Abstract machine for MPL (AMPL)

June 17, 2017

1 Abstract Machine for MPL (AMPL)

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

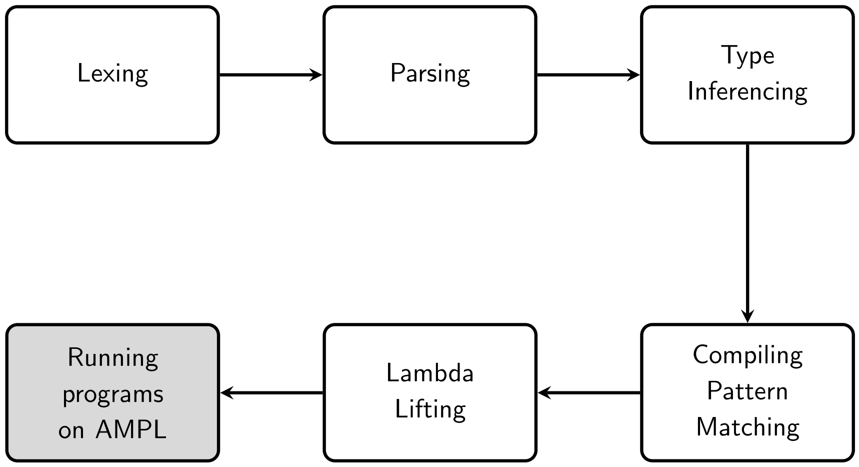

Figure 1: Compilation Stages

1.1 Introduction to Abstract Machines

Abstract machines are useful tools when developing 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 using an abstract machine is the ease of porting the language to different hardwares. Using an abstract machine one essentially reduces the problem of porting the programming language to the problem of porting the abstract machines. This is an easier task as the abstract machines are simpler and smaller than the programming language. This is the compilation philosophy that the programming language **Java** uses with its abstract machine **JVM**. Some examples of famous *abstract machines* are as follows:

- Landin's SECD Machine
- Modern-SEC Machine

- Categorial Abstract Machine (CAM)
- Krivine's Machine
- Three Instruction Machine (TIM)
- Warren's Abstract Machine (WAM)
- Java Virual Machine (JVM)

1.2 Introduction to AMPL

AMPL is the runtime of MPL. It is the *abstract machine* which runs an MPL program. MPL is conceptually separated into two levels, namely **Sequential MPL** and **Concurrent MPL**. This conceptual separation is also seen in the design of AMPL. AMPL can be thought to consist of two machines:

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

This conceptual separation results in a modular description of AMPL. The advantage of this approach is that the design of either of the AMPL components can be changed 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, *modern-SEC* machine doesn't have constructs to which *data types* and *codata types* can be compiled. SAMPL augments the basic *modern-SEC* machine with *case* and *constructor* constructs to compile data types and *record* and *destructor* constructs to compile codata types. SAMPL also adds constructs for basic functions like *addition*, *multiplication*, *equality testing* etc which are built into the language. The *SAMPL* constructs are strictly evaluated except the constructs for codata types (*record* and *destructor*) which are lazily evaluated.

2.1 SAMPL Commands

The table provides a list of *SAMPL* commands with their brief description.

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

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 . This conversion from Sequential MPL code to SAMPL commands happens via the *Core MPL(CMPL)* stage. The translation scheme has been described below. $[\![]\!]_v$ signifies the compilation of a *CMPL* construct 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 the variable names in the body of the *MPL* construct with a position number to indicate which variable was intended to be used.

$$
\begin{aligned}
[\![\mathsf{record}\{D_1 : t_1, ...D_n : t_n)\}]\!]_v &= \mathsf{Rec}[[\![t_1]\!]_v\mathsf{Ret}, ..., [\![t_n]\!]_v\mathsf{Ret}] && (1)\\
[\![(D_i(x, t_1, ..., t_n)]\!]_v &= [\![t_n]\!]_v...[\![t_1]\!]_v\ [\![x]\!]_v\mathsf{Dest}(i,n) && (2)\\
[\![\mathsf{Cons}_i(t_1, .., t_n)]\!]_v &= [\![t_n]\!]_v...[\![t_1]\!]_v\ \mathsf{Cons}(i,n) && (3)\\
[\![\mathsf{case}\ t\ \{\mathsf{Cons_i}\ x_{1_i}, ..., x_{n_i} \mapsto t_i\}_{i\in[1,m]}\ ]\!]_v &= [\![t]\!]_v\mathsf{Case}[[\![t_i]\!]_{x_1,...,x_{n_i},v}\ \mathsf{Ret}]_{i\in[1,m]} && (4)\\
[\![x]\!]_v &= \mathsf{Access}(n) \quad \text{where } n = \mathsf{index}\ v\ x && (5)\\
[\![a \text{ op } b]\!]_v &= [\![b]\!]_v\ [\![a]\!]_v\mathsf{Op} && (6)\\
[\![k]\!]_v &= \mathsf{Const}_T(k) && (7)
\end{aligned}
$$

Equation 1 describes the compilation scheme for the record construct. D_i as the i^{th} destructor and t_i is the corresponding term for that destructor.

Equation 2 describes the compilation scheme for a destructor construct represented by D_i. The

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

Table 1: Machine Transitions for the SAMPL

first argument to the D_i here is the record to be destructed and the subsequent arguments are the arguments of D_i.

Equation 3 describes the compilation of constructor represented by $Cons_i$. The terms $t_1, \ldots, t_n$ 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 of constructor number and the number of arguments that the constructor takes.

Equation 4 describes the compilation of a case construct. A few things worth noting in the compilation of case constructs are as following:

- The term t_i corresponding to the $Cons_i$ is compiled in the context enhanced with the arguments of $Cons_i$.
- *Ret* 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.

Equation 5 describes 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.

Equation 6 describes the compilation of infix functions. These functions are converted to their *postfix* function 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 *environment* and *stack* need to be introduced in order to discuss the working of the machine. Both *environment and stack* are *LIFO* data structures. In our implementation they have been implemneted using *lists*.

Environment keeps track of what variables should or shouldn't be visible in the code at a given point and is thus a transient data structure meaning that it changes as one traverses through the code. *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). Excecution of SAMPL code means starting with a initial state, changing states recursively based on the transition table for SAMPL described in Table 1 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 very easily implementable in Haksell using its pattern matching feature.

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

The initial state of the machine is formed by putting the compiled code as the first argument of the triple and intialising the *enviornment* and *stack* as empty lists.

$$\text{Final State : } ([\,],\ [\,],\ S)$$

The final state is reached the code list and the environment is empty. The top of the stack contains the output of the code.

In Table 1, $\mathsf{clos}(c, e)$ denotes closure of code c with environment e and $e(n)$ is the n^{th}-element of the environment.

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* were also used to represent the state of *Sequential MPL* programs and have the same meaning here.

- *Stack* holds the intermediate value and the final result when executing a process.
- *Environment* is used to determined which variables are visible at a given point of the concurrent program.
- *Code* is the list of Concurrent AMPL commands corresponding to the MPL program.

Translation is an additonal structure used in describing the state of a *Concurrent MPL* program. This additonal structure is required because concurrent MPL programs also have channels which the sequential MPL programs don't have. *Translation* converts the local channel name into global channel names. This is required because a channel connecting two processes may be named differently inside the individual processes. In order to ensure that the two channels are referring to the same channel, there needs to be a mapping/table that translates the local channel name to global channel names.

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

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 *enviornment* and *stack* which are the first and the third arguments respectively, as empty lists.

$$\text{Final State : } (S, [\,], [\,], [\,])$$

The final state is reached when the code list, the environment and the translation is empty. The top of the stack contains the output of the code.

CAMPL consists of 2 main components:

- **Channel Mananger**, $\mathcal{C}$
- **Process Manager**, $\mathcal{P}$

3.1 Channel Mananger

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 Mananger

Process Manager is a set of processes. **Process Manager** 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-determenenstically. In practise one may want to execute this non-determenenstically selected process not just by one step but for mutliple steps or until it has a concurrent action before interrupting it.

3.3 Interaction between Process Manager and Channel Manager

The interaction between **Process Manager** and **Channel Manager** can be understood with the help of below example:

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 maybe 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 supsended process is attached to one queue of the channel and a value is

get α; C	get a value on channel α
put α; C	put a value on channel α
split α into $(\alpha_1, \alpha_2); C$	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$\{C_1, .., C_n\}$	the cases on receiving a "handle"
close $\alpha; C$	closing a channel
halt α	halting process attached to a single channel
plug $[\alpha_1, ..., \alpha_n]$ $\Gamma_1.C_1$ $\Gamma_2.C_2$	two processes to communicate on n channels
run t $\langle$process$\rangle$	runs a process with local channel to caller channel translation t
id $\alpha = \beta$	identifying channels.

Table 2: Basic concurrent commands

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 intreaction 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 one starts with a look at the CAMPL commands. Corresponding to every concurrent MPL construct, there is one CAMPL command. Table 2 lists the concurrent commands with a brief description of each. The execution of CAMPL commands have been studied as the actions of the two components of the CAMPL, namely **Process Manager Actions** (see section 3.4.1) and **Channel Manager Actions** (see section 3.4.2).

3.4.1 Process Manager's Actions

This section defines one execution step corresponding to the various 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 **Transtion Table/Process Execution Steps** for CAMPL (described in Table 3) have columns for both **Process Manager** and **Channel Manager** in the *before* and *after* column. An explanation of the **Process Manager's** actions corresponding to the various CAMPL commands has been 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 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" – which is a protocol constructor – 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.

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 4 shows the communication of values. Row 2 shows the communication of the 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

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

Table 3: Process execution steps (α with input polarity)

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

Table 4: Channel manager actions

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 4 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 4.