

Abstract machine for MPL (AMPL)

June 28, 2017

1 Abstract Machine for MPL (AMPL)

In the previous chapters the different stages of interpretation of MPL programs (listed in Figure in 1) have been described. In this chapter, the last stage of interpretation, namely running MPL programs on MPL's abstract machine (called AMPL) is discussed.

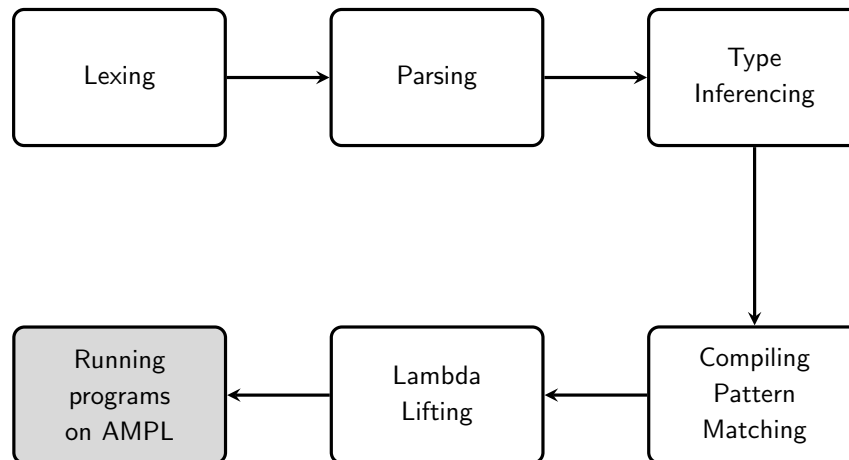


Figure 1: Compilation Stages

1.1 Introduction to Abstract Machines

Abstract machines are useful conceptual tools when implementing a programming language because they omit the many details of real machines and thus bridge the gap between the programming language and the physical machine. An added advantage of abstract machines is the ease of porting the language to different platforms. An abstract machine one reduces the problem of porting the programming language to the problem of porting the abstract machine. This is an easy task as the abstract machines are simpler and smaller than the programming language itself. This strategy was used by the programming language Java which used the abstract machine **Java Virtual Machine (JVM)**. Some famous examples of *abstract machines* are:

- Landin's SECD Machine was one of the first abstract machines for functional programming languages specifically designed to evaluate lambda calculus expressions **by value**.

- Modern-SEC Machine is an improved and efficient version of the SECD machine.
- Cardelli’s Functional Abstract Machine (FAM) is an extended and optimized SECD machine. It was used in the first native-code implementation of ML.
- Cousineau, Curien and Mauny’s Categorical Abstract Machine (CAM). Its instructions correspond to the constructions of a *Cartesian Closed Category*: identity, composition, abstraction, application, pairing, and selection. It was the basis for the CAML programming language.
- Three Instruction Machine (TIM) is a simple abstract machine for evaluation of super-combinators.
- Spineless-Tagless G-Machine (STG-Machine) is the abstract machine for Haskell.
- Warren Abstract Machine (WAM) is the abstract machine for Prolog.

1.2 Introduction to AMPL

AMPL is an abstract machine which runs MPL programs. Just as MPL is separated into two levels, **Sequential MPL** and **Concurrent MPL**, so too can AMPL can be thought of as two machine levels:

- **Sequential AMPL (SAMPL)** which runs **Sequential MPL code**
- **Concurrent AMPL (CAMPL)** which runs **Concurrent MPL code**

This conceptual separation means AMPL has a modular description and design. Thus, either of the AMPL components can be modified without disturbing the other.

2 Sequential Abstract Machine for MPL (SAMPL)

Sequential MPL code compiles to Sequential AMPL(SAMPL). SAMPL is inspired by modern-SEC machine. However, the modern-SEC machine doesn’t have explicit commands to which data types and codata types can be compiled. SAMPL augments the basic modern-SEC machine with **case** and **constructor** commands to compile data types and **record** and **destructor** commands to compile codata types. SAMPL also adds commands for basic functions like addition, multiplication, equality testing etc which are built-in to the language. The SAMPL commands are evaluated **by-value** except for the **record** and **destructor**, the codata type commands, which are lazily evaluated.

2.1 SAMPL Commands

Table 1 provides a list of SAMPL commands and a brief description of each command.

2.2 Compilation of Sequential MPL Code to SAMPL Commands

Sequential MPL code is converted to a list of SAMPL commands which are then executed on the abstract machine described in Table 2. Sequential MPL is translated to SAMPL Code. The

| Instruction | Explanation |
|---|--|
| Store Access(n) Ret call $\langle \text{code} \rangle$ | pushes the top stack element into the environment put n^{th} value in the environment onto the stack. return the top stack value and jump to the continuation below jump to the code |
| Built in instructions: | |
| Const $_T(k)$ Add Mul Leq | push the constant k of basic type T on the stack Pop two arguments from the top of the stack and add them Pop two arguments from the top of the stack and multiply them Pop two arguments from the top of the stack and compare them etc. |
| Data instructions: | |
| Cons(i, n) Case $[c_1, \dots, c_n]$ | push the i^{th} constructor onto the stack with arguments the top n elements of the stack, Cons(i, s_1, \dots, s_n) . when Cons(i, t_1, \dots, t_n) is on the stack remove it and push t_1, \dots, t_n into the environment and evaluate c_i . |
| Codata instructions: | |
| Rec $[c_1, \dots, c_n]$ Dest(i, n) | create a record on the stack with current environment, rec($[c_1, \dots, c_n], e$) destruct a record: choose the i^{th} function closure (c_i, e) and run c_i in environment e supplemented with the first n values on the stack. |

Table 1: Machine Transitions for the SAMPL

translation scheme is described below:

$$\llbracket \text{record } \{D_i \ x_1, \dots, x_{n_i} \mapsto t_i\}_{i \in [1, m]} \rrbracket_v = \text{Rec}[\llbracket t_i \rrbracket_{x_1, \dots, x_{n_i}, v} \text{Ret}]_{i \in [1, m]} \quad (1)$$

$$\llbracket (D_i(x, t_1, \dots, t_n)) \rrbracket_v = \llbracket t_n \rrbracket_v \dots \llbracket t_1 \rrbracket_v \llbracket x \rrbracket_v \text{Dest}(i, n) \quad (2)$$

$$\llbracket \text{Cons}_i(t_1, \dots, t_n) \rrbracket_v = \llbracket t_n \rrbracket_v \dots \llbracket t_1 \rrbracket_v \text{Cons}(i, n) \quad (3)$$

$$\llbracket \text{case } t \{ \text{Cons}_i \ x_1, \dots, x_{n_i} \mapsto t_i \}_{i \in [1, m]} \rrbracket_v = \llbracket t \rrbracket_v \text{Case}[\llbracket t_i \rrbracket_{x_1, \dots, x_{n_i}, v} \text{Ret}]_{i \in [1, m]} \quad (4)$$

$$\llbracket x \rrbracket_v = \text{Access}(n) \quad \text{where } n = \text{index } v \ x \quad (5)$$

$$\llbracket a \text{ op } b \rrbracket_v = \llbracket b \rrbracket_v \llbracket a \rrbracket_v \text{Op} \quad (6)$$

$$\llbracket k \rrbracket_v = \text{Const}_T(k) \quad (7)$$

$\llbracket \rrbracket_v$ signifies the compilation of a *Core MPL*'s sequential constructs to *SAMPL* commands in context of an *environment* v . The environment acts as a repository of variables used in a piece of code being compiled. It is used to replace a variable name with the relative position in the program body.

A description of these compilation steps is:

(1) is the compilation scheme for the **record** construct. D_i is the i^{th} destructor and t_i is the corresponding term for that destructor.

(2) is the compilation scheme for a **destructor** represented by D_i . The first argument of D_i is

the record to be destructed and the subsequent arguments are the arguments of D_i .

(3) is the compilation of a **constructor** $Cons_i$. The terms t_1, \dots, t_n which are the arguments of the constructors are recursively compiled and concatenated in the order opposite to which they occur. The constructor name is gotten rid in the compilation process and is instead represented by a pair, the constructor number and the number of arguments that the constructor takes.

(4) is the compilation of a **case** construct. A few things worth noting in the compilation of **case** constructs are as follows:

- The term t_i corresponding to the $Cons_i$ is compiled in the context enhanced with the arguments of $Cons_i$.
- **Ret (return)** should be the last command in the compiled code corresponding to any constructor $Cons_i$.
- Before the compilation of the **Case** construct starts, it should be ensured that the constructors of the data type are arranged in the same order as they have been defined.

(5) is the compilation of a **variable** x . The variable is looked up in the context v . The depth of the variable in the context is the argument to the **Access** command.

(6) is the compilation of infix functions. These functions are converted to their postfix forms with their arguments recursively compiled.

2.3 Transition Table for SAMPL

Once **Sequential MPL** has been compiled to SAMPL code, the code can then be executed on the abstract machine. The *Code* is represented as a list of SAMPL commands. Two other structures namely the *Environment* and the *Stack* need to be introduced in order to show how the machine works. Both *Environment* and *Stack* are Last In First Out (LIFO) data structures. In our implementation of MPL, they have been implemented using Haskell lists.

The *Environment* keeps track of which variables are visible in the code at a given point and is thus a transient data structure: it changes as one traverses through the code. The *stack* acts as storage for the intermediate values and the final result during the execution of the code.

The state of SAMPL at any given time can be described as the triple of the compiled SAMPL *Code*, *Environment*, and *Stack* represented as (C, E, S) . Executing SAMPL code means starting with an initial state and changing states based on the transition table for SAMPL described in Table 2 until a final state has been reached. The final state contains the result of the computation. The selection of transition at any step of code execution is done by **pattern matching** the machine state against the transition table. Thus, the given description of the SAMPL is easily implemented in Haskell using its pattern matching feature.

The initial state of the machine is formed by putting the compiled code as the first argument of the triple and initialising the *Environment* and *Stack* as empty lists.

$$\text{Initial State} : (C, [], [])$$

The final state is reached when the *Code* and the *Environment* is empty. The top of the *Stack*

contains the output value.

Final State : ($[\]$, $[\]$, S)

In Table 2, $\text{clos}(c, e)$ denotes closure of *Code* c with *Environment* e and $e(n)$ is the n^{th} -element of the environment.

| Before | | | After | | |
|---------------------------------|-----|---|-------|--------------------------|---|
| Code | Env | Stack | Code | Env | Stack |
| Store; c | e | $v : s$ | c | $v : e$ | s |
| Access(n); c | e | s | c | e | $e(n) : s$ |
| Call(c) : c' | e | s | c | e | $\text{clos}(c', e) : s$ |
| Ret : c | e | $v : \text{clos}(c', e') : s$ | c' | e' | $v : s$ |
| Cons(i, n) : c | e | $v_1 : \dots, v_n : s$ | c | e | $\text{cons}(i, [v_1, \dots, v_n]) : s$ |
| Case(c_1, \dots, c_n) : c | e | $\text{Cons}(i, [v_1, \dots, v_n]) : s$ | c_i | $v_1 : \dots : v_n : e$ | $\text{clo}(c, e) : s$ |
| Rec(c_1, \dots, c_n) : c | e | s | c | e | $\text{rec}([c_1, \dots, c_n], e) : s$ |
| Dest(i, n) : c | e | $\text{rec}([c_1, \dots, c_n], e') : v_n : \dots : v_1 : s$ | c_i | $v_1 : \dots : v_n : e'$ | $\text{clo}(c, e) : s$ |
| Const $_T(k)$: c | e | s | c | e | $\text{const}_T(k) : s$ |
| Add : c | e | $n : m : s$ | c | e | $(n + m) : s$ |
| Mul : c | e | $n : m : s$ | c | e | $(n * m) : s$ |
| Leq : c | e | $n : m : s$ | c | e | $(n \leq m) : s$ |

Table 2: Machine Transitions for the SAMPL

2.4 Example : Compiling And Executing Sequential MPL Programs

Table 3 provides an example of compiling Sequential MPL to SAMPL. The Sequential MPL code being compiled is a function $f1$ whose body **destructs** a record with a **destructor** App . App takes two arguments: a **record** to destruct and a second argument. The second argument 2 is indirectly an argument to **record** $App(y) := y + 1$ being destructed. The function $f1$ is equivalent to the lambda application $(\lambda x. x + 1) 2$. The function $f1$ demonstrates MPL's technique of simulating higher-order functions using codata types.

Second row of Table 3 shows an example of the step by step compilation of function $f1$ to its corresponding SAMPL code. The compilation steps have been labelled to identify which rules have been used in any given step.

Table 4 shows an example of the step by step execution of SAMPL commands. The code selected for execution is the SAMPL code generated for function $f1$ in Table 3. In Table 4, too keep the table readable, c has been used for CInt 1; Access 1; Add; Ret .

Execution of the code starts with empty *Environment* and *Stack*. In the final step *Code list* and the *Environment* are empty and the result is obtained on top of the *Stack*.

| | |
|--|----------------------|
| MPL Program corresponding to $(\lambda x. x + 1) 2$ | |
| <pre> codata C -> Exp(A,B) = App :: A,C -> B fun f1 = -> App(record(App(y) := y + 1),2) </pre> | |
| Step wise compilation to SAMPL | |
| $\llbracket \text{App}(\text{record}(\text{App}(y) := y + 1, 2)) \rrbracket_{[]}$ | (Dest. rule) |
| $\llbracket 2 \rrbracket_{[]} ++ \llbracket \text{record}(\text{App}(y) := y + 1) \rrbracket_{[]} ++ [\text{Dest } 1 \ 1]$ | (Const. & Rec rule) |
| $[\text{CInt } 2] ++ \text{Rec}[\llbracket y + 1 \rrbracket ++ [\text{Ret}]] ++ [\text{Dest } 1 \ 1]$ | (Op (Infix) rule) |
| $[\text{CInt } 2] ++ \text{Rec}[\llbracket 1 \rrbracket_{[y]} ++ \llbracket y \rrbracket_{[y]} ++ [\text{Add}] ++ [\text{Ret}]] ++ [\text{Dest } 1 \ 1]$ | (Const. & Var. rule) |
| $[\text{CInt } 2] ++ \text{Rec}[\text{CInt } 1] ++ [\text{Access } 1] ++ [\text{Add}] ++ [\text{Ret}]] ++ [\text{Dest } 1 \ 1]$ | |
| $[\text{CInt } 2, \text{Rec}[\text{CInt } 1, \text{Access } 1, \text{Add}, \text{Ret}], \text{Dest } 1 \ 1]$ | (SAMPL code) |
| Compiled SAMPL Code | |
| <pre> [CInt 2, Rec [CInt 1, Access 1, Add, Ret], Dest 1 1] </pre> | |

Table 3: Example : Compilation of Sequential MPL to SAMPL

| Code | Env | Stack |
|----------------------------|------------------------|--|
| CInt(2); Rec[c]; Dest 1 1 | ε | ε |
| Rec[c]; Dest 1 1 | ε | cint 2 : ε |
| Dest 1 1 | ε | rec([c], ε) : cint 2 : ε |
| CInt 1; Access 1; Add; Ret | cint2 : ε | clo(ε , ε) : ε |
| Access(1) : Add : Ret | cint2 : ε | cint 1 : clo(ε , ε) : ε |
| Add : Ret | cint 2 : ε | cint 2 : cint 1 : clo(ε , ε) : ε |
| Ret | cint 2 : ε | cint 3 : Clo(ε , ε) : ε |
| ε | ε | cint 3 : ε |

Table 4: Executing Code on SAMPL

3 Concurrent Abstract Machine for MPL (CAMPL)

Concurrent Abstract Machine for MPL (CAMPL) is the machine on which the Concurrent MPL code is run. Concurrent MPL code comprises of processes. The state of a process is represented as a four tuple of *Stack*, *Translation*, *Environment*, and *Code* (S, t, E, C). *Stack*, *Environment* and *Code* are also used to represent the state of Sequential MPL programs and have the same meaning here.

- The *Stack* holds the intermediate value and the final result when executing a process.
- The *Translation* is an additonal structure used in describing the state of a *Concurrent MPL* program. This additonal structure is required because Concurrent MPL programs have channels, a feature that the sequential MPL programs lack. *Translation* acts as a map of the local channel names to global channel names. This is required because a channel connecting two processes may be named differently inside each process. Processes can then use the *translation* to determine which local channels refer to which gobal channels.
- The *Environment* is used to determine which variables are visible at a given point of the Concurrent program.
- The *Code* is the list of Concurrent AMPL commands corresponding to the MPL program.

Initial State : ([], t , [], C)

The initial state of CAMPL is formed by loading the compiled code corresponding to Concurrent MPL program (processes) as the fourth argument of the tuple, the *Translation* of channels corresponding to a process as the second argument, and intialising the *Environment* and *Stack* which are the first and the third arguments respectively, as empty lists.

Final State : (S , [], [], [])

The final state is reached when the *Code*, the *Environment* and the *Translation* is empty. The top of the stack contains the output of the code.

CAMPL consists of two main components, the **Channel Mananger**, \mathcal{C} and the **Process Manager**, \mathcal{P} .

3.1 Channel Manager

The **Channel Manager**, \mathcal{C} is a set of pairs of channels and their queues. It is represented as $\mathcal{C}\{(\alpha, q' \mid q)\}$ which indicates that a channel α has the communication queue $q' \mid q$ associated with it. A communication queue has two parts q' , the output polarity queue and q , the input polarity queue. If a value x is being put on a channel, it will be added to the front of the output queue, represented as $x : q'$. If a channel α is expecting a value y , then y is put at the back of the input queue represented as $q : y$. It is the first element of the output queue and last element of the input queue (represented as x and y in the representation $(x : q' \mid q : y)$ which are the subject to communication actions described in section 3.4.2.

3.2 Process Manager

The **Process Manager** is a set of processes. It is represented as $\mathcal{P}\{(S, t, E, C)\}$ signifying that it has selected one process that will be advanced by one execution step. The **Process Manager** selects this process non-deterministically. In practise one may want to execute this non-deterministically selected process not just by one step but for multiple steps, until it has a concurrent action, before interrupting it.

3.3 Interaction between Process Manager and Channel Manager

The interaction between the **Process Manager** and the **Channel Manager** can be understood with the help of some examples.

Suppose there are two MPL processes connected by a channel such that the first process **P1** puts a value on the channel and the second process **P2** receives the value on the channel. Since MPL assumes no fixed order of execution of processes, the process receiving the value on the channel may be executed before the process putting the value on the channel. When process **P2** is executed before **P1**, the **Channel Manager** suspends the process **P2** and attaches it to one of the queues (say output queue) of the channel, ch . When process **P1** is scheduled and executed, the value is obtained on the channel and put on its input queue. Once **Channel Manager** sees the configuration where a suspended process is attached to one queue of the channel and a value is present on the other queue of the channel, it reactivates the suspended process and puts it in the set of active processes. The **Process Manager** can then schedule this active process for execution.

Here the interaction between the **Process Manager** and **Channel Manager** has been explained in terms of just the **get-put** command pair. However, similar interaction between the two components of CAMPL takes places for **split-fork**, **hput-hcase** and **close-halt** pairs.

3.4 Concurrent Commands

In order to understand the working of CAMPL, we start by considering the CAMPL commands. Corresponding to every concurrent MPL construct, there is one CAMPL command. Table 5 lists the concurrent commands with a brief description of each. The execution of CAMPL commands have been described by describing the role of **Process Manager** and **Channel Manager** for each command. **Process Manager's** role is described in **Process Manager Actions** (see section 3.4.1) and **Channel Manager's** role is described in **Channel Manager Actions** (see section 3.4.2).

| | |
|--|---|
| get α ; C | get a value on channel α |
| put α ; C | put a value on channel α |
| split α into (α_1, α_2) ; C | split channel α into two (new) channels |
| fork α as α_1 with $\Gamma_1.C_1$ α_2 with $\Gamma_2.C_2$ | forking on a channel into two distinct processes |
| hput α n ; C | put a “handle” on channel α |
| hcase $\{C_1, \dots, C_n\}$ | the cases on receiving a “handle” |
| close α ; C | closing a channel |
| halt α | halting process attached to a single channel |
| plug $[\alpha_1, \dots, \alpha_n]$ $\Gamma_1.C_1$ $\Gamma_2.C_2$ | two processes to communicate on n channels |
| run t (process) | runs a process with local channel to caller channel translation t |
| id $\alpha = \beta$ | identifying channels. |

Table 5: Basic Concurrent Commands

3.4.1 Process Manager’s Actions

This section describes the execution step corresponding to the CAMPL commands. When a command executes it not only changes the state of the running process shown by $\mathcal{P}\{(s, t, e, c)\}$ but may also change the state of the queues associated with a channel. Hence, the **Transition Table** for CAMPL (described in Table 6) have columns for both **Process Manager** and **Channel Manager** in the **before** and the **after** column. An explanation of the **Process Manager’s** actions corresponding to the various CAMPL commands is provided below:

- **Get/Put:** These are the two basic communication commands. **put** transmits a value on a communication channel and **get** receives a value on a communication channel. To **put** a value on a channel one simply places the value on the input queue for that channel. To **get** a value the process suspends itself on the communication channel with the demand for a value. When the value appears (is put) on the channel the communication manager passes the value and enables the process waiting to get the value.
- **Split:** The **split** instruction splits the channel α into two channels (α_1 and α_2). One therefore adds to the appropriate queue notification of the two “global” channels into which the channels are split. This means one must choose two new channel names (here β_1 and β_2), and must remember the translation from the local channel names, $t[\beta_1/\alpha_1, \beta_2/\alpha_2]$. Once **split** is done, the machine continues with the execution of the remaining code.
- **Fork:** Fork command is dual to the **split** command is the fork command. This command creates two processes which are supposed to communicate on the new channels assigned by a split command. However, the splitting may not have happened when the fork command is executed. Thus, on a fork command the process suspends itself and attaches itself to the channel which is to be split. The process does not fork, however until the corresponding channel action of splitting is performed (as a communication action) ... and at that stage the translations for the local channel names to global names are adjusted and the forked processes are enabled (i.e. added into the process manager).
- **Plug:** The plug command allows two processes to communicate along certain of their channels. A plug command is a command in a process which has certain channels connecting it to the outside world. As in a fork command these channels must be divided amongst the two process

| | | | |
|---|--|---|--|
| $\mathcal{C}\{(t(\alpha), q' \mid q)\}$ | $\mathcal{P}\{(s, t, e, \text{get } \alpha; c)\}$ | $\mathcal{C}\{(t(\alpha), q' \mid q; \mathbf{g}(s, t, e, c))\}$ | $\mathcal{P}\{\}$ |
| $\mathcal{C}\{(t(\alpha), q' \mid q)\}$ | $\mathcal{P}\{(v:s, t, e, \text{put } \alpha; c)\}$ | $\mathcal{C}\{(t(\alpha), q' \mid q; v)\}$ | $\mathcal{P}\{(s, t, e, c)\}$ |
| $\mathcal{C}\{(t(\alpha), q' \mid q)\}$ | $\mathcal{P}\{(\llbracket, t, e, \text{split } \alpha \text{ into } (\alpha_1, \alpha_2); c\}\}$ | $\mathcal{C}\left\{\begin{pmatrix} t(\alpha), \\ q' \mid q: \langle \beta_1, \beta_2 \rangle \\ (\beta_1, \varepsilon) \\ (\beta_2, \varepsilon) \end{pmatrix}\right\}$ | $\mathcal{P}\{(\llbracket, t \left[\frac{\beta_1}{\alpha_1} \right], e, c)\}$ |
| $\mathcal{C}\{(t(\alpha), q' \mid q)\}$ | $\mathcal{P}\{(s, t, e, \text{close } \alpha; c)\}$ | $\mathcal{C}\{t(\alpha), q' \mid q; \text{close}\}$ | $\mathcal{P}\{(s, t \setminus \alpha, e, c)\}$ |
| $\mathcal{C}\{(t(\alpha), q' \mid q)\}$ | $\mathcal{P}\{(\llbracket, t, e, \alpha_1 \text{ with } \Gamma_{1.c_1}, \llbracket, \alpha_2 \text{ with } \Gamma_{2.c_2}\}$ | $\mathcal{C}\left\{\begin{pmatrix} t(\alpha), \\ q' \mid q: \left[t, e, \frac{\alpha_1}{\Gamma_{1.c_1}} \right] \\ \left[t, e, \frac{\alpha_2}{\Gamma_{2.c_2}} \right] \end{pmatrix}\right\}$ | $\mathcal{P}\{\}$ |
| $\mathcal{C}\{\}$ | $\mathcal{P}\{(\llbracket, t, e, \text{id } \alpha := \gamma; c, \llbracket)\}$ | $\mathcal{C}\{\}$ | $\mathcal{P}\{(\llbracket, t[t(\gamma)/\alpha], e, c)\}$ |
| $\mathcal{C}\{\}$ | $\mathcal{P}\{(\llbracket, t, e, \text{plug}[\alpha_1, \dots, \alpha_n]_{\Gamma_{1.c_1}, \Gamma_{2.c_2}}, \llbracket)\}$ | $\mathcal{C}\{(\gamma_i, \varepsilon \mid \varepsilon)_{i=1 \dots n}\}$ | $\mathcal{P}\left\{\begin{pmatrix} (\llbracket, t[\gamma_i/\alpha_i]_{\Gamma_1}, e, c_1), \\ (\llbracket, t[\gamma_i/\beta_i]_{\Gamma_2}, e, c_2) \end{pmatrix}\right\}$ |
| $\mathcal{C}\{(t(\alpha), q' \mid q)\}$ | $\mathcal{P}\{(\llbracket, t, e, \text{halt } \alpha)\}$ | $\mathcal{C}\{t(\alpha), q' \mid q; \text{halt}\}$ | $\mathcal{P}\{\}$ |
| $\mathcal{C}\{\}$ | $\mathcal{P}\{(s, t, e, \text{run } t' \text{ } c)\}$ | $\mathcal{C}\{\}$ | $\mathcal{P}\{(s, t; t', e', c)\}$ |
| $\mathcal{C}\{(t(\alpha), q' \mid q)\}$ | $\mathcal{P}\{(s, t, e, \text{hput } \alpha \text{ } n; c)\}$ | $\mathcal{C}\{(t(\alpha), q' \mid q; \text{h}(n))\}$ | $\mathcal{P}\{(s, t, e, c)\}$ |
| $\mathcal{C}\{(t(\beta), q' \mid q)\}$ | $\mathcal{P}\{(s, t, e, \text{hcase } \beta \{c_i\})\}$ | $\mathcal{C}\{(t(\beta), q' \mid q; (s, t, e, \text{hc}\{c_i\}))\}$ | $\mathcal{P}\{\}$ |

Table 6: Process execution steps (α with input polarity)

being plugged together: in addition, new global communication channels must be assigned to the channels along which they wish to communicate. After a plug command both processes are enabled (i.e. added into the process manager).

- **Handle:** The hput command, which sends a “handle” – a protocol constructor – which is matched by a listening process which will react according to which handle is received by an hcase command. These commands behave somewhat like a “put” and “get”, except that for the latter one chooses the code which is to be run based on the handle received rather than simply using the value in the subsequent code.
- **Call:** This “jumps” to some code for a predefined process. The only subtlety in the call command is that one must start by setting up the translation of the local channels of the predefined process into global channel names.
- **Close/Halt:** Closing a channel, close α , causes the channel to be removed. Corresponding to closing a channel is the halt command. Only when all other channels of a process have been closed can a process call a halt α on the one remaining channel.
- **Id $\alpha := \beta$:** This provides an identity map between two channels. It alters the translation of the process by defining the translation of α to be the same as the translation of β . This command is used, in particular, when “bending” wires to simulate the negation of linear logic.

3.4.2 Channel Manager’s Actions

This section describes how CAMPL deals with the dual command pairs of get-put, hput-hcase, split-fork and close-halt when get occurs before put, hcase occurs before hput, fork occurs before split. These commands are discussed with respect to conventional channels and not **Services**. **Channel Manager’s Actions** for **Services** are discussed in Section 3.4.3.

An explanation of the **Channel Manager’s** actions:

- **Communication of values or a handle:** When a value/handle is waiting on a channel and there is a suspended process waiting for the value then one can transmit the value and re-enable the process which was waiting. Row 1 of Table 7 shows the communication of **values**. Row 2 shows the communication of a **handle** facilitated by the **Channel Manager**.
- **Split/Fork communication:** When a suspended process which wishes to fork is waiting on a channel and the corresponding split has been made on the channel one assigns new global channel names to the channels introduced by the fork and enable the processes which are forked modifying the translations of the processes. Row 3 of Table 7 shows the **split – fork** communication facilitated by the **Channel Manager**.
- **Close/Halt communication:** A channel can be completely removed from the channel manger only when a **close** command matches a **halt** command. This is shown by the row 4 of Table 7.

| | | | |
|---|---------------|--|---|
| $\mathcal{C}\{(\beta, q:v \mid g(s, t, e, c))\}$ | \mathcal{P} | $\mathcal{C}\{(\beta, q \mid \varepsilon)\}$ | $\mathcal{P}\{(v:s, t, e, c)\}$ |
| $\mathcal{C}\{(t(\beta), q:h(i) \mid (s, t, e, hc \{c_j\})))\}$ | \mathcal{P} | $\mathcal{C}\{(\beta, q \mid \varepsilon)\}$ | $\mathcal{P}\{(s, t, e, c_i)\}$ |
| $\mathcal{C}\{(\beta, \langle \beta_1, \beta_2 \rangle \mid \left[\begin{array}{l} t, e, \alpha_1/\Gamma_1.c_1 \\ \alpha_2/\Gamma_2.c_2 \end{array} \right] \})\}$ | \mathcal{P} | $\mathcal{C}\{\}$ | $\mathcal{P}\left\{ \left(\left[\begin{array}{l} \beta_1/\alpha_1, e, c_1 \\ \beta_2/\alpha_2, e, c_2 \end{array} \right] \right) \right\}$ |
| $\mathcal{C}\{(\beta, \text{close} \mid \text{halt})\}$ | \mathcal{P} | $\mathcal{C}\{\}$ | \mathcal{P} |

Table 7: Channel Manager’s Actions for Normal Channels

3.4.3 Channel Manager’s Actions for Services

Services are built in channels in MPL that allow an MPL program to interact with the outside world. Differences in **Channel Manager**’s behaviour for services follow from the fact that **Services** are different from conventional channels:

- Conventional channels are connected to two MPL processes on their two ends whereas **Services** are connected to just one process.
- Actions like **plug**, **split** and **fork** can’t be done on **Service** channels. Permissible actions on **Services** are **hput**, **get-put** and **close-halt**.

The standard way to use a **Service** channel is to **hput** a handle on it and then do any of the **get**, **put**, **close**, or **halt** actions. A protocol commonly used with an output **Service** channel has been defined in Figure 2. The name of the protocol is **IntTerm** and one can recursively **get-put** on a **Service** and close a **Service** using the handles **GetInt**, **PutInt** and **Close** respectively. The **Channel**

```

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

```

Figure 2: Example : Protocol Definition for an Output Service Channel

Manager’s actions for an output **Service** channel have been described in Table 8. In Table 8, **h(1)**, **h(2)** and **h(3)** refer to the first, second and the third handle respectively.

| | | | |
|--|---------------------------------|--|---------------------------------|
| $\mathcal{C}\{(\beta, \varepsilon \mid \varepsilon : h(1) : \text{get})\}$ | $\mathcal{P}\{(s, t, e, c)\}$ | $\mathcal{C}\{(\beta, \varepsilon \mid \varepsilon)\}$ | $\mathcal{P}\{(v:s, t, e, c)\}$ |
| $\mathcal{C}\{(\beta, \varepsilon \mid \varepsilon : h(2) : v)\}$ | $\mathcal{P}\{(v:s, t, e, c)\}$ | $\mathcal{C}\{(\beta, \varepsilon \mid \varepsilon)\}$ | $\mathcal{P}\{(s, t, e, c)\}$ |
| $\mathcal{C}\{(\beta, \varepsilon \mid \varepsilon : h(3) : \text{close})\}$ | \mathcal{P} | $\mathcal{C} \setminus \beta$ | \mathcal{P} |
| $\mathcal{C}\{(\beta, \varepsilon \mid \varepsilon : h(3) : \text{halt})\}$ | \mathcal{P} | $\mathcal{C} \setminus \beta$ | \mathcal{P} |

Table 8: Channel Manager's Actions for Services(Output Channel β)

3.4.4 Example : Compiling and Executing Concurrent MPL Programs

This section shows an example of compiling and executing Concurrent MPL commands on CAMPL.

In Table 9 a simple MPL program is taken and its compiled CAMPL code is provided with the description of what each command does. The given MPL program consists of just one process, the **main process** which takes two numbers on an output **Service** channel named **intTerm1**, sums the numbers and displays them on the same **Service**.

In Table 10 and 11, the CAMPL code from Table 9 is executed stepwise based on the **Channel Manager's Actions for Services** from Table 8 and the **Process Execution Steps** from Table 6, till the final state is reached. If the program had normal channels in addition to the **Service** channels, then Table 7 detailing the **Channel Manager's Actions for Normal Channels** would also be referenced.

In Table 10 and 11,

- the notation $\mathcal{C} \{n, q \mid q'\}$ for **Channel Manager** means that channel number n has two associated queues, q and q' , and
- *Translation* represented by Γ and is a mapping of local channels to global channels. The translation $\Gamma [(0, 0), (-1, -1)]$ means that channels known locally as 0 and -1 are known globally by the same numbers. This is because all the channels in the given program are **Service** channels. **Service** channels are global channels and are known in every process by the same numbers. **Service** channels are represented by numbers less than or equal to 0. 0 is used for a special **Service** channel known as **Console**.

The first row of Table 10 describes the initial state in the execution of the main process. Notice the values of different columns in the first row. The **Channel Manager** starts with empty queues for all the channels in the main process. *Stack* and *Environment* are empty and *Translation* is loaded with local to global mapping of channels of the main process. **Code** is initialised to the compiled CAMPL code for the process generated in Table 9.

Last row of Table 11 describes the final state in the execution of the main process. Every column is empty except the **stack** which contains the sum of the numbers (output) as its top element.

| Before | After |
|---|---|
| <pre> protocol IntTerm (A) => P = GetInt :: Get (A P) => P PutInt :: Put (A P) => P Close :: Top => P coprotocol CP => Console (A) = GetIntC :: CP => Put (A CP) PutIntC :: CP => Get (A CP) CloseC :: CP => TopBot run console => intTerm1 -> do hput GetInt on intTerm1 get num1 on intTerm1 hput GetInt on intTerm1 get num2 on intTerm1 hput PutInt on intTerm1 put (num1+num2) on intTerm1 hput Close on intTerm1 close intTerm1 hput CloseC on console halt console </pre> | <pre> -- put first handle on channel -1 which is a service -- channel called intTerm1.First handle suggests get -- action on -1 hput -1 1 get -1 -- get a value on channel -1 store -- store the value obtained from -1 -- get another value from -1 and store hput -1 1 get -1 store access 2 -- load the second number access 1 -- load the first number add -- add the two numbers -- put second handle on channel -1. Second Handle -- suggests a value will be put on -2 hput -1 2 put -1 -- put the sum on channel -1 -- put third handle on channel -1. Handle 3 -- suggests that channel will be closed. hput -1 3 close -1 -- close channel -1 hput 0 3 -- put third handle on channel 0 halt 0 -- halt channel 0 </pre> |

Table 9: Example : Compilation of Concurrent MPL Program to CAMPL

| Channel Manager | Stack | Trans | Env | Code |
|--|---------------------------|---|---------------------------|--|
| $\mathcal{C} \left\{ \begin{pmatrix} 0, \varepsilon & & \varepsilon \\ -1, \varepsilon & & \varepsilon \end{pmatrix}, \right\}$ | ε | $\Gamma \left[\begin{pmatrix} 0, 0, \\ -1, -1 \end{pmatrix} \right]$ | ε | hput -1 1 : c_1 |
| $\mathcal{C} \left\{ \begin{pmatrix} 0, \varepsilon & & \varepsilon \\ -1, \varepsilon & & \varepsilon : h(1) \end{pmatrix}, \right\}$ | ε | $\Gamma \left[\begin{pmatrix} 0, 0, \\ -1, -1 \end{pmatrix} \right]$ | ε | get -1 : c_2 |
| $\mathcal{C} \left\{ \begin{pmatrix} 0, \varepsilon & & \varepsilon \\ -1, \varepsilon & & \varepsilon : h(1):get \end{pmatrix}, \right\}$ | ε | $\Gamma \left[\begin{pmatrix} 0, 0, \\ -1, -1 \end{pmatrix} \right]$ | ε | c_2 |
| $\mathcal{C} \left\{ \begin{pmatrix} 0, \varepsilon & & \varepsilon \\ -1, \varepsilon & & \varepsilon \end{pmatrix}, \right\}$ | $v_1 : \varepsilon$ | $\Gamma \left[\begin{pmatrix} 0, 0, \\ -1, -1 \end{pmatrix} \right]$ | ε | store : c_3 |
| $\mathcal{C} \left\{ \begin{pmatrix} 0, \varepsilon & & \varepsilon \\ -1, \varepsilon & & \varepsilon \end{pmatrix}, \right\}$ | ε | $\Gamma \left[\begin{pmatrix} 0, 0, \\ -1, -1 \end{pmatrix} \right]$ | $v_1 : \varepsilon$ | hput -1 1 : c_4 |
| $\mathcal{C} \left\{ \begin{pmatrix} 0, \varepsilon & & \varepsilon \\ -1, \varepsilon & & \varepsilon : h(1) \end{pmatrix}, \right\}$ | ε | $\Gamma \left[\begin{pmatrix} 0, 0, \\ -1, -1 \end{pmatrix} \right]$ | $v_1 : \varepsilon$ | get -1 : c_5 |
| $\mathcal{C} \left\{ \begin{pmatrix} 0, \varepsilon & & \varepsilon \\ -1, \varepsilon & & \varepsilon : h(1):get \end{pmatrix}, \right\}$ | ε | $\Gamma \left[\begin{pmatrix} 0, 0, \\ -1, -1 \end{pmatrix} \right]$ | $v_1 : \varepsilon$ | c_5 |
| $\mathcal{C} \left\{ \begin{pmatrix} 0, \varepsilon & & \varepsilon \\ -1, \varepsilon & & \varepsilon \end{pmatrix}, \right\}$ | $v_2 : \varepsilon$ | $\Gamma \left[\begin{pmatrix} 0, 0, \\ -1, -1 \end{pmatrix} \right]$ | $v_1 : \varepsilon$ | store : c_6 |
| $\mathcal{C} \left\{ \begin{pmatrix} 0, \varepsilon & & \varepsilon \\ -1, \varepsilon & & \varepsilon \end{pmatrix}, \right\}$ | ε | $\Gamma \left[\begin{pmatrix} 0, 0, \\ -1, -1 \end{pmatrix} \right]$ | $v_2 : v_1 : \varepsilon$ | access 2 : c_7 |
| $\mathcal{C} \left\{ \begin{pmatrix} 0, \varepsilon & & \varepsilon \\ -1, \varepsilon & & \varepsilon \end{pmatrix}, \right\}$ | $v_1 : \varepsilon$ | $\Gamma \left[\begin{pmatrix} 0, 0, \\ -1, -1 \end{pmatrix} \right]$ | $v_2 : v_1 : \varepsilon$ | access 1 : c_8 |
| $\mathcal{C} \left\{ \begin{pmatrix} 0, \varepsilon & & \varepsilon \\ -1, \varepsilon & & \varepsilon \end{pmatrix}, \right\}$ | $v_2 : v_1 : \varepsilon$ | $\Gamma \left[\begin{pmatrix} 0, 0, \\ -1, -1 \end{pmatrix} \right]$ | $v_2 : v_1 : \varepsilon$ | add : c_9 |
| $\mathcal{C} \left\{ \begin{pmatrix} 0, \varepsilon & & \varepsilon \\ -1, \varepsilon & & \varepsilon \end{pmatrix}, \right\}$ | $v_1 + v_2 : \varepsilon$ | $\Gamma \left[\begin{pmatrix} 0, 0, \\ -1, -1 \end{pmatrix} \right]$ | $v_2 : v_1 : \varepsilon$ | hput -1 2 : c_{10} |

Table 10: Executing Code on CAMPL (Continued On Table 11)

| Channel Manager | Stack | Trans | Env | Code |
|---|---------------------------|--|---------------------------|-------------------------------|
| $\mathcal{C} \left\{ \begin{array}{l} (0, \varepsilon \mid \varepsilon), \\ (-1, \varepsilon \mid \varepsilon : \mathbf{h}(2)) \end{array} \right\}$ | $v_1 + v_2 : \varepsilon$ | $\Gamma \left[\begin{array}{l} (0, 0), \\ (-1, -1) \end{array} \right]$ | $v_2 : v_1 : \varepsilon$ | put -1 : c_{10} |
| $\mathcal{C} \left\{ \begin{array}{l} (0, \varepsilon \mid \varepsilon), \\ (-1, \varepsilon \mid \varepsilon : \mathbf{h}(2):\mathbf{put}) \end{array} \right\}$ | $v_1 + v_2 : \varepsilon$ | $\Gamma \left[\begin{array}{l} (0, 0), \\ (-1, -1) \end{array} \right]$ | $v_2 : v_1 : \varepsilon$ | c_{10} |
| $\mathcal{C} \left\{ \begin{array}{l} (0, \varepsilon \mid \varepsilon), \\ (-1, \varepsilon \mid \varepsilon) \end{array} \right\}$ | $v_1 + v_2 : \varepsilon$ | $\Gamma \left[\begin{array}{l} (0, 0), \\ (-1, -1) \end{array} \right]$ | $v_2 : v_1 : \varepsilon$ | hput -1 3 : c_{11} |
| $\mathcal{C} \left\{ \begin{array}{l} (0, \varepsilon \mid \varepsilon), \\ (-1, \varepsilon \mid \varepsilon : \mathbf{h}(3)) \end{array} \right\}$ | $v_1 + v_2 : \varepsilon$ | $\Gamma \left[\begin{array}{l} (0, 0), \\ (-1, -1) \end{array} \right]$ | $v_2 : v_1 : \varepsilon$ | close -1 : c_{12} |
| $\mathcal{C} \left\{ \begin{array}{l} (0, \varepsilon \mid \varepsilon), \\ (-1, \varepsilon \mid \varepsilon : \mathbf{h}(3):\mathbf{close}) \end{array} \right\}$ | $v_1 + v_2 : \varepsilon$ | $\Gamma \left[\begin{array}{l} (0, 0), \\ (-1, -1) \end{array} \right]$ | $v_2 : v_1 : \varepsilon$ | c_{12} |
| $\mathcal{C} \left\{ (0, \varepsilon \mid \varepsilon) \right\}$ | $v_1 + v_2 : \varepsilon$ | $\Gamma \left[(0, 0) \right]$ | $v_2 : v_1 : \varepsilon$ | hput 0 3 : c_{13} |
| $\mathcal{C} \left\{ (0, \varepsilon \mid \varepsilon : \mathbf{h}(3)) \right\}$ | $v_1 + v_2 : \varepsilon$ | $\Gamma \left[(0, 0) \right]$ | $v_2 : v_1 : \varepsilon$ | halt 0 : ε |
| $\mathcal{C} \left\{ (0, \varepsilon \mid \varepsilon : \mathbf{h}(3):\mathbf{halt}) \right\}$ | $v_1 + v_2 : \varepsilon$ | $\Gamma \left[(0, 0) \right]$ | $v_2 : v_1 : \varepsilon$ | ε |
| $\mathcal{C} \left\{ \right\}$ | $v_1 + v_2 : \varepsilon$ | $\Gamma \left[\right]$ | ε | ε |

Table 11: Executing Code on CAMPL (Continued from Table 10)