process P.

□

Proposition 2.2. *Let $\mathcal{P} \to \mathcal{P}'$ be a rewrite. Then, if each process in $\mathcal{P}$ has a connectivity tree, so does each process in $\mathcal{P}'$.*

Proof: We will go through each rewrite individually and determine the change in the connectivity tree. First, we look at the internal rewrites. Consider the rewrites corresponding to close and halt, namely:

$$(\text{close } \alpha).P\ {}_{\alpha}\!\nabla_{\beta}\ (\text{halt } \beta) \to P$$

$$(\text{halt } \alpha)\ {}_{\alpha}\!\nabla_{\beta}\ (\text{close } \beta).P \to P$$

From the term formation rule for (halt α) we see that the only channel attached to this process is α. Thus, it is a leaf node in the connectivity tree. Then, this rewrite removes the edge corresponding to this this channel, and the leaf node. Since removing a leaf from a tree gives a tree, we see that if $\mathcal{P} \to \mathcal{P}'$ is a rewrite of this form, then if each process in $\mathcal{P}$ has a connectivity tree, so does $\mathcal{P}'$.

Consider the rewrites corresponding the id, namely:

$$(\alpha \mid = \mid \beta)\ {}_{\beta}{\nabla}_{\gamma}\ P \to P[\gamma/\alpha]$$
$$P\ {}_{\alpha}{\nabla}_{\beta}\ (\beta \mid = \mid \gamma) \to P[\alpha/\gamma]$$

In this case, consider the process $(\alpha| = \mid \beta)$. From the term formation rules, we see that this process has a single positive channel, α, and a single negative channel β. This rewrite takes the positive end of β and connects it to the negative end of α. So, in terms of the connectivity tree, this rewrite removes a node of degree 2, and connects its two neighbours. Note that this operation reduces both the node and edge count by the same amount, namely 1, thus the new connectivity graph is still a tree, as required.

Consider the rewrites corresponding to get, put, hput, hcase, and drive namely:

$$(\text{get } n \text{ on } \alpha); P\ {}_{\alpha}{\nabla}_{\beta}\ (\text{put } m \text{ on } \beta).Q \to P[n/m]\ {}_{\alpha}{\nabla}_{\beta}\ Q$$
$$(\text{put } m \text{ on } \alpha); P\ {}_{\alpha}{\nabla}_{\beta}\ (\text{get } n \text{ on } \beta).Q \to P\ {}_{\alpha}{\nabla}_{\beta}\ Q[n/m]$$
$$(\text{hput } G_i \text{ on } \alpha); P\ {}_{\alpha}{\nabla}_{\beta}\ \text{hcase } \beta \text{ of } \left\{ \begin{array}{c} G_1 \to P_1 \\ \vdots \\ G_n \to P_n \end{array} \right\} \to P\ {}_{\alpha}{\nabla}_{\beta}\ P_i$$
$$(\text{hput } G_i \text{ on } \alpha); P\ {}_{\alpha}{\nabla}_{\beta}\ \text{drive } \beta \text{ of } \left\{ \begin{array}{c} G_1 \to P_1 \\ \vdots \\ G_n \to P_n \end{array} \right\} \to P\ {}_{\alpha}{\nabla}_{\beta}\ \tilde{P_i}$$
$$\text{hcase } \alpha \text{ of } \left\{ \begin{array}{c} F_1 \to P_1 \\ \vdots \\ F_n \to P_n \end{array} \right\}\ {}_{\alpha}{\nabla}_{\beta}\ (\text{hput } F_i \text{ on } \beta).P \to P_i\ {}_{\alpha}{\nabla}_{\beta}\ P$$
$$\text{drive } \alpha \text{ of } \left\{ \begin{array}{c} F_1 \to P_1 \\ \vdots \\ F_n \to P_n \end{array} \right\}\ {}_{\alpha}{\nabla}_{\beta}\ (\text{hput } F_i \text{ on } \beta).P \to \tilde{P_i}\ {}_{\alpha}{\nabla}_{\beta}\ P$$

Note that these rewrites do not alter the connectivity structure, as in each case, the only change occurs across the cut ${}_{\alpha}{\nabla}_{\beta}$, without removing, changing, or adding to the structure of this channel. Thus, the new process has the same connectivity structure.

Consider the rewrites corresponding to split and fork, namely:

$$(\text{split } \alpha \text{ into } \alpha_1, \alpha_2); P\ {}_{\alpha}{\nabla}_{\beta}\ \text{fork } \beta \text{ as } \left\{ \begin{array}{c} \beta_1 \to P_1 \\ \beta_2 \to P_2 \end{array} \right\} \to (P\ {}_{\alpha_1}{\nabla}_{\beta_1}\ P_1)\ {}_{\alpha_2}{\nabla}_{\beta_2}\ P_2$$

$$\text{fork } \alpha \text{ as } \left\{ \begin{array}{c} \alpha_1 \to P_1 \\ \alpha_2 \to P_2 \end{array} \right\}\ {}_{\alpha}{\nabla}_{\beta}\ (\text{split } \beta \text{ into } \beta_1, \beta_2).P\ {}_{\alpha}{\nabla}_{\beta}\ \to P_1\ {}_{\alpha_1}{\nabla}_{\beta_1}\ (P_2\ {}_{\alpha_2}{\nabla}_{\beta_2}\ P)$$

From the term formation rule for fork, we see that each channel, other than the one being split, connected to P is associated to exactly one of P_1 or P_2. So, this rewrite alters the connectivity graph by spliting P into two nodes, where each edge is connected to exactly one of the new nodes, with the singular exception on the channel being split, which is connected to both. Thus, this operation adds exactly one node, and exactly one edge to the connectivity tree. Thus, if the original connectivity graph was a tree, the new connectivity graph is still a tree, as required.

So we have seen that internal rewrites can only alter the connectivity structure of a process in ways that preserve the property of being a tree.

Now consider the external rewrites. First, we look at close:

$$(\text{close } \alpha); P \; {}_{\alpha}{}^{\nabla} \; [] \to P$$

In this case, external channel is closed. Thus, the connectivity graph will be reduced by one leaf node, corresponding to the strategy on the closing channel, and one edge, corresponding to that channel. Thus, since a tree is still a tree after a leaf is removed, we see that the tree structure is maintained.

Next consider the halt and id rules:

$$(\text{halt } \alpha) \; {}_{\alpha}{}^{\nabla} \; \langle\rangle \to \emptyset$$
$$((\alpha \mid = \mid \beta) \; {}_{\alpha}{}^{\nabla} \; s) \; {}_{\beta}{}^{\nabla} \; t \to \emptyset$$

Note that in both of these cases, every channel connected to the process is an external channel. Also note that the rewrite causes the whole structure to vanish. So, if $\mathcal{P} \to \mathcal{P}'$ is a rewrite generated from one of these ,we see that $\mathcal{P}'$ is list of all but one of the processes in $\mathcal{P}$. Thus, if all the processes in $\mathcal{P}$ have connectivity trees, we see that the same is true for $\mathcal{P}'$.

Next consider get, put ,hput, hcase, and drive:

$$(\text{get } n \text{ on } \alpha); P, \; {}_{\alpha}{}^{\nabla} \; !m.s \to P[m/n] \; {}_{\alpha}{}^{\nabla} \; s$$
$$(\text{put } m \text{ on } X); P \; {}_{\alpha}{}^{\nabla} \; ?\{s_i\} \to P \; {}_{\alpha}{}^{\nabla} \; s_m$$
$$(\text{hput } G_i \text{ on } \alpha); P, s(\alpha) = ?_c\{s_i\} \to P \; {}_{\alpha}{}^{\nabla} \; s_{G_i}$$
$$\text{hcase } \alpha \text{ of } \left\{ \begin{array}{c} F_1 \to P_1 \\ \vdots \\ F_n \to P_n \end{array} \right\} \; {}_{\alpha}{}^{\nabla} \; !_c F_i.s \to P_i \; {}_{\alpha}{}^{\nabla} \; s$$
$$\text{drive } \alpha \text{ of } \left\{ \begin{array}{c} F_1 \to P_1 \\ \vdots \\ F_n \to P_n \end{array} \right\} \; {}_{\alpha}{}^{\nabla} \; !_c F_i.s \to \tilde{P}_i \; {}_{\alpha}{}^{\nabla} \; s$$

These rewrites do not alter the connectivity structure, so if the process had a connectivity tree before the rewrite, it has the same tree after the rewrite.

Next consider the external split rewrite:

$$(\text{split } \alpha \text{ into } \alpha_1, \alpha_2); P \; {}_{\alpha}{}^{\nabla} \; s; t \to (P \; {}_{\alpha_1}{}^{\nabla} \; s) \; {}_{\alpha_2}{}^{\nabla} \; t$$

This rewrite splits the leaf node corresponding to $s; t$ into two leaf nodes corresponding to s and t. Note that adding a leaf to a tree still produces a tree. Now consider the external fork rewrite:

$$\text{fork } \alpha \text{ as } \left\{ \begin{array}{c} \alpha_1 \to P_1 \\ \alpha_2 \to P_2 \end{array} \right\} \; {}_{\alpha}{}^{\nabla} \; s; t \to (P_1 \; {}_{\alpha_1}{}^{\nabla} \; s), (P_2 \; {}_{\alpha_2}{}^{\nabla} \; t)$$

This is the only rewrite that splits one process into two, thus requiring us to work with processes. So suppose that the original process had a connectivity tree. Then there is a leaf node and edge corresponding us to α. Also, from the term formation rule for fork, we see that every other channel connected to the atomic process containing the fork, corresponds to exactly one of P_1 or P_2. So, we first perform the same transformation we did for the internal fork rewrite. However, this causes the node corresponding to the strategy $s;t$ to have degree 2. We require this to be a leaf node, thus we split this node into two, each one taking one of the edges on it. This divides the tree into two trees. However, none of these operations could add a cycle to the graph, thus both sides are still trees, as required.

Therefore, if we have a rewrite $\mathcal{P} \to \mathcal{P}'$ where $\mathcal{P}$ had a connectivity tree, then $\mathcal{P}'$ also has a connectivity tree.

□

Theorem 2.3. *MPL is deadlock-free*

Proof:

Let P be a process at some point in its evaluation. Then, by the above propositions, P has a (list of) connectivity tree(s). Now consider some node in this tree. Since each node corresponds to an atomic process, we see that there is some term formation rule, other than cut, that determines the first part of the term. As each rewrite deals with the first part of any term, that is what we want to look at.

We see that each piece of a term corresponds to exactly one channel. For example, the line `get n on ch1` corresponds to channel ch1, or the line
`fork ch into ch1 -> P1, ch2 -> P2` corresponds to channel ch. Now, if this channel is external, we see that there is exactly one possible external rewrite possible on this atomic processes, whereas if this channel is internal, then depending on the state of the process on the other side of the channel, there are at most one possible internal rewrites that can occur on that channel.

Now, suppose that processes P_1 and P_2 are connected by some channel, such that $P_1 \ {}_{\alpha}\!\overset{\nabla}{}_{\beta}\ P_2$. Then, if the first term in P_1 is on α, and the first term on P_2 is on β, they must be compatible. This is because, for each possible type of α and β there is exactly one possible term that could denote the next action on that channel. Also note, that each of the internal rewrites are designed such that if $P_1 \ {}_{\alpha}\!\overset{\nabla}{}_{\beta}\ P_2 \to Q$ is an internal rewrite, then it is of the form that α has the next action on P_1, and β has the next action P_2.

Therefore, if P_1 will next act on α and P_2 will next act on β, we see that there is an internal rewrite determining that action, since α and β must have the same type, by the cut rule.

Now, consider the connectivity forest of P. We know that there must be more nodes than edges, by the properties of trees. Also, since each node corresponds to either an atomic process, or a strategy, we see that can assign an edge to each function, by assigning to an atomic process the next channel it will act on, and to a strategy, the channel it corresponds to. Now, by the pigeonhole principle, we see that there must be some edge assigned to more than one node. Furthermore, since each node can only be assigned to one of the edges connected to it, we see that this edge must be assigned to the nodes on both ends.

Now, suppose that one of the nodes is a strategy. Then, the other node is an atomic process. However, if an atomic process next is able to communicate on an external channel, there is some external rewrite that can be performed. Namely, the rewrite corresponding that that external channel, and its type.

On the other hand, if both nodes are atomic processes, then we see that both are ready to act on the same channel. Then, as we have shown, there is some internal rewrite, determined by the type of the channel, that can be performed on these processes.

Therefore, P cannot be in deadlock. However, since we simply assumed that P was an arbitrary well-formed MPL program, at an arbitrary point in its execution, we see that MPL is deadlock-free.

□

2.2. **Starvation-free.** The idea of being starvation-free is a very strong notion of progess, that may encapsulate many ideas we may want to hold true, but will be shown to be too strong for MPL. We will provide two similar definitions for starvation-free, and show that both that they are distinct, and that there are MPL programs that do not satisfy these properties.

Definition 2.3. *Let $P : \Gamma \Rightarrow \Delta$ be a process. Then P is* ***weakly starvation-free*** *if for every channel X in Γ or Δ and strategy on the other channels in Γ and Δ, P has a path of rewrites that leads to communication over the external channel X, and after this rewrite, P evolves into some P' which is also weakly starvation-free.*

Definition 2.4. *Let $P : \Gamma \Rightarrow \Delta$ be a process. Then P is* ***strongly starvation-free*** *if for every channel X in Γ or Δ and strategy on the other channels in Γ and Δ, Every path of rewrites from P eventually leads to communication over the external channel X, and after this chain of rewrites, P evolves into some P' which is also strongly starvation-free.*

From these definitions, we see that if a program is strongly starvation-free, then it is also weakly starvation-free. We will show that MPL satifies neither of these properties, and that the two properties are distinct.

Proposition 2.4. *MPL is not weakly starvation-free*

Consider the following program:

```
protocol InfGet (A) => P =
      GetInt   :: Put (A|P) => P

data Bool -> C =
    False :: -> C
    True  :: -> C


proc p1 :: |InfGet (Int) => InfGet (Int), InfGet (Int) =
      | ch1 => ch2,ch3 -> do
          drive f(| => ch2, ch3) by ch1 where
            GetInt -> do
               get n on ch1
               case n >= 0 of
                 True -> do
                    hput GetInt on ch2
```

```
                   put n on ch2
                   f(| => ch2,ch3)

                False -> do
                   hput GetInt on ch3
                   put n on ch3
                   f(| => ch2,ch3)


run ch1 => ch2, ch3 -> do
       p1 (|ch1 => ch2,ch3)
```

Notice that if every input received over ch1 is a non-negative integer, no communication will ever occur over ch3. Thus, this process is neither weakly nor strongly starvation-free. Since this is a well-typed process in MPL, we see that MPL is not starvation-free. However, from Propositions 1 and 2, we see that if we have a program in MPL, such that every atomic process is starvation-free, then the whole program is as well.

Proposition 2.5. *There exist programs that are weakly starvation-free, but not strongly starvation-free in MPL*

```
coprotocol CP => InfGet (A) =
      GetInt   :: CP => Get (A|P)

data Bool -> C =
    False :: -> C
    True  :: -> C

proc p1 =
   | => ch4 -> do
       split ch4 into ch5, ch6
       close ch5
       halt ch6

proc p2 :: |InfGet (Int) => InfGet (Int), InfGet (Int) =
      | ch4 => ch2,ch3 -> do
          fork ch4 as
               ch5 -> drive f(|ch5 => ) by ch2 where
                    GetInt -> do
                       get n on ch2
                       f(|ch5 =>)

               ch6 -> drive f(|ch6 => ) by ch3 where
                    GetInt -> do
                       get n on ch3
                       f(|ch6 => )



run => ch2, ch3 -> do
```

```
    p1 (|=> ch4)
    p2 (|ch4 => ch2,ch3)
```

Note that this program has exactly two external channels. This program sets up a fork, so that we have two separate atomic processes dealing with each of these channels. Both of these channels can recieve an integer from the external channel at any time. Since at any time in execution there are two possible rewrites, one on each external channel, we see that this process is not strongly starvation-free, since we for either external channel, we can always choose to perform a rewrite over it. Also, this process is weakly starvation-free, since we could choose to perform an infinite sequence of rewrites over ch2, and thus never communicate over ch3.

□

Now, to know which definition of starvation-free is more relevent, we must look at the system on which MPL is being run. If MPL is being run on a truly concurrent system, where each atomic process has its own dedicated core, then we would want to consider if program is weakly starvation-free. This is because, in a concurrent system, all possible rewrites will be performed simultaneously, therefore, if there is a rewrite, or path of rewrites, that involves communication of some channel, that channel will be communicated over.

On the other hand, if MPL is being run on a system that only simulates concurrency, but still runs on the same core, then it depends on the way that operations are scheduled. If the system is given a fair scheduler, that is, if any possible rewrite only has a bounded amount of time to wait before it is performed, then can once again only worry about being weakly starvation-free. However, if the scheduler is not fair, we may have starvation in weakly starvation-free programs, but never in strongly starvation-free programs.

2.3. **Other Progress Conditions.**

The ideas of dead-lock and starvation have been directly adapted from literature discussing progress guarantees in concurrent or distributed systems [8, 7]. As such, one might wonder if other notions, such as wait-free or lock-free might also be adapted to this context and studied.

The primary difference between this system and those studied in the cited literature is the nature of the interaction between different processes and threads. In our system, the processes communicate directly with each other, passing messages back and forth. On the other hand, the notions of wait-free and lock-free are generally applied to systems in which different threads only communicate by working with shared resources. For such a system, there is a meaningful notion of a *blocking* property. Namely, ”Such an implementation is called blocking, because an unexpected delay by one thread can prevent others from making progress.” [7, p. 59].

We could try and adapt this notion to our system, and say that an MPL program would be *blocking*, if an unexpected delay in one of it's processes, would cause other processes to wait indefinitely. However, in this system, this must always be the case, since for any message to be passed across a channel, one of the processes connected to the channel is required to wait for the other to pass the message, thus if the processes required to pass the message is unexpectedly delayed, the other channel will be left waiting on the channel forever. Therefore, MPL can not satisfy any non-blocking property, as an unexpected delay in one process can not necessarily be ignored by the other processes.

Since wait-free can be seen as non-blocking version of starvation-free, and lock-free can be see as a non-blocking version of deadlock-free, we see that MPL cannot satisfy this progress properties.

2.4. **Livelock.**

We have seen that deadlock-freeness is a very weak notion of progress satisfied by MPL, and we have considered the notion of starvation, and seen that it is possible in MPL. Finally, we will consider the stronger notion of progress than deadlock-free, and show that MPL does satisfy this as well.

Definition 2.5. *Let $P : \Gamma \Rightarrow \Delta$ be a process. Then P is in **livelock** if there exists some a set of strategies on the external channels, such that P has an infinite sequence of internal evaluation steps. P is **livelock-free** it for every possible strategy on the external channels, there does not exists a (possibly empty) sequence of evaluation steps such that P evolves into some P', where P' is in **livelock**.*

Now, to prove that MPL is livelock free, we will first show how we can view channels as being parity games, and processes as mediating strategies between the games.

First, recall that channel types are generated as follows. We have $\top$ and $\bot$ as our base types. Then, if X and Y are channel types, and A is a message type, then $A \circ X, A \bullet X, X \oplus Y$, and $X \otimes Y$ are all channel types. Also, if F and G are endofunctors in the category of channel types with an initial algebra, or final co-algebra, respectively, then $\mu x.F(x)$ and $\nu x.G(x)$ are also channel types.

We can represent a channel type with a parity game. The translation is based on the representation of μ-terms as parity games in [11].

A parity game is defined as a tuple $G(S, h, \kappa, \epsilon)$, where S is a finite directed graph of positions and moves, h is an assignment of each position to either natural number less than or equal to some n, or ω, where if ω is assigned to the position, it has no outgoing moves. $\kappa : \{1, 2, \ldots, n\} \to \{\mu, \nu\}$ assigns either μ or ν to each natural number less than or equal to n, and ϵ assigns to each position a player, either σ or π. For example, Figure 10 is the parity game corresponding to the type $\nu y.(\bot \times \mu x.(\top + (x \times y)))$.

Notice how this is just a directed graph, in which each vertex is labeled with either σ or π, and placed within a level, where each level is labeled either μ or ν. Now, for this to be a game, we have to describe what each player does, and what the winning condition is. The game is played by starting at a given vertex. In this case, our starting vertex is the only vertex in level 3, as denoted by the arrow with no source. Then, at a given vertex, the player who is denoted by the vertex chooses some outgoing edge from that vertex to follow. As such, we develop a path through the graph. If the path ends at a leaf, that is, a vertex with no outgoing edges, the player designated by that vertex loses, as they cannot make a move. If the path is an infinite cycle, and the highest level reached infinitely often is labeled with ν, then σ wins, on the other hand, if the highest level reached infinitely often is labeled with μ, then π wins.

Thus, in 10, there is no winning strategy for π, however, there is a winning strategy for σ. To win, σ must never choose to go to the leaf labeled σ is level 1, and may only choose to return to level 2 a finite number of times consecutively. That is, σ must go to level 3 infinitely may times to win.

We can build parity games for channel types inductively as follows. The parity game corresponding to $\top$ is:

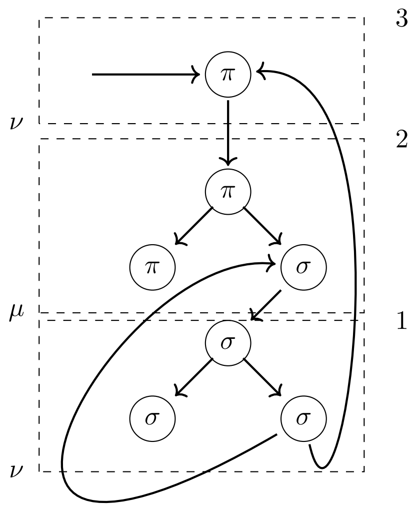
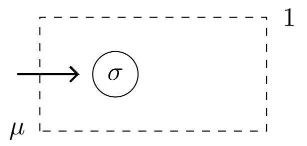
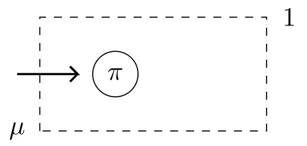


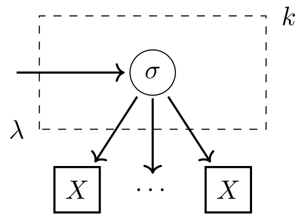
FIGURE 10. Example Parity Game



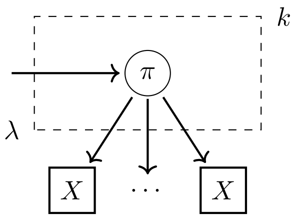
Likewise for $\perp$:



Let k be the highest level in X, and let $\lambda = \epsilon(k)$. The following is the diagram for $A \circ X$, where the arrows are indexed by elements of A and each arrow points to a separate copy of the game of X.



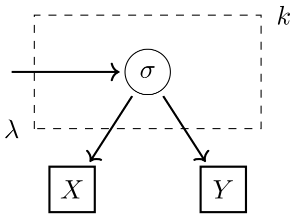
Similarly, we the diagram for $A \bullet X$ is:



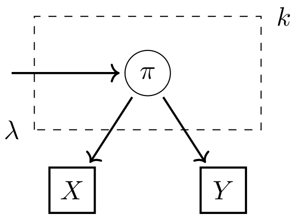
We will come back to $X \oplus Y$ and $X \otimes Y$ after discussing the inductive and co-inductive channel types.

For the inductive and co-inductive types, we allow sums and products of types within the functors defining them, so we must show how that affects the game, we also will need to show what to do when we reach the x term in $F(x)$.

Let k be the highest level in either X or Y, and let $\lambda = \epsilon(k)$. Then, for $X + Y$:

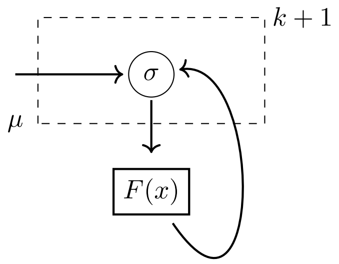


Likewise for $X \times Y$:

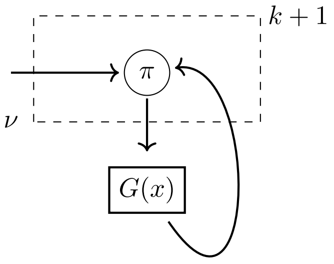


The way cycles are introduced in parity games are through inductive and co-inductive types:

Letting k be the highest level in $F(x)$, then for $\mu x.F(x)$, we have:

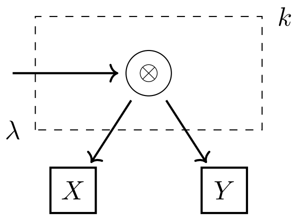


Likewise, for $\nu x.G(x)$:



The arrows coming out of $F(x)$ and $G(x)$ correspond to every instance of the parameter x being used. Rather than pointing to some parity game corresponding to x, it points back at itself. This is how we encode the recursive structure of $\mu x.F(x)$ and $\nu x.G(x)$.

The way that the parity games corresponding to $X \otimes Y$ and $X \oplus Y$ are defined is based on the the application of $\otimes$ and $\oplus$ to games in [1]. In each of these cases, the new game has both players playing on both X and Y. Thus, the winning condition must be defined differently, as an instance of a game may not be a single path through the game graph, but a tree. Note that the graph for this parity game $X \otimes Y$ looks like:



If the game reaches the $\otimes$ position, then neither player chooses a move, but instead, the game continues along both edges, and the players must make moves on both sides. As such, and instance of the game is no longer a path through the graph of the parity game, but a tree. In the case of $\otimes$, the winning condition is defined as follows. We say that σ wins and instance of the game $X \otimes Y$, of both branches of the tree lead to a win for σ That is, σ wins $X \otimes Y$ if and only if σ wins both X and Y. Equivalently, σ loses $X \otimes Y$, and thus π wins, if and only if σ loses either X or Y, that is, if π wins at least one of X and Y.

In the case of $X \oplus Y$, the game tree is constructed the same way, however the winning condition is reversed. We say that σ wins on $X \oplus Y$ if they win on either X or Y, whereas now π must win on both to win the composite game.

Notice that if we have a channel type that includes some $\otimes$ or $\oplus$, the game corresponding to this type will not be a single path through the game tree, but will divide into pieces. Thus, to define a winning strategy we must look at this as many possible paths, and our winning condition is defined on the whole. Then, a strategy for σ is a winning strategy, if for every possible choice of branches off of a $\otimes$ node, there is a choice of branches off of the $\oplus$ nodes such that the strategy, reduced these branches is a winning strategy.

Dually, a strategy for π is a winning strategy, if there is a choice of branches off of each $\otimes$ node, such that for every choice of branches off of the $\oplus$ nodes, π has a winning strategy. Note that these conditions are the negation of each other. So even for these conbined games, exactly one of σ or π can have a winning strategy.

Now, we can think of processes as mediating strategies on its external channels. In [10], this is described in detail for processes with two channels. Here, we will generalize these ideas and consider processes with an arbitrary number of channels.

We define a mediating strategy between n games to be a map from the set of games to $\{\sigma, \pi\}$, determining which player the strategy is for, along with a strategy for σ for the game with the all the positions and moves in all the games, but with the player names switched in any game where the mediating strategy should play as π. We call a mediating strategy a winning mediating strategy, if it is a winning strategy for any of the games it is between.

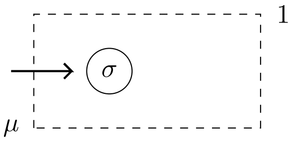
Next, we will show that if P is a well-formed MPL program, then it is a winning mediating strategy on the games corresponding to its channels, where it plays as σ on the channels with positive polarity, and π on those with negative polarity.

Theorem 2.6. *Let P be an MPL program, then if we consider P as a mediating strategy between the parity games its channels are interpreted as, then it is a winning strategy on at least one of its channels.*

Proof:
We will prove this via structural induction. First, consider the rule $\top_r$:

$$\frac{}{(\text{halt }\alpha) :: \emptyset \mid \ \Vdash \alpha : \top}\ \top_r$$

This is a base case, is there are no assumptions in this rule, only a conclusion. Now, note that our process has but channel, of type $\top$. From our translation, we see that $\top$ corresponds to the parity game:



Since this channel has positive polarity, we see that the process plays as σ. However, there is but a single move for σ, at which point the game ends as a win for σ. Thus, the only strategy on this game is a winning strategy for σ.

Similarly the rule $\bot_\ell$ has a single channel, of type $\bot$, on which there is only a winning strategy for π.

Therefore, the process (halt α) is a winning strategy on at least one of its channels.

Now consider the rule $\top_\ell$ (similarly $\bot_r$):

$$\frac{P :: \Phi \mid \Gamma \Vdash \Delta}{(\text{close }\alpha); P :: \Phi \mid \Gamma, \alpha : \top \Vdash \Delta}\ \top_\ell$$

$$\frac{P :: \Phi \mid \Gamma \Vdash \Delta}{(\text{close }\alpha); P :: \Phi \mid \Gamma \Vdash \alpha : \top, \Delta}\ \bot_r$$

Now, suppose that P is a winning strategy on at least one of its channels. Then (close α); P is the strategy which first loses on α and then continues on P. Therefore, since P wins on at least one of its external channels, we see that (close α); P is also a mediating strategy that wins on at least one of its channels (but that channel is not α).

Next let us consider the getting and putting rules:

$$\frac{P :: \Phi, n : A \mid \Gamma, \alpha : X \Vdash \Delta}{(\text{get } n \text{ on } \alpha); P :: \Phi \mid \Gamma, \alpha : A \circ X \Vdash \Delta} \circ_\ell$$

$$\frac{\Phi \vdash n : A \quad P :: \Psi \mid \Gamma, \alpha : X \Vdash \Delta}{(\text{put } n \text{ on } \alpha); P :: \Phi, \Psi \mid \Gamma, \alpha : A \bullet X \Vdash \Delta} \bullet_\ell$$

$$\frac{\Phi \vdash n : A \quad P :: \Psi \mid \Gamma \Vdash \alpha : X, \Delta}{(\text{put } n \text{ on } \alpha); P :: \Phi, \Psi \mid \Gamma \Vdash \alpha : A \circ X, \Delta} \circ_r$$

$$\frac{P :: \Phi, n : A \mid \Gamma \Vdash \alpha : X, \Delta}{(\text{get } n \text{ on } \alpha); P :: \Phi \mid \Gamma \Vdash \alpha : A \bullet X, \Delta} \bullet_r$$

Suppose that in each of these cases, P is a mediating strategy that wins on one of its external channels. The $n : A$ is the context is saying that in $\bullet_\ell$ and $\circ_r$, there is some n available to use for this strategy, and in $\bullet_r$ and $\circ_\ell$, we have a strategy for any n of type A. If the winning strategy for P was not on $\alpha : X$, then it is still a winning strategy for (get n on α); P, and (put n on α) : P.

Now, suppose that P was a winning strategy on $\alpha : X$. Then, if it has an $n : A$, as in the cases $\bullet_\ell$ and $\circ_r$, it can choose a path corresponding to that $n : A$ in the parity game for $A \bullet X$ or $A \circ X$ respectively. So (put n on α); P has a winning strategy on α.

On the other hand, if it has a winning strategy for any $n : A$ on X, that is the same as having a winning strategy for all the X in the games for $A \bullet X$ or $A \circ X$. So, even if P is not the player choosing which path corresponding to an element of A, P still has a winning strategy for that game. So (get n on α); P has a winning strategy on α.

Next, we will consider the $\otimes$ and $\oplus$ rules. Recall that the game corresponding to $X \otimes Y$ is the combined game X and Y, in which σ must win both games to win $X \otimes Y$, likewise $X \oplus Y$ is the combined game X And Y, however σ needs only win at least one game to win $X \oplus Y$. First, we look at $\otimes_r$

$$\frac{P_1 :: \Phi \mid \Gamma_1 \Vdash \Delta_1, \alpha_1 : X \quad P_2 :: \Psi \mid \Gamma_2 \Vdash \alpha_2 : Y, \Delta_2}{\text{fork } \alpha \text{ as } \left\{ \begin{array}{c} \alpha_1 \to P_1 \\ \alpha_2 \to P_2 \end{array} \right\} :: \Phi, \Psi \mid \Gamma_1, \Gamma_2 \Vdash \Delta_1, \alpha : X \otimes Y, \Delta_2} \otimes_r$$

Here we assume that P_1 is a mediating strategy that wins on one of Γ_1, Δ_1, X, and P_2 is a mediating strategy that wins on one of Γ_2, Y, Δ_2. Then the process fork α as $\left\{ \begin{array}{c} \alpha_1 \to P_1 \\ \alpha_2 \to P_2 \end{array} \right\}$ is represented by the strategy which interleaves playing the P_1 strategy on Γ_1, Δ_1 and X, with playing the P_2 strategy on Γ_2, Δ_2, and Y.

So, suppose that P_1 is a winning strategy on a channel Y in Γ_1, Δ_1. Then the new process is also a winning strategy in Y, as it plays the same over that channel. Otherwise, suppose that P_1 is not a winning strategy on Γ_1, Δ_1. Then, since it is a winning strategy on at least one of its channels, it must win on X. So, suppose P_2 wins on some Z in Γ_2, Δ_2. Then the new process wins on Z. Otherwise, P_2 must win on Y, so the new process wins on X and Y. Since it is playing as σ on X and Y, we see that it will thus win on $X \otimes Y$. Now consider $\oplus_\ell$:

$$\frac{P_1 :: \Phi \mid \Gamma_1, \alpha_1 : X \Vdash \Delta_1 \quad P_2 :: \Psi \mid \alpha_2 : Y, \Gamma_2 \Vdash \Delta_2}{\text{fork } \alpha \text{ as } \left\{ \begin{array}{c} \alpha_1 \to P_1 \\ \alpha_2 \to P_2 \end{array} \right\} :: \Phi, \Psi \mid \Gamma_1, \alpha : X \oplus Y, \Gamma_2 \Vdash \Delta_1, \Delta_2} \oplus_\ell$$

The analysis here is identical to that of $\otimes_r$. The difference is that the process plays as π on X and Y, and thus if it wins on both X and Y, it wins on $X \oplus Y$, as required. So, let us consider $\oplus_r$:

$$\frac{P :: \Phi \mid \Gamma \Vdash \alpha_1 : X, \alpha_2 : Y, \Delta}{(\text{split } \alpha \text{ into } \alpha_1, \alpha_2); P :: \Phi \mid \Gamma \Vdash \alpha : X \oplus Y, \Delta} \oplus_r$$

In this case, we assume that P is a mediating strategy that wins on at least one of Γ, X, Y, and Δ. Note that the strategy of (split α into α_1, α_2); P is the same as P, only now considering the channels $\alpha_1 : X$ and $\alpha_2 : Y$ as the single channel $\alpha : X \oplus Y$. This makes sense, as the $\oplus$ of two games is just the combined game, where you can make a move on either whenever, just as a mediating strategy does between its games.

Suppose that it wins on some channel Y in Γ, Δ. Then the new process does as well, as it is the same strategy. On the other hand if it wins on X or Y, we see that it also wins on $X \oplus Y$ as it is playing as σ in $X \oplus Y$, and thus only needs to win in at least one of X, Y to win the $X \oplus Y$ game. So the new process also wins on at least one of its channels. Consider now, $\otimes_\ell$:

$$\frac{P :: \Phi \mid \Gamma, \alpha_1 : X, \alpha_2 : Y \Vdash \Delta}{(\text{split } \alpha \text{ into } \alpha_1, \alpha_2); P :: \Phi \mid \Gamma, \alpha : X \otimes Y \Vdash \Delta} \otimes_\ell$$

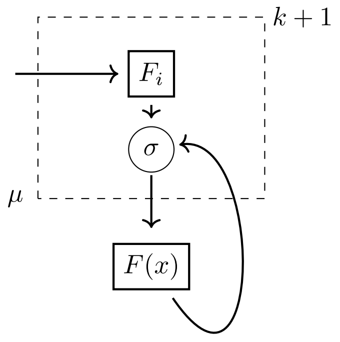
The analysis here is the same as $\oplus_r$. The only differences are that the process now plays as π on X and Y. And thus, only needs to win in at least one of them to win in $X \otimes Y$.

So, splitting and forking preserves the property of being a winning strategy on at least one of the external channels.

Now, we must consider the protocols and co-protocols. Consider the hput rules:

$$\frac{P :: \Phi \mid \Gamma \Vdash \alpha : F_i(\mu x.F(x)), \Delta}{(\text{hput } F_i \text{ on } \alpha); P :: \Phi \mid \Gamma \Vdash \alpha : \mu x.F(x), \Delta} \text{hput}_r$$

So, suppose that P is a winning strategy on one of $\Gamma, F_i(\mu x.F(x)), \Delta$ If it is on a member of Γ or Δ, then we see that (hput F_i on α); P is also a winning strategy on that channel, as a single move on α will not change the strategy on the other channels. On the other hand, consider the parity game of $F_i(\mu x.F(x))$:



There are two cases for this game. Either the strategy on $F_i(\mu x.F(x))$ allows the game to progress through the edge coming out of the F_i, or it does not. Suppose that it does not. Then, if we define the strategy on $\mu x.F(x)$, such that σ first chooses to go to F_i, then this is a winning strategy for σ, as it can win in F_i without passing through the bottom edge.

On the other hand, suppose that it does pass through that edge. Then, it may enter a cycle in which it passes back up to level $k+1$. Now, since this is a winning strategy on $F_i(\mu x.F(x))$, we see that this cannot happen an infinite number of times. Thus, if must either terminate within some constructor of $F(x)$, or have an infinite cycle completely within that constructor. So, we can still set the strategy for $\mu x.F(x))$ to first choose F_i and then continue with the strategy for $F_i(\mu x.F(x))$. The only difference is that it will pass through the top node one more time. However, this remains finite, so it is still a winning strategy.

Consider the other hput rule:

$$\frac{P :: \Phi \mid \Gamma, \alpha : G_i(\nu x.G(x)) \Vdash \Delta}{(\text{hput } G_i \text{ on } \alpha); P :: \Phi \mid \Gamma, \alpha : \nu x.G(x) \Vdash \Delta}\ \text{hput}_\ell$$

Here the argument is the same, but from the persepective of π.

Next consider the hcase rule.

$$\frac{P_1 :: \Gamma \Vdash \alpha : G_1(\nu x.G(x)), \Delta \quad \dots \quad P_n :: \Gamma \Vdash \alpha : G_n(\nu x.G(x)), \Delta}{\text{hcase } \alpha \text{ of } \left\{ \begin{array}{c} G_1 \to P_1 \\ \vdots \\ G_n \to P_n \end{array} \right\} :: \Phi \mid \Gamma \Vdash \alpha : \nu x.G(x), \Delta}\ \text{hcase}_r$$

We can suppose that all of P_1, up through P_n have winnings strategies on one of Γ, Δ, of $G_i(\nu x.G(x)$. Then, we have a winning strategy on $\Gamma \Vdash \nu x.G(x), \Delta$. Since the first move for the game of $\nu x.G(x)$ belongs to π, we must have a strategy for what to do in the case of any of these moves. However, for each possible move, it reduces to the case of one of the P_i. Thus, as we have a winning strategy for each of those, we can simply execute the strategy corresponding to the factor of $G(x)$ that π chooses to go down. Thus, there is a winning strategy for this process as well.

The argument is similar for the hcase_ℓ rule.

Next, we will consider drive:

$$\begin{array}{|c|}
\hline
\forall \alpha : X \;\; \Phi \mid \Gamma \Vdash \alpha : X, \Delta \\
\hline
\\
\dfrac{\dfrac{\vdots}{P_1 :: \Phi \mid \Gamma \Vdash \alpha : G_1(x), \Delta} \quad \ldots \quad \dfrac{\vdots}{P_n :: \Phi \mid \Gamma \Vdash \alpha : G_n(x), \Delta}}{X : \Phi \mid \Gamma \Vdash G(x), \Delta} \\
\hline
\end{array}$$

$$\text{drive } f(X|\Gamma \Rightarrow \Delta) \text{ with } T \text{ by } \alpha \text{ where } \left\{ \begin{array}{c} G_1 \to P_1 \\ \vdots \\ G_n \to P_n \end{array} \right\} :: T : \Phi \mid \Gamma \Vdash \alpha : \nu x.G(x), \Delta$$

Here, we are given a strategy for $G(x)$, where we allow the usage of an extra axiom, that is, a strategy on channels of type $\Gamma \Vdash X, \Delta$. Using this strategy corresponds to using the move in the $\nu x.G(x)$ parity game pointing out of $G(x)$ back to the top. Here, we suppose that we have a winning strategy on $G(x)$, where passing out through the bottom arrow is considered a win. However, we want to show that this is a winning strategy on $\nu x.G(x)$. Since every time we pass through that bottom arrow we restart the game, we have two cases. Either, we eventually win on this game within some G_i, or we pass through the arrows pointing out of the bottom of each G_i an infinite number of times.

In the first case, σ can win on $\nu x.G(x)$, since the strategy always satisfies a winning condition within some G_i. In the second case, σ can win on $\nu x.G(x)$, since we have an infinite path that passes through the top level of $\nu x.G(x)$ an infinite number of times. Since the top level is labeled with ν, we see that this is a win for σ.

Similarly, if we use the drive_ℓ rule, we can perform this same argument as π.

Now, we must consider the cut rule:

$$\frac{P_1 :: \Phi \mid \Gamma_1 \Vdash \Delta_1, \alpha : X \quad P_2 :: \Psi \mid \beta : X, \Gamma_2 \Vdash \Delta_2}{\text{plug } \{P_1(\alpha), P_2(\beta)\} :: \Phi, \Psi \mid \Gamma_1, \Gamma_2 \Vdash \Delta_1, \Delta_2} \text{ cut}$$

Suppose that P_1 has some winning strategy on one of $\Gamma_1, \Delta_1, \alpha$, and P_2 has a winning strategy on some $\Gamma_2, \beta, \Delta_2$. Since α and β are the same type X, we see that there either exists a winning strategy for σ, or a winning strategy for π on X, but not both. Suppose there is a winning strategy for σ. Then there cannot be a winning strategy for π, so P_2 cannot have a winning strategy on β. Thus, it has a winning channel of one of Γ_2, Δ_2 Therefore P also has a winning strategy on Γ_2, Δ_2, as it follows the same strategy as P_2 on these channels.

Similarly, if there is a winning strategy for π on X, then P_1 cannot have a winning strategy in X. Therefore, it has a winning strategy on one of Γ_1, Δ_1. So P does as well. Therefore, when two processes that are winning mediating strategies are cut together, the new process is also a winning mediating strategy.

Therefore, if P is a well-formed MPL program, it encodes a winning mediating strategy on its external channels.

$\square$

Now we can use this theorem to prove that MPL is live-lock free using a similar method as we did to prove that MPL is dead-lock free.

Theorem 2.7. *MPL is livelock-free*

Proof: Let P be a well-formed MPL program. Then, by the propositions in the deadlock-free section, P has a (list of) connectivity tree(s). Now consider some node in this tree. Each node corresponds to either an atomic process, which is also a well-formed MPL program, or a strategy. By the above theorem, every MPL program has a winning strategy on one of its channels, thus every atomic process does as well. Therefore, we can produce a map from atomic processes to channels, where each atomic process is mapped to a channel on which it has a winning strategy. Note that this map must be injective, since if each channel can only have a winning strategy on one of its ends.

Now, suppose that P is in livelock. Then, there exists a subset of the nodes and edges in the connectivity forest for which every node in this subset either performs an infinite number of evaluation steps or is eventually closed, and every edge in the connectivity forest has either an infinite number of messages passed over it or is eventually closed. Since MPL is deadlock-free, we see that this subset must be non-empty. Note that if an edge is in this subset, so are the nodes on each sides. Thus this is a subgraph of the connectivity forest. Since every subgraph of a forest is a forest, we see that this subset is also a forest. Since P is in livelock, we see that none of the edges are external channels, and every node is an atomic process.

Now, since this subgraph is a forest, we see that there are more nodes than edges. Therefore, there cannot be an injective map from nodes in this subgraph to edges in this subgraph. Therefore, the map from atomic processes to channels on which they have a winning strategy, cannot be contained within this subgraph. Therefore, there is some node in this graph which has a winning strategy on some channel outside of this graph. So this node with either eventually close, or it will perform an infinite number of evaluation steps. Thus it will perform every evaluation step in a particular route on its strategy. So, it must win on every channel on which it has a winning strategy. However, at least one of these channels is outside of this graph. But, a win involves either the closure of a channel, or an infinite number of messages passed. Therefore, this channel must be inside the graph. This is a contradiction. Therefore P cannot be in livelock.

□

References

[1] Samson Abramsky and Radha Jagadeesan. Games and full competeness for multiplicative linear logic. *The Journal of Symbolic Logic*, 1994.

[2] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR 2010 - Concurrency Theory*, pages 222–236, Paris, France, September 2010.

[3] R. Cockett and Spencer D. Strong categorical datatypes ii. *Theoretical Computer Science*, 139:69–113, 1995.

[4] R. Cockett and C. Pastro. The logic of message passing. *Science of Computer Programming*, 74:498–553, 2009.

[5] Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.

[6] Prashant Kumar. Implementation of message passing language. Master's thesis, University of Calgary, 2018.

[7] Herlihy M. and Shavit N. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[8] Herlihy M. and Shavit N. On the nature of progress. *OPODIS*, 2011.

[9] Dimitris Mostrous. Proof nets as processes. Technical report, University of Lisbon, Department of Informatics, 10 2012.

[10] Luigi Santocanale. From parity games to circular proofs. *Electronic Notes in Theoretical Computer Science*, 65:1–12, 2002.

[11] Luigi Santocanale. μ-bicomplete categories and parity games. *Theoretical Informatics and Applications*, 36:195–227, 2002.

[12] K. C. T. Kuo-Chung Tai. Definitions and detection of deadlock, livelock, and starvation in concurrent programs. In *1994 Internatonal Conference on Parallel Processing Vol. 2*, volume 2, pages 69–72, Aug 1994.

[13] V. T. Vasconcelos. Fundamentals of session types. *Information and Computation*, 217:52–70, 2012.