Concurrent MPL

July 27, 2017

1 Concurrent MPL

For the ease of understanding, MPL programs can be conceptually divided into two levels, namely concurrent MPL programs and sequential MPL programs. In this chapter, various constituents of the concurrent MPL programs are described.

Concurrent MPL programs consist of protocols, coprotocols and processes. Protocols and coprotocols are the concurrent analogues of data types and codata types respectively from sequential MPL. In concurrent MPL, programs are structured using processes. The processes communicate with each other by sending messages over channels. Section 2 describes channels, Section 3 describes processes and Section 4 describes (co)protocols.

2 Channels

In message passing view of concurrency, channels are important as they act as the conduit for the exchange of messages between processes.

Figure 1 describes a concurrent MPL program with three processes $\mathbf{P_1}$, $\mathbf{P_2}$ and $\mathbf{P_3}$ represented as circles. A description of the components of the diagram and the relationship between them is below:

- **ch1** connects process $\mathbf{P_1}$ with process $\mathbf{P_2}$ which is drawn as a solid line between the two processes. Channel **ch1** connects the two processes by acting as an output polarity channel for $\mathbf{P_1}$ and an input polarity channel for $\mathbf{P_2}$. The `(+)` sign on channel **ch1** towards $\mathbf{P_1}$ and `(-)` sign towards $\mathbf{P_2}$ suggests that **ch1** acts as an output channel for process $\mathbf{P_1}$ and an input channel for process $\mathbf{P_2}$.
- Similarly, **ch2** connects processes $\mathbf{P_2}$ and $\mathbf{P_3}$ by acting as an output channel for $\mathbf{P_2}$ and an input channel for $\mathbf{P_3}$.
- Normal MPL channels like **ch1** and **ch2** are attached to a process on its either ends.
- In addition to a normal channel **ch1**, process $\mathbf{P_1}$ also has a service channel **s1**. Service channels allow a process to interact with the outside world. For example, using a service channel a process can ask the user to enter an input or it can display its output to the user.
- The outside world has been represented through dotted circles in order to differentiate them from normal processes. The service channels have been represented by dotted lines to differentiate them from normal channels.

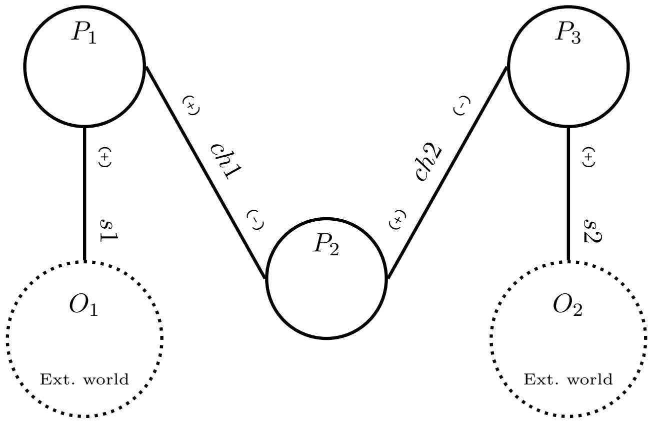

Figure 1: Processes Connected by Channels

- A noticeable thing about service channels is that they are connected to processes at only one end and not at both the ends like normal channels. Channel **s1** acts as an output polarity service channel as suggested by the `(+)` sign towards process $\mathbf{P_1}$.
- Service channels can be of either polarity, output or input. Input polarity service channels have a special name, **console**.

3 Processes

Concurrent MPL programs are organised using processes. An example of an MPL process definition has been provided in Figure 2. A few noticeable things about process definitions in MPL are below:

- An MPL process is defined with the keyword **proc** followed by the process name. The process name of the process defined in the example is **someProc**.
- An MPL process has sequential and concurrent parameters divided by | sign. Sequential parameters hold the sequential terms are passed to the process when a process call is made. Concurrent parameters are the channels of that process which can be both input and output. => separates the input polarity channels from the output polarity channels.
- In the example, the second line of process definition contains the process parameters, `val | co => s1`. **val** is the sequential parameter, **co** is the input concurrent parameter or the input channel and **s1** is the output concurrent parameter or the output channel.
- MPL also allows the programmer to annotate the processes with a type. In the example, the type annotation of **someProc** function is present in the first line followed by `::` symbol. The type of the process is `Int | Console(A) => IntTerm(A)` which means that the type of the sequential argument is **Int**, the type of the input channel is a coprotocol **Console**(**A**) and the type of the output polarity channel is a protocol **IntTerm**(**A**).
- The process body is comprised of a sequence of concurrent MPL constructs on the right hand side of `->`. In the example, the process body starts from line 2. Keyword **do** is used to define a layout for the the concurrent constructs underneath it, i.e the concurrent constructs can be grouped using alignment and using curly braces is not required.

```
proc someProc :: Int | Console (A) => IntTerm (A) =
    val | co => s1 -> do
        hput PutInt on s1
        put val on s1
        hput Close on s1
        close s1
        hput CloseC on co
        halt co
```

Figure 2: Example of an MPL Process

- A few examples of concurrent MPL constructs are get, put, id, hput, hcase etc. Concurrent MPL constructs have been discussed in Section 5.

4 Protocols/Coprotocols

An important feature of MPL is that it brings a notion of type safety to the concurrent world. (Co)protocols are central to this notion of type safety. A (co)protocol is the type of a channel. A protocol represents the type of an input channel and a coprotocol represents the type of an output channel.

4.1 Achieving Type Safety Using Protocols

Consider the example of a program in Figure 3 that demonstrates how MPL achieves type safety using protocols. The program has two processes $\mathbf{P_1}$ and $\mathbf{P_2}$ connected via a channel **ch**. Both the processes try to get a value on the channel **ch**. get is an MPL construct that fetches a value from a channel and put is an MPL construct that places a value on a channel. close construct closes a channel.

Intutitively, the configuration in Figure 3 seems wrong because both the processes can't simulatenously get a value on a channel. One process needs to put a value on a channel before another process can read that value from the channel. A configuration like this should result in type error and halt the execution of the program.

The strategy MPL uses to throw a type error in a configuration like this is to infer the protocols of channel **ch** from the two processes it is connecting and check that the two protocols match. If the two protocols don't match a type error is reported.

Corresponding to every MPL construct, there is an associated base protocol. In order to infer the protocol of channel **ch**, consider a few base protocols, **Get**, **Put** and **TopBot**. This is not the exhaustive set of base protocols. The set of all the base protocols have been provided in section 4.2.

- On an output channel, get and put constructs result in **Get** and **Put** base protocols respectively.

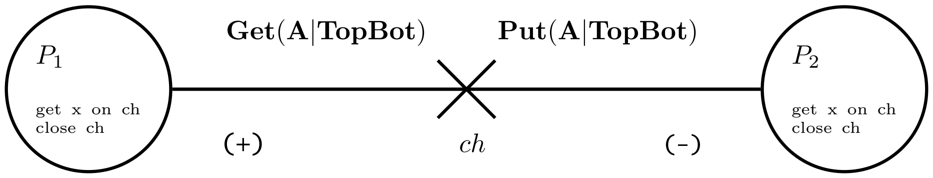

Figure 3: Example of an MPL Process

- However, on an input channel, `get` and `put` constructs result in **Put** and **Get** base protocols respectively. This is done so that the protocols obtained from the processes on the opposite polarities of the channel match.
- Closing a channel of any polarity results in the protocol **TopBot**.

On inferring the protocol of the channel **ch** from processes $\mathbf{P_1}$ and $\mathbf{P_2}$, **Get**(**A** | **TopBot**) and **Put**(**A** | **TopBot**) are obtained respectively. Since, the two protocols don't agree, a type error in thrown halting the execution of the program.

4.2 Base Protocols

All the (co)protocols in MPL are constructed using the following base protocols.

- Get
- Put
- $\otimes$ (Tensor)
- $\oplus$ (Par)
- TopBot

4.3 Defining (Co)Protocols in MPL

MPL allows the programmers to define complex (co)protocols using base protocols. An example of user defined (co)protocol definitions can be seen in Figure 4. A few observations about the protocol definition are below:

- A protocol is defined using the keyword **protocol**.
- The name of the defined protocol in the example is **IntTerm**. This protocol is polymorphic in type variable **A**.
- The syntax `IntTerm(A) => P` is inspired by the syntax of a data type. In data type definition this syntax is used to convey that a data type is an initial algebra. Here, the same syntax has been used to suggest that protocols are data types of the concurrent world. Very naively, **P** can be thought as a synonym or a place holder for **IntTerm**(**A**).
- Protocol **IntTerm** has three constructors, **GetInt**, **PutInt** and **Close**. In concurrent MPL, these constructors are known as handles.
- If the type of a channel is **IntTerm** then it means that any of the three handles of **IntTerm** can be put on the channel. Once a handle has been put on a channel, the only actions allowed on the channel are the actions permitted by the protocols of that handle.

```
protocol IntTerm (A) => P =
      GetInt :: Get (A|P) => P
      PutInt :: Put (A|P) => P
      Close  :: TopBot    => P

coprotocol CP => Console (A) =
    GetIntC   :: CP => Put (A|CP)
    PutIntC   :: CP => Get (A|CP)
    CloseC    :: CP => TopBot
```

Figure 4: (Co)Protocol Definition in MPL

- **GetInt** and **PutInt** handles are recursive.

Coprotocols are the codata types of the concurrent world and its syntax tries to mimick the syntax of codata types. Coprotocols are associated with the input channels. The destructors of the coprotocols are known as cohandles in concurrent MPL.

5 Concurrent MPL constructs

Concurrent MPL constructs are the constructs using which concurrent MPL programs are written. They are plug, id, hput-hcase, get-put, split-fork and close-halt. A few noticeable observations about concurrent MPL constructs are below:

- MPL constructs with the exception of the run contruct perform an action on a channel. In MPL concurrency is modeled using message passing between processes and channels are the conduit through which the messages are exchanged. So, it is only logical that most constructs deal with channel.
- Most concurrent MPL constructs come in pair, i.e get-put, hput-hcase, split-fork and halt-close. id, run and plug are standalone constructs. The constructs of the pair are dual to each other. This is intuitive because in message passing view of concurrency for a process to respond to an action, another process should drive that action on the opposite end of the channel and viceversa. Thus one of the constructs in any pair of constructs is a driver construct and the other is a reaction construct. For example, for a process to get a value on a channel, some other process should have put a value on the channel. Thus, get is a reaction construct driven by put construct.

A brief description of all the concurrent MPL constructs has been provided in Table 1. In the next few sections (Section 5.1 to Section 5.7), various concurrent MPL constructs are described in details.

5.1 run Construct

run construct is used to call an already defined process. This can be thought of as the concurrent counterpart of a function call. To successfully run (call) a process from other processes, :

- the process being called should already be defined and
- the process call should happen with appropriate arguments, i.e the number and type of the process arguments in run should be the same as the number and type of the parameters in the process definition.

run	calls a process
id	equates two channels
plug	connects two processes by a channel
get	gets a value on a channel
put	puts a value on a channel
hput	puts a handle on a channel
hcase	cases on the handles obtained on channel
split	splits a channel into two channels
fork	forks two new processes
close	closes a channel
halt	closes a channel. Usually the last channel is halted.

Table 1: Machine Transitions for the SAMPL

5.1.1 run Example

Table 2 shows an example of an MPL program where `run` construct is used. The output of the program is integer 5 printed on its output service channel, intTerm1. Notesworthy facts about the program are below:

- Program defines a recursive protocol, IntTerm and a recursive coprotocol, Console for the input and output service channels respectively.
- Inside the main process, process **p1** is `run` (called) with 3 arguments, one sequential and two concurrent. The sequential argument in the process call is the integer value **5**. Concurrent arguments of a process call are the channels of that process. `=>` separates concurrent input arguments (input channels) from concurrent output arguments (output channels). The concurrent input argument in the example is the service channel console and the concurrent output argument is the service channel intTerm1.
- The process definition for **p1** has 1 sequential parameter, 1 input concurrent parameter and 1 output concurrent parameter. Thus the process call is consistent with the process definition with respect to the number of arguments of each type.

5.2 plug Construct

`plug` command connects two processes via a channel. Once the two processes are connected, they can exchange messages. Syntax of plug is shown below:

```
plug
    p1 ( | console => ch)
    p2 ( | ch => intTerm1)
```

Syntax consists of the keyword `plug` along with the processes being plugged. For a channel to be successfully plugged between two processes,

- it should act as an output channel for one process and an input channel for the other process, and
- the protocol of the channel inferred from both the processes it is plugged between should be the same.

```
protocol IntTerm (A) => P =
    GetInt    :: Get (A|P) => P
    PutInt    :: Put (A|P) => P
    Close     :: TopBot    => P

coprotocol CP => Console (A) =
    GetIntC   :: CP => Put (A|CP)
    PutIntC   :: CP => Get (A|CP)
    CloseC    :: CP => TopBot

proc p1 :: Int | Console (A) => IntTerm (A) =
    val | co => s1 -> do
        hput PutInt on s1
        put val on s1
        hput Close on s1
        close s1
        hput CloseC on co
        halt co

run => intTerm2 -> do
      p1 ( 5 | console => intTerm1 )
```

Table 2: Example : run construct

One can plug multiple processes using the plug construct as can be seen below where **ch1** is plugged between **p1** and **p2** and **ch2** is plugged between **p2** and **p3**.

```
plug
    p1 ( | console => ch1)
    p2 ( | ch1 => ch2)
    p3 ( | ch2 => intTerm1)
```

5.3 get-put Constructs

These are the simplest MPL constructs. get receives a value on a channel and put puts a value on a channel.

- On an output channel, get results in **Get** protocol and put results in **Put** protocol.
- On an input channel, get and put constructs result in **Put** and **Get** protocols respectively.

Thus, get/put constructs result in different protocols depending on the polarity of the channel over which they act. This is done in order to infer the same protocol for a channel from the two processes attached at the two ends of the channel.

5.3.1 get-put Example

Table 3 shows an example of an MPL program that uses get-put construct. Figure 5 shows the diagrammatic representation for the example program in Table 3.

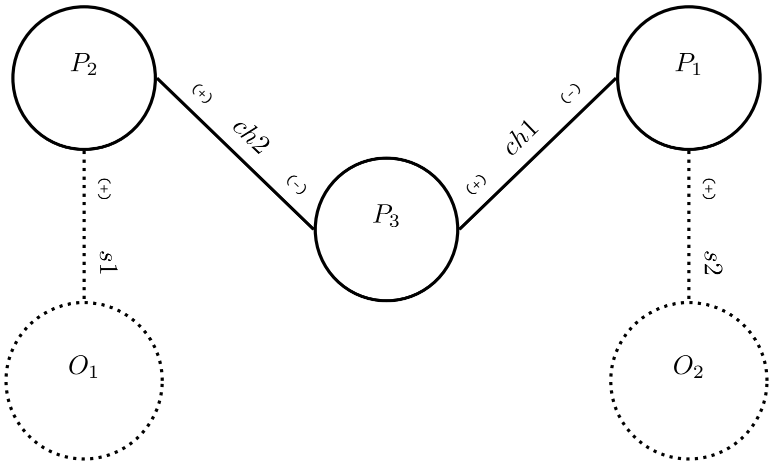

Figure 5: get-put Example

In the program there are three processes, **p1**, **p2** and **p3** apart from the main process. These processes have been represented by three circles named as $\mathbf{P_1}$, $\mathbf{P_2}$ and $\mathbf{P_3}$ respectively in the diagram.

Channel **ch2** is plugged between processes **p2** and **p3** and channel **ch1** is plugged between processes **p3** and **p1**. **ch2** acts as an output channel for process **p2** and an input channel for process **p3**. The (+) sign on **ch2** at process $\mathbf{P_2}$'s end signifies that **ch2** acts as an output channel for process $\mathbf{P_2}$. The (-) sign on **ch2** at process $\mathbf{P_3}$'s end signifies that **ch2** acts as an input channel for process $\mathbf{P_3}$. Both $\mathbf{P_2}$ and $\mathbf{P_1}$ have an output service channel named **S1** and **S2** respectively.

When the program is executed, a terminal (service channel intTerm1) pops up asking for a value to be entered. Once the user has entered a value, another terminal (service channel intTerm2) pops up displaying this value.

Process $\mathbf{P_2}$ asks for a value on its service channel **S1** after putting the appropriate handle on the channel. $\mathbf{P_2}$ then puts this value on its output channel **ch2**.

Process $\mathbf{P_3}$ reads the value on channel **ch2** which is an input channel for $\mathbf{P_3}$. $\mathbf{P_3}$ then puts this value on its output channel **ch1**.

Process $\mathbf{P_1}$ reads the value on channel **ch1** which is an input channel for $\mathbf{P_1}$. $\mathbf{P_1}$ then puts this value on service channel **S2** after putting the appropriate handle. This value is now visible to the outside world.

5.3.2 Concurrency Semantics and get Before put

The program in Table 3 is simple to understand if one considers that the the processes are scheduled and executed in the order $\mathbf{P_2}$, $\mathbf{P_3}$, and $\mathbf{P_1}$. i.e, $\mathbf{P_2}$ gets a value and passes it to $\mathbf{P_3}$ which in turn passes to $\mathbf{P_1}$. However, the concurrency semantics for MPL allows the processes to be scheduled in any order. For example - order $\mathbf{P_1}$, $\mathbf{P_2}$, and $\mathbf{P_3}$ is also a valid schedule of the processes. The problem with the second schedule is that process $\mathbf{P_1}$ is requesting a get on its input channel **ch1** even before process $\mathbf{P_3}$ had the opportunity to put the value on that channel. Similarly, process $\mathbf{P_3}$ is trying to get a value on channel **ch2** even before $\mathbf{P_2}$ puts a value on **ch2**.

```
protocol IntTerm (A) => P =
    GetInt    :: Get (A|P) => P
    PutInt    :: Put (A|P) => P
    Close     :: TopBot    => P


proc p2 :: |  => IntTerm (Int),Put(Int|TopBot) =
    |  => s1,ch2 -> do
        hput GetInt on s1
        get x on s1
        put x on ch2
        hput Close on s1
        close s1
        halt ch2


proc p3 :: |  Put (Int|TopBot) => Put (Int|TopBot) =
    | ch2 => ch1 -> do
       get x on ch2
       put x on ch1
       close ch2
       halt ch1

proc p1 :: | Put (Int|TopBot) => IntTerm (Int) =
     | ch1 => s2 -> do
        get x on ch1
        hput PutInt on s2
        put x on s2
        hput Close on s2
        close s2
        halt ch1

run => intTerm1,intTerm2 -> do
    plug
      p2 ( | => intTerm1,ch2)
      p3 ( | ch2 => ch1)
      p1 ( | ch1 => intTerm2 )
```

Table 3: Example : get-put constructs

MPL handles the above mentioned scenario of trying to get a value before put has happened by putting the process requesting the get operation to sleep and removing it from the set of active processes. Once the value has been put on the channel, the process requesting get is worken up and put in the set of active processes from where it can scheduled. This mechanism is used not only with get-put but also with other concurrent construct pairs like split-fork and hput-hcase, when the process containing the reaction construct of the pair is scheduled before the process containing the driver construct.

5.4 id Construct

id construct equates two channels. Syntax `ch2 |=| ch1` means that channel **ch1** has been equated with channel **ch2** meaning:

- messages written on **ch1** are now available on **ch2**, i.e one can read messages available on **ch1** using **ch2**, and
- messages to be sent via **ch1** can now be sent via **ch2**, i.e. one can write a message on **ch2** such that it can be read on **ch1**.

5.4.1 id Example

Table 4 represents an example of a program using the id construct. The process connectivity diagram for the program has been shown in Figure 6.

Process $\mathbf{P_1}$ has an input polarity service channel **s1** and an output channel **ch1**. Process $\mathbf{P_2}$ has an input channel **ch1** and an output channel **ch2**. Process $\mathbf{P_2}$ equates channel **ch1** with channel **ch2**. Process $\mathbf{P_3}$ has an input channel **ch2** and an output polarity service channel **s2**.

Process $\mathbf{P_1}$ gets a value from outside world on **s1** and puts this value on **ch1**. Since, **ch1** and **ch2** have been equated, the value put on **ch1** is available on **ch2** which is read by process $\mathbf{P_3}$. $\mathbf{P_3}$ subsequently puts the value obtained on **ch2** on its service channel **s2**. Similarly, process $\mathbf{P_3}$ reads a value on channel **s2** which is received by $\mathbf{P_1}$ via process $\mathbf{P_2}$. $\mathbf{P_1}$ then outputs the received value on its service channel **s1**.

5.5 hput-hcase Constructs

hput puts a (co)handle of a (co)protocol on a channel. A handle is hput on a channel at one end and hcased at the other end. hcase can be thought of as concurrent counter part of the case construct from the sequential MPL. case branches on the constructors of a data type and executes a sequential construct corresponding to the data constructor selected from the branch. hcase branches on the handles of a protocol and run a sequence of concurrent commands corresponding to the selected handle from the branch.

One hputs a handle on an output channel and hcases on handles on an input channel. Conversely, one hputs a cohandle on an input channel and hcases on cohandles on an output channel.

5.5.1 hput-hcase Example

Table 5 shows an example of a program that uses the hput-hcase constructs. The process connectivity diagram for the program has been shown in Figure 7.

```
coprotocol CP => Console (A) =
    GetIntC   :: CP => Put (A|CP)
    PutIntC   :: CP => Get (A|CP)
    CloseC    :: CP => TopBot

protocol IntTerm (A) => P =
    GetInt   :: Get (A|P) => P
    PutInt   :: Put (A|P) => P
    Close    :: TopBot    => P

proc p1 =
  | s1 => w1 -> do
       hput GetIntC on s1
       get val on s1
       put val on w1
       get val2 on w1
       hput PutIntC on s1
       put val2 on s1
       close w1
       hput CloseC on s1
       halt s1

proc p2 =
  | w1 => w2 -> do
       w2 |=| w1

proc  p3 =
    | w2 => s2 -> do
       get val on w2
       hput PutInt on s2
       put val on s2
       hput GetInt on s2
       get val2 on s2
       put val2 on w2
       hput Close on s2
       close s2
       halt w2


run console => intTerm1 -> do
    plug
      p1 ( | console => w1)
      p2 ( | w1 => w2)
      p3 ( | w2 => intTerm1)
```

Table 4: Example : id construct

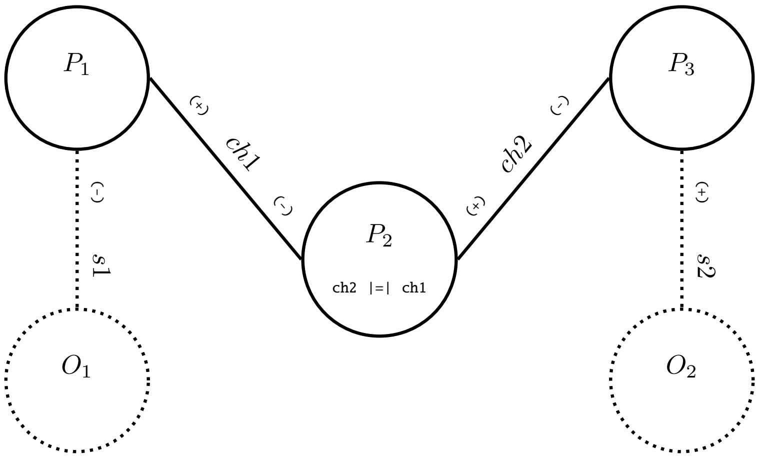

Figure 6: id Example

Process $\mathbf{P_1}$ has an output service channel **i1** and a normal output channel **ch**. Process $\mathbf{P_2}$ has an output service channel **i2** and a normal input channel **ch**. **ch** has been plugged between processes $\mathbf{P_1}$ and $\mathbf{P_2}$.

Process $\mathbf{P_1}$ performs the following actions in sequence:

- It reads an integer value on **i1**
- It hputs a handle **PutInt** on channel **ch** indicating that it plans to put a value on the channel.
- It puts the value read from on **i1** on the channel **ch**.
- It hputs a handle **GetInt** on channel **ch** indicating that it expects to receive a value on the channel.
- It receives a value on the channel **ch** and then puts this value on **i1**.

Process $\mathbf{P_2}$ hcases on the handle obtained on channel **ch** and performs the following actions for the below handles:

- **GetInt** : Corresponding to this handle, the process reads value from its service channel **i2** and puts it on channel **ch**.
- **PutInt** : Corresponding to this handle, the process reads a value from channel **ch** and puts it on **i2**.
- **Close** : Corresponding to this handle, the channels are closed and the process is terminated.

The final output is a terminal that pops up and asks for a integer value. Once the value is entered, another terminal pops up and displays the value entered on the first terminal. The second terminal then asks for a value which is displayed in the first terminal. The popping terminals are the service channels.

```
oprotocol CP => Console (A) =
    GetIntC   :: CP => Put (A|CP)
    PutIntC   :: CP => Get (A|CP)
    CloseC    :: CP => TopBot

protocol IntTerm (A) => P =
    GetInt   :: Get (A|P) => P
    PutInt   :: Put (A|P) => P
    Close    :: TopBot    => P

proc p1 =
    | co => i1,ch -> do
        hput GetInt on i1
        get x on i1
        hput PutInt on ch
        put x on ch
        hput GetInt on ch
        get y on ch
        hput PutInt on i1
        put y on i1
        hput Close on i1
        close i1
        hput Close on ch
        close ch
        hput CloseC on co
        halt co

proc p2 =
    | ch => i2 ->
        hcase ch of
            GetInt -> do
                hput GetInt on i2
                get x on i2
                put x on ch
                p2 ( | ch => i2)

            PutInt -> do
                get x on ch
                hput PutInt on i2
                put x on i2
                p2 ( | ch => i2)

            Close  -> do
                hput Close on i2
                close i2
                halt ch

run console => intTerm1,intTerm2 ->
      plug
         p1 ( | console => ch,intTerm1)
         p2 ( | ch => intTerm2)
```

Table 5: Example : hput-hcase constructs

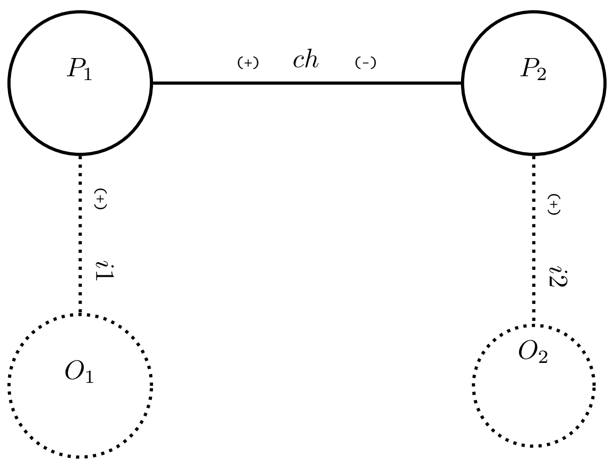

Figure 7: hput-hcase Example

5.6 split-fork Constructs

Creating/forking new processes is a natural requirement in many concurrent programs. In MPL, creation of new processes is achieved using the fork construct that creates two new processes from an old process. In MPL processes are usually connected to other processes via channeels. However, after forking both the newly forked processes need their own channel to maintain the connectivity that the parent process had with the other processes. split takes a channel and divides it into two channels.

split and fork are dual to each other. Splitting an input channel or forking on an output channel results in the protocol $\otimes$ (tensor). Splitting an output channel or forking on an input channel results in the protocol $\oplus$ (par).

5.6.1 split-fork Example

Table 6 shows an example of a program that uses the split-fork constructs. The configuration of the processes before and after the split-fork occurs is shown through the two diagrams in Figure 8. The first diagram shows the before configuration and the second diagram shows the after configuration.

In the before diagram, process $\mathbf{P_1}$ is connected to process $\mathbf{P_2}$ by a channel **ch**. **ch** is an output channel for $\mathbf{P_1}$ and an input channel for $\mathbf{P_2}$. Process $\mathbf{P_1}$ has an output service channel, **interm1** and process $\mathbf{P_2}$ has two output service channels, **intTerm2** and **intTerm3**.

In the after diagram, $\mathbf{P_1}$ splits channel **ch** into **ch1** and **ch2**. $\mathbf{P_2}$ forks two new processes $\mathbf{P_1}$ and $\mathbf{P_2}$ on the newly generated channels **ch1** and **ch2**. Consider the type of the newly generated channels $\mathbf{ch_1}$ and **ch2** as T_1 and T_2 respectively. The type T of the original channel **ch** in this configuration is $T_1 \oplus T_2$ as the output channel **ch** is split and input channel **ch** is being forked on.

Channel **ch** has been plugged between processes $\mathbf{P_1}$ and $\mathbf{P_2}$. Process $\mathbf{P_1}$ gets a value on its service channel **intTerm1**, splits its output channel **ch** into two channels **ch1** and **ch2**, and puts the obtained value from the service channel on **ch1** and **ch2**. Process $\mathbf{P_2}$ forks two new processes $\mathbf{P_{21}}$ and $\mathbf{P_{22}}$ on the channels **ch1** and **ch2** respectively. $\mathbf{P_{21}}$ then gets the value that was put on $\mathbf{ch_1}$, squares it and puts the square on its service channel **intTerm2**. Similarly, $\mathbf{P_{22}}$ gets the value

```
protocol IntTerm (A) => P =
      GetInt   :: Get (A|P) => P
      PutInt   :: Put (A|P) => P
      Close    :: TopBot    => P

coprotocol CP => Console (A) =
      GetIntC   :: CP => Put (A|CP)
      PutIntC   :: CP => Get (A|CP)
      CloseC    :: CP => TopBot

proc p1 =
  | console => ch,intTerm1 -> do
        hput GetInt on intTerm1
        get x on intTerm1
        split ch into ch1,ch2
        put x on ch1
        put x on ch2
        close ch1
        close ch2
        hput Close on intTerm1
        close intTerm1
        hput CloseC on console
        halt console

proc p21 =
    | ch1 => intTerm2 -> do
       get x on ch1
       hput PutInt on intTerm2
       put x*x on intTerm2
       hput Close on intTerm2
       close intTerm2
       halt ch1
```

```
proc p22 =
    | ch2 => intTerm3 -> do
       get x on ch2
       hput PutInt on intTerm3
       put x*x*x on intTerm3
       hput Close on intTerm3
       close intTerm3
       halt ch2

proc p2  =
    | ch => intTerm2,intTerm3 ->
       fork ch as
         ch1 -> do
           p21 ( | ch1 => intTerm2)
         ch2  -> do
           p22 ( | ch2 => intTerm3)

run console => intTerm1,intTerm2,intTerm3 ->
      plug
         p1 ( | console => ch,intTerm1)
         p2 ( | ch => intTerm2,intTerm3)
```

Table 6: Example : split-Fork construct

that was put on $\mathbf{ch_2}$, cubes it and puts the cube on its service channel **intTerm3**

5.7 close-halt Constructs

Before closing an MPL process, it is required to close all the channels associated with that process. close-halt constructs are used to close a channel. By convention all the channels except the last channel are closed. The last channel is halted signifying the halting of the process.

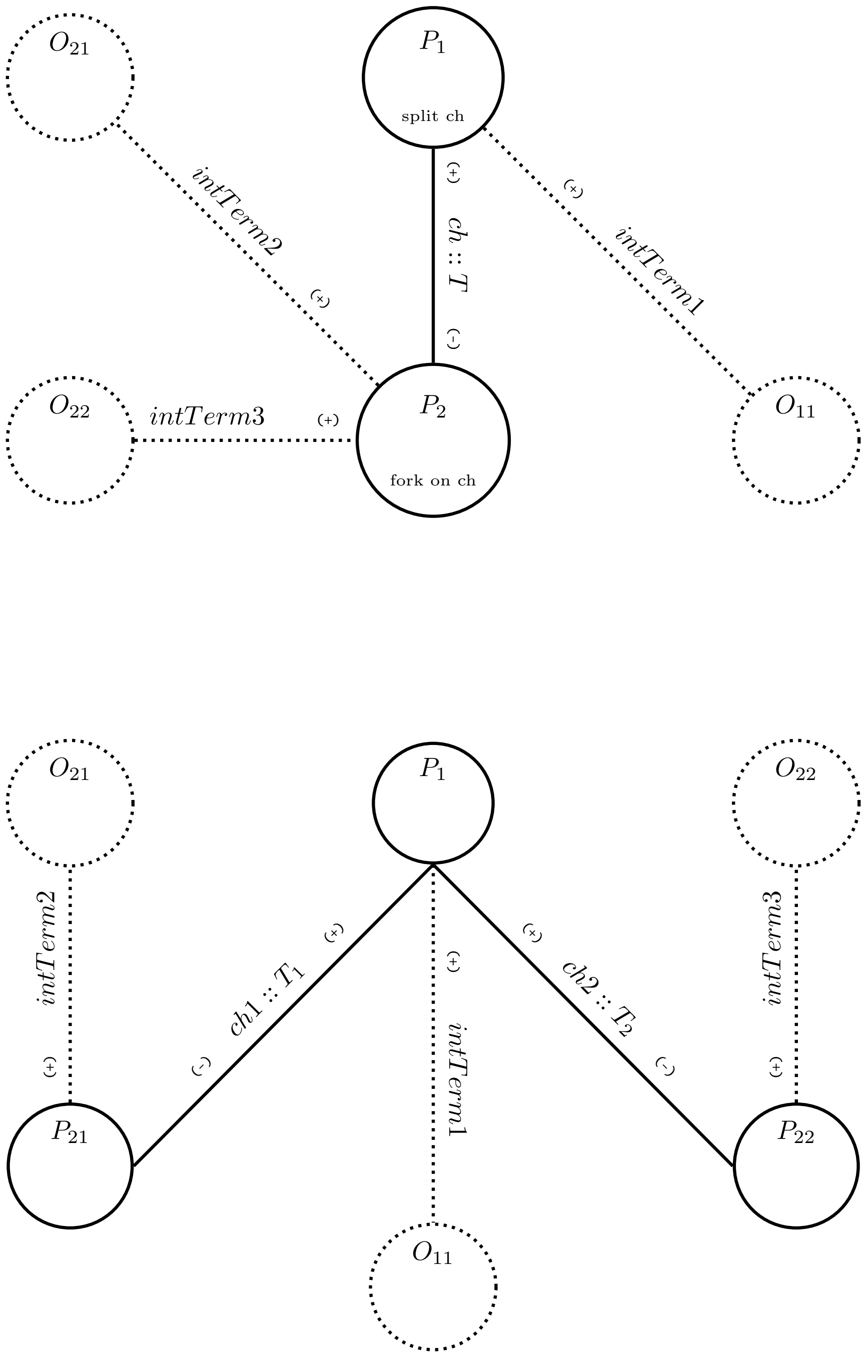

Figure 8: split-Fork Example $(T = T_1 \oplus T_2)$