Notes on AMPL, an abstract concurrent machine for MPL

J.R.B. Cockett *
Department of Computer Science, University of Calgary,
Calgary, T2N 1N4, Alberta, Canada

May 4, 2020

1 Introduction

These notes describe a basic abstract machine, AMPL, for concurrent process using a message passing paradigm. The semantics for message passing which it is designed to support is described in [**?**] and this semantics is realized in a programming language called MPL (for Message Passing Logic). Pure MPL is a strongly typed language which not only has a Curry-Howard style correspondence between proofs and programs and a categorical semantics (in linear actegories) but also comes with guarantees of deadlock and livelock freedom. However, the abstract machine which is described below is untyped and, thus, can be used to support a broader class of message passing language: in particular, it can be used to support a weakly typed version of MPL which not only allows arbitrary connection topologies (the strong type system allows only acyclic and connected systems) but also permits non-determinism so that the scheduler becomes important in determining the behaviour of programs.

Pure MPL can be distributed amongst different machines and the order of evaluation can be highly dependent on the scheduling algorithm employed – although, of course, the semantics of the process which is implemented is not dependent on the scheduler. There is some debate concerning what constitutes a concurrent language and what does not: some would argue that, as strongly typed MPL is deterministic, it is a "parallel" rather than "concurrent" language. Here we take the broader view of "concurrent language" which includes this sort of schedule independent parallelism.

For those who do insist on true concurrency we offer two inducements. First weakly typed MPL does exhibits nondeterministic behaviour (many other the hallmarks of a concurrent language) as the behaviour of these processes *is* highly dependent on scheduling. Thus pure MPL is merely isolating, by a more demanding type discipline, a class of "well-behaved" concurrent programs from the weakly typed system. Second, we discuss a "race" construct which allows actions to be dependent on the order in which the inputs are received by a process: this increases the expressive power of the language considerably.

While we shall present the concurrent abstract machine for MPL in a particular style, of course, it need not be implemented in that style. Indeed it can be greatly optimized and elaborated for actual implementations in a number of ways. In addition, these machines must in practice be implemented in a "service environment" which allows access to real physical devices and communication ports.

*Partially supported by NSERC, Canada.

2 The machine

The abstract machine has two almost distinct components: these components are for implementing respectively the concurrent and the sequential actions. One can essentially use *any* sequential machine with the concurrent machine we specify. Here we have chosen a sequential machine which is well-suited to a functional style language which uses both inductive and coinductive data. This sequential machine has three components a stack of current terms and continuations, S, an environment, E, and a code stream C. The code stream most usually is implemented as a pointer to the current instruction although here it is abstractly represented as a list. To this basic structure, we add some message passing and channel handling primitives and a **channel translation**, t, which translates local channel names into global channel names. So we get a STEC machine!

Below we shall first concentrate on the concurrent aspects of the machine as these are the novel components. We refer to the whole machine as AMPL – the Abstract machine for MPL. The concurrent part has two main components: a **channel manager**, $\mathcal{C}$, and a **process manager**, $\mathcal{P}$, both are abstractly sets in that they are regarded as being unordered. The channel manager consist of a set of pairs consisting of a channel name and a communication queue: we shall write as $\mathcal{C}\{(\alpha, q' \mid q)\}$ to indicate that channel α has the communication queue $q' \mid q$ associated with it. A **communication queue** has two parts: the output polarity queue, q', and the input polarity queue q. Each channel associated to a process is either an input or output polarity channel: the process writes its communication for an input polarity channel onto the input queue of that channel and for an output polarity channel on the output queue. We add a communication, x, to the back of the input queue by writing $q'{:}x$ and we shall also use this as a pattern to indicate that x is the last item on that queue. Similarly on the output queue we shall add a communication item, x to the front of the queue $x{:}q$. To access items on queues we shall use pattern matching: for example to access the first-in item, x, of the input queue of a channel – which is the last item as we are writing it – we shall use the pattern $q' : x$. We shall write the empty queue as ε. Note that it is the two first-in items x and y in a communication queue $q'{:}y \mid x{:}q$ which are subject to communication actions (see below).

Similarly for the process manager we shall write $\mathcal{P}\{(S, t, E, C)\}$ to indicate that we have selected a process which we plan to advance by an execution step – the $\mathcal{P}$ always stands for the non-selected processes. The machine is arranged so that all processes held by the processes manager can be executed. When a process is suspended (for example because it requires an input) it is moved to the channel manager and attached to the channel(s) on which it is expecting a communication action. The machine does not name processes, rather it names communication channels. A process state (S, t, E, C) consists of S a stack of terms and continuations, a channel translation from local channel names to global channel names t (this also holds the polarity of the channel –which we discuss below), an environment which holds values of variables, thus $E\{x := t\}$ – much as above – is read as meaning that looking up x in the environment with give t, and a code pointer (more below).

The machine works by non-deterministically choosing a process to execute for a step. Of course, in practice one may want to exhaust the execution of a process until it has a concurrent action or at least to run it for a large number of steps before interrupting it. However, the machine – as specified – does not assume any particular execution schedule (and, indeed, in the non-deterministic semantics with races the evaluation schedule can change the outcome of evaluation).

The communication manager not only keeps the state of the channels in its communication queues, but also has some associated execution actions which result from communication on the channels. For example, when a process, in order to continue, needs to "get" a value on a channel it cannot execute any further until the process it is communicating with has "put" a value on that channel. The order in which these actions, the "put" and the "get" occur does not matter. When

get α; C	get a value on channel α (in the π-calculus $\alpha?[\_]$)
put α; C	put a value on channel α (in the π-calculus $\alpha![\_]$)
split α into $(\alpha_1, \alpha_2); C$	split channel α into two (new) channels
fork α as α_1 with $\Gamma_1 \to C_1$ α_2 with $\Gamma_2 \to C_2$	forking on a channel into two distinct processes
plug $[\alpha_1, ..., \alpha_n]$ $\Gamma_1 \to C_1$ $\Gamma_2 \to C_2$	two processes to communicate on n channels
hput α $n; C$	put a "handle" on channel α
hcase$\{C_1, .., C_n\}$	the cases on receiving a "handle"
run t $\langle$`process`$\rangle$	runs a process with local channel to caller channel translation t
close $\alpha; C$	closing a channel
halt α	halting process attached to a single channel
id $\alpha = \beta$	identifying channels
race $\alpha_1 \to C_1$ $\alpha_2 \to C_2$	racing channels α and β

Table 1: Basic concurrent commands

a process wishes to "get" a value on a channel it is immediately suspended by attaching it to the appropriate communication queue for that channel. It then waits on that queue until a value is placed on the opposing queue for that channel. When a value is put onto the queue for that channel – or if it is already there – then a communication step can take place: in effect the step brings the process back to life by returning it to the process manager filled with the communicated value.

Similarly, if a process needs to "fork" a channel to continue it needs the corresponding "splitting" action to be performed by the process with which it is communicating. Thus, when a process wants to "fork" it is suspended and placed on the appropriate queue for that channel. When a corresponding splitting action is made on the channel the communication step produces two new processes communicating on two new channels and adds these to the process manager.

In general, all communication actions come in pairs (put/get, split/fork, hput/hcase, close/halt, etc.) which reflect the fact the each of the basic components of protocols (concurrent types) must come in an input and output "polarized" form whose reaction of a communicated datum against a suspension results in the communication. We refer to a protocol as being negative when it will results in a suspension on the channel and positive if it results in depositing a datum on the channel. This is important when we come to describe races as one can only race channels which are negatively typed (i.e. that are expecting input).

3 Concurrent commands

To illustrate the concurrent aspects of the machine we will start by exploring the concurrent commands. The basic machine instructions for communication primitives are displayed in Table 1. The actions which result from these commands and the state of the machine is described below.

Two basic basic execution steps are distinguished: the process execution manager actions (see Table 2) and the channel manager actions (see Table 3). The machine is written so that these can be performed in any order: in a sequential machine the order will be determined by a scheduler. With a strongly typed MPL program the effect of the program will not be altered by the scheduler (unless

it fails to execute an action). The channel manager actions take place whenever two complementary first-in communication actions are juxtaposed (e.g. get against a put, or a split against a fork, a handle, an hput against a handler, hcase) in a channel communication queue. It is not unnatural to want to perform these channel manager actions as the communications are created on the queues of the communication manager by the process of the process manager. However, it makes for a more economical presentation and a more flexible implementation to express the process managers actions and the channel managers actions independently.

3.1 Process manager actions

An explanation of the process manager actions is:

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. Normally a get α command will immediately be followed by a load x which will make the value passed available in the sequential environment of the receiver. Similarly a put α command will be preceded by code which produces a value from the sequential machine.

Split: the split instruction says that we should split the channel α into two channels (α_1 and α_2). We therefore add to the appropriate queue notification of the two "global" channels into which we will split the channel. This means we must choose two new channel names (here β_1 and β_2), and must remember the translation from the local channel names, $t[\beta_1/\alpha_1, \beta_2/\alpha_2]$ then we just continue with the code.

Fork: 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 ... of course, 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.

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

Table 2: Process execution steps (α with input polarity)

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. The command alters the (global) translation of the channels by indicating to the channel manager that the translation of α should be set 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 ... or, indeed when programming an identity map.

Race: This allows the program to "race" one or more channels which have a negative protocol (α and β) and depending on which one gets an input datum first to initiate the corresponding code (while removing the code from the loosing channels).

3.2 Channel manager actions

The channel managers is responsible for making sure the communications actually happen. Thus, if a process passes a value along a channel it is first held by the channel manager in a queue for that channel. A queued action can be completed when a complementary action is made on the channel. In each pair of complementary actions one will cause the process to be suspended (or go to sleep) until its complementary action appears (if it is already there the sleep will be for a short time!). The one action which does not follow this prescription in an obvious way is the Id action: an identity action is always a last action in a process, thus the process finishes after such a command. Its effect is to amalgamate the channels, so that subsequent communication is direct along the resulting channel.

Communication of values or a handle: When a value/handle is waiting on a channel and there

$\mathcal{C}\{(\beta, q{:}v \mid \mathsf{g}(s,t,e,c))\}$	$\mathcal{P}$	$\mathcal{C}\{(\beta, q \mid \varepsilon)\}$	$\mathcal{P}\{(v{:}s,t,e,c)\}$
$\mathcal{C}\{(t(\beta), q{:}\mathsf{h}(i) \mid (s,t,e,\mathsf{hc}\ \{c_j\}))\}$	$\mathcal{P}$	$\mathcal{C}\{(\beta, q \mid \varepsilon)\}$	$\mathcal{P}\{(s,t,e,c_i)\}$
$\mathcal{C}\{(\beta, \langle\beta_1,\beta_2\rangle \mid \left[t,e,\begin{smallmatrix}\alpha_1/\Gamma_1.c_1\\ \alpha_2/\Gamma_2.c_2\end{smallmatrix}\right]\}$	$\mathcal{P}$	$\mathcal{C}\{\}$	$\mathcal{P}\left\{\begin{smallmatrix}([], t_{\Gamma_1}[\beta_1/\alpha_1], e, c_1, []),\\ ([], t_{\Gamma_2}[\beta_2/\alpha_2], e, c_2, [])\end{smallmatrix}\right\}$
$\mathcal{C}\{(\beta, \mathsf{close} \mid \mathsf{halt})\}$	$\mathcal{P}$	$\mathcal{C}\{\}$	$\mathcal{P}$
$\mathcal{C}\{(\beta, q \mid \mathsf{id}\ \beta=\alpha), (\alpha, \varepsilon \mid q')\}$	$\mathcal{P}$	$\mathcal{C}\{(\alpha, q \mid q'\}\}$	$\mathcal{P}\{\}$
$\mathcal{C}\left\{\begin{smallmatrix}(t(\alpha), q'{:}a \mid \mathsf{r}([\beta],s,t,e,c){:}q),\\ t(\beta), r' \mid \mathsf{r}([\alpha],s,t,e,c') : r)\end{smallmatrix}\right\}$	P{}	$\mathcal{C}\left\{\begin{smallmatrix}(t(\alpha), q'{:}a \mid q),\\ (t(\beta), r' \mid r)\end{smallmatrix}\right\}$	$\mathcal{P}\{(s,t,e,c\}$

Table 3: Channel manager actions

is a suspended process waiting for the value then one can transmit the value and re-enable the process which was waiting.

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.

Close/Halt communication: A channel can be completely removed from the channel manger only when a close command matches a halt command.

Id $\alpha = \beta$: The channel manager on seeing the Id command at the top of a queue (it always must be last) will remove one of the two channels and amalgamate it into the other channel. To do this the other channel must have an empty queue of the opposite polarity of the command. in order to be able to perform this the channel manager must keep track of the identifications in such a manner that they are actually made! A simple way to effect this is to let the channel manager have a second translation which keeps track of the global identifications.

Race: This command is initiated when an input is received on one of the channels being raced (this is the channel which wins the race). At this stage the race command is removed from the competing channels and the code of the race of the winning channel is used.

3.3 External Services

When a process is run certain channels will be connected to the external world: these channels must then be serviced by the AMPL machine. The AMPL machine therefore must have certain builtin channels which can be used when running a process and which have certain predefined protocols. When messages are passed to these builtin channels the AMPL machine will provide a predefined service to the process.

For example consider the simplest sort of terminal: one which allows one to output an integer, read an integer, or close the terminal. This is, for example. provided by the builtin channels `intTerminal`, `intTerminal1`, and `intTerminal2` . When a process is run which is connected to a builtin terminal, the terminal is brought up and the process then runs and communicates with the terminal. These terminals might have protocol:

```
protocol IntTerminal => C =
                    IntTerminalGet:: Get(C | Int) => C
                    IntTerminalPut:: Put(C | Int) =>  C
                    IntTerminalClose:: Bot => C
```

The AMPL machine while scheduling the channel communications determines whether the process has connected to a service which AMPL should provide (negative channel numbers). If such a request is detected then AMPL services the request by adjusting the channel in the channel manager with what is mandated by the service.

For example if $(-1, \mathsf{Cint}\ 27 : \mathsf{h}(2) \mid \varepsilon)$ is the state of a builtin channel "-1", which is an integer terminal, then the second handler is being called and AMPL should output "27" to the terminal. On the other hand if $(-1, \mathsf{h}(1) \mid \varepsilon)$ is encountered then AMPL must read the next integer from the terminal and put it on the queue. If the user types "33" then the channel is modified to $(-1, \varepsilon \mid \mathsf{Clnt}(33))$.

4 Sequential AMPL machine

This machine is a modification of the modern SECD machine, which is, in turn, a simplification of Landin's original SECD machine. It implements a by-value reduction of terms using data and lazy codata implemented by "records". It requires that one compiles into a simple assembly language for MPL: called SMPL-code. When the code is run it produces essentially a weak head normal form (if it exists) of the program. The main modifications which distinguish it from the modern SECD machine is that it provides a direct support for both data and codata. In particular, higher-order functions are viewed as a special case of codata. It employs a pleasingly simple instruction set, and has, following the modern SECD machine, no dump.

The SMPL machine is a state machine. The state is a triple consisting of:

- A **code** pointer, c, which points to the current instruction.
- An **environment**, e, which holds the values of variables.
- A **stack** s holding both intermediate results and continuations.

One must augment the basic SMPL machine with builtin operations (addition, multiplication, etc.). For arithmetic we add "constants" for basic types (integers, characters, floats, ...) and instructions for basic operations (addition, multiplication, comparisons, etc.). The resulting instructions for a SMPL machine are:

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

The translation is:

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

The tansitions for the sequential SMPL Machine are:

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	$(n \leq m) : s$
...					

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

The machine is started with a code pointer and the environment and stack empty:

$$\begin{aligned} \text{Code} &= c \\ \text{Environment} &= \mathsf{Nil} \\ \text{Stack} &= \mathsf{Nil} \end{aligned}$$

The final state is reached when the code stack is empty: the answer should then be sitting on the stack

$$\begin{aligned} \text{Code} &= \mathsf{Nil} \\ \text{Environment} &= \mathsf{Nil} \\ \text{Stack} &= v \qquad \text{the answer is } v! \end{aligned}$$

5 Examples

Example : As an example of an evaluation in the AMPL machine, consider the program:

```
codata C -> Exp(A,B) =  App: C, A -> B
     -- the destructors for codata can now take an argument (here A)

App (rec {App: x.x+1},2)
```

This is the program $(\lambda x.x + 1)\ 2$ written in MPL.

It translates into the following code:

```
CInt 2
Rec [ CInt 1
          Access 1
          Add
          Ret
        ]
Dest 1 1
```

Code	Env	Stack
$\mathsf{CInt}(2); \mathsf{Rec}[c]; \mathsf{Dest}\ 1\ 1$	ε	ε
$\mathsf{Rec}[c]; \mathsf{Dest}\ 1\ 1$	ε	$\mathsf{cint}\ 2 : \varepsilon$
$\mathsf{Dest}\ 1\ 1$	ε	$\mathsf{rec}([c], \varepsilon) : \mathsf{cint}\ 2 : \varepsilon$
$\mathsf{CInt}\ 1; \mathsf{Access}\ 1; \mathsf{Add}; \mathsf{Ret}$	$2 : \varepsilon$	$\mathsf{clo}(\varepsilon, \varepsilon) : \varepsilon$
$\mathsf{Access}(1) : \mathsf{Add} : \mathsf{Ret}$	$2 : \varepsilon$	$\mathsf{cint}\ 1 : \mathsf{clo}(\varepsilon, \varepsilon) : \varepsilon$
$\mathsf{Add} : \mathsf{Ret}$	$\mathsf{cint}\ 2 : \varepsilon$	$\mathsf{cint}\ 2 : \mathsf{cint}\ 1 : \mathsf{clo}(\varepsilon, \varepsilon) : \varepsilon$
Ret	$\mathsf{cint}\ 2 : \varepsilon$	$\mathsf{cint}\ 3 : \mathsf{Clo}(\varepsilon, \varepsilon) : \varepsilon$
ε	ε	$\mathsf{cint}\ 3 : \varepsilon$

where $c := \mathsf{CInt}\ 1; \mathsf{Access}\ 1; \mathsf{Add}; \mathsf{Ret}$.

Example 2: The following example talks to a channel of type `IntTerminal` and is run connected to a builtin channel `intTerminal1` which allows integers to be entered or output on a terminal. The program then reads in two numbers from the terminal and outputs the sum to the terminal.

```
adder ::  |  =>  IntTerminal
adder | =>  ch =
         hput IntTerminalGet on ch
         get x on ch
         hput IntTerminalGet on ch
         get y on ch
         hput IntTerminalPut
         put x+y on ch
         hput IntTerminalClose on  ch
         halt ch

run adder | => intTerminal1
```

This gets translated into machine code as:

```
hput -1 1
-- -1 is a built in channel number which connects to an integer terminal
--  the 1 corresponds to the first handler "IntTerminalGet"
get -1                    -- this loads the read value onto the stack
store                     -- pushes into the environment
hput -1 1
get -1
store
access 2              -- loads first number read onto stack
```

```
access 1              -- loads second number read onto stack
add                      -- adds the two top elements of the stack
hput -1 2             -- calls the second handler, "IntTerminalPut" to output an integer
put -1
hput -1 3             -- calls the third handler, "IntTerminalClose" to output an integer
halt -1
```

Example 3: — not yet updated! The following example is an example of plugging processes together:

```
adder'::   |  Get(int,Put(Int,Top)) => Get(Int,Get(Int,Put(Int,Top)))
  = | ch1 => ch do
       get x on ch
       get y on ch
       put x+y on ch1
       get z on ch1
       close ch1
       put z+z on ch
       halt ch

 inc'::  |  =>  Get(Int,Put(Int,Top))
     =  |  => alpha do
       get x on alpha
       put x+1 on alpha
       halt alpha

proc = adder  | => ch2 do
          plug
                alpha = ch1
                inc' ( | => alpha)
                adder'( | ch1 => ch2)
```

Example 4: Memory Cell:

```
protocol MEM (A) => P =
  PUT :: Put (A|P) => P
  GET :: Get (A|P) => P
  CLS :: TopBot    => P

  proc memory :: Int | MEM(Int) =>
    = x | ch =>
    do
      hcase ch of
        PUT
          do get y on ch
               memory(y|ch => )
        GET
          do put x on  ch
               memory(x|ch => )
        CLS
          do halt ch
```

Example 5: Parallel OR:

```
proc pOR::  | Put(Bool,Top), Put(Bool,Top) =>  Put(Bool,Top)
        =   | ch1, ch2 => ch4 do
              plug
                 sOR( | ch1, ch2 => ch3 )
                 do
                      split ch3 to (ch31,ch32) ->
                          close ch32
                          ch31 = ch4


proc sOR:: Put(Bool,Top), Put(Bool,Top) =>  Put(Bool,Top) (*) Top
           = | ch1,  ch2 => ch4 do
                 get b on ch1; close ch1
                 case b of
                     True ->
                          fork ch4 as
                              ch41 -> put True on ch41
                              ch42 -> get _ on ch2; close ch2; halt ch42
                     False ->
                          get b' on ch2; close ch2
                          case b' of
                              True -> fork ch4 as
                                  ch41 -> action( | ch3 => ch41)
                                  ch42 -> halt ch42
                              False -> fork ch4 as
                                  ch41 -> noaction( | ch3 => ch41)
                                  ch42 -> halt ch42
```