REDESIGNING THE ABSTRACT MACHINE FOR CAMPL.

STUDENT: JARED PON. SUPERVISOR: DR. ROBIN COCKETT

Abstract. This document discusses a redesign of the abstract machine for CaMPL designed by [6]. We describe missing details of [6] which lead to unclear and inefficient implementations, and redesign the channel manager to save runtime channel polarity checks and simplify the number of cases needed for state transitions. We also include instructions for permitting constant stack space of programs via tail recursion.

1. Background

The work of [6] designed and implemented the programming language CaMPL with both the front end and the back end. In this work, we will be improving the abstract machine written by [6] for CaMPL.

Often, abstract machines [4] are used for back ends of programming languages because the operational semantics of how the source language in question works and how the actual hardware works are often very different. Hence, during the compilation process, having an abstract machine provides a way to promote correctness by providing a smaller step between compiling the source language to the actual hardware. There are many examples of abstract machines for functional languages today – see [3, 7]

In this document we aim to discuss some improvements of [6]. The first shortcoming of the original abstract machine was that it was completely sequential i.e., while [6] claimed CaMPL was a concurrent programming language, its implementation was completely sequential and could only execute one instruction at a time. As such, we desired to make CaMPL a concurrent language as claimed, which also would permit the addition of the `race` command for increasing the expressivity of the language for non-deterministic computations.

In the reimplementation of the abstract machine to be concurrent, it was observed that the original presentation with its numerous unspecified bits unsurprisingly presented new implementation challenges. In particular, early attempts which followed the presentation of [6] required mutexes over the entire channel manager which significantly reduced the concurrency of the system.

So, in this work, we discuss an alternative presentation which pays particular attention to how one would actually implement a concurrent implementation on a real machine. We will also discuss other "cleanliness" improvements such as removing channel polarity information at run time and using a consistent mechanism for accessing actual function / process parameters.

2. Abstract Machine

In this section, we describe the abstract machine for which a CaMPL program runs on. This amounts to defining a an intermediate *core CaMPL* language with just the "important

parts" of the original CaMPL AST, defining a translation from the core CaMPL language to an intermedate *CaMPL machine code*, and defining one step reductions of the *CaMPL machine code* with the overall state of the machine.

2.1. **Core CaMPL Language and CaMPL Bytecode.** In this section, we define the core CaMPL language. This language will closely resemble the language given in [6] with some minor adaptations.

We give the definition as a Haskell data type as follows

```
-- | A core program is a list of declarations
type CProg = [CDecl]

-- | Objects (data, codata, protocol, coprotocols) can be decayed down to
-- indices
type CObjIx = Int

-- | A type alias for names in the program
type CName = String

data Polarity
    = Input
    | Output

-- | A type alias for channels
type CCh = (String, Polarity)

-- | A declaration in the core language is either a process or a function
data CDecl
    = CProc CName [CName] [CName] [CName] [CCh] CTerm
        -- ^ Process name, higher order functions, higher order processes,
        -- sequential arguments, channels
    | CFun CName [CName] [CName] CTerm
        -- ^ Function name, higher order arguments, sequential arguments

-- | A term is the body of a process or function.
data CTerm
    = CVar CName
        -- ^ A variable access
    | CCst Int
        -- ^ A constant literal. This language may be extended to support more
        -- constant literals

    | CCase CTerm [ ([CName], CTerm) ]
        -- ^ An expression to case on, and a list of cases with variables that
        -- are bound in each respective case
    | CCons CObjIx [CTerm]
```

```
    -- ^ Constructs a constructor with an object index and a list of
    -- actual arguments.

| CDest CObjIx CTerm [CTerm]
    -- ^ Calls the destructor with an object index, a
    -- distinguished actual argument -- which should be a record,
    -- and a list of actual arguments

| CRec [ ([CName], CTerm) ]
    -- ^ Constructs a record given a list of functions.

| CCall CName [CName] [CTerm]
    -- ^ Calls a function with a list of higher order actual arguments and
    -- a list of actual arguments

| CHCall CName [CTerm]
    -- ^ Calls a higher order function with a list of arguments

| CRun CName [CName] [CName] [CTerm] [CCh]
    -- ^ Calls a process name with higher order functions, higher order
    -- processes, sequential arguments, channels

| CHRun CName [CTerm] [CCh]
    -- ^ A higher order proces call

| CGet CName CCh CTerm
    -- ^ Binds a result to the first argument from a channel (second
    -- argument), and continues as the last argument

| CPut CTerm CCh CTerm
    -- ^ Evaluates the first argument and sends it across a channel (second
    -- argument), and continues as the last argument

| CSplit CCh CCh CCh CTerm
    -- ^ Channel to split, distinct channels to split into, continues as
    -- last argument

| CFork CCh CForkPhrase CForkPhrase
    -- ^ Channel to fork and phrases.

| CHCase CCh [CTerm]
    -- ^ Channel to hcase on and case options.

| CHPut CName CCh CTerm
```

```
        -- ^ protocol on channel and continue with term

    | CChEq CCh CCh
        -- ^ Channels to regard as equal

    | CClose CCh CTerm
        -- ^ Closes a channel and continue with a term

    | CHalt CCh
        -- ^ Closes a channel and halts the process

    | CRace (CCh, CTerm) (CCh, CTerm)
        -- ^ A binary race between two channels

    | CPlug CName (CTerm, CTerm)
        -- ^ A channel for the two terms to communicate on

    | CAssign CName CTerm CTerm
        -- ^ Assigns a variable to a term, and continues as the next term.

-- | Phrase for a fork. The tuple projections are the new binder for the
-- channel, the context to take with, and the term to continue as.
type CForkPhrase = (CCh, [CCh], CTerm)
```

We will omit the rules that make such terms "well-formed," but an interested reader can adapt the formalities given in [2, 8]. Moreover, the `race` and `plug` commands are binary, but can be extended to race and plug arbitrarily many processes.

We give the definition of the CaMPL bytecode language for which we wish to translate core CaMPL to. Instead of giving a Hakell data type, we informally describe the instructions this machine has in the following bullet points, and in later sections we will more precisely define how these bytecode instructions will execute.

To better understand the instructions of this machine, keep in mind that when executing code we have a stack of intermediate values and an environment which maps the local names to actual values.

- `access` n: accesses the nth value in the environment and puts that at the top of the stack.
- `ret`: return the top stack value and jump to the continuation below the top value on the stack.
- `call` *code* n: performs a function call to *code* with n arguments.
- `tailcall` *code* n: performs a tail call to *code* with n arguments.
- `hcall` i n: performs a function call to the ith value in the environment with n arguments.
- `htailcall` i n: performs a function call to the ith value in the environment with n arguments.

We include instructions for working with primitive data types.

- `cst` k: pushes a constant k onto the stack.
- `add`: pops the top two values from the stack, adds them together, and pushes the result onto the top of the stack.
- `mul`: pops the top two values from the stack, multiplies them together, and pushes the result onto the top of the stack.
- etc.

We include some instructions for working with data types given in the source language.

- `cons` i n: pushes the ith constructor onto the stack with top n arguments on the stack.
- `case` $[c_1, \ldots, c_n]$: pops the top of the stack (which should be a constructor) and chooses the corresponding code c_i to execute based the top of the stack. Moreover, when c_i is executed the corresponding arguments are put in the environment.

We include instructions for working with codata types given in the source language.

- `rec` $[c_1, \ldots, c_n]$: creates a record on the top of the stack with the current environment.
- `dest` i n: destructs a record at the top of the stack by choosing the ith function closure with the remaining top n elements on the stack put in the environment as the arguments.

We include instructions for working concurrency primitives provided by the source language.

- `get` ch: waits and gets a value from a channel ch, and pushes the value onto the top of the stack.
- `put` ch: puts the value at the top of the stack on the channel ch.
- `split` ch: splits the channel ch into two new channels.
- `close` ch: puts a close command on the channel ch.
- `halt` ch: puts a halt command on the channel ch and terminates this current process.
- `fork` ch `as` $\{ch_1 \to c_1; ch_2 \to c_2\}$: suspends this process on ch with this current fork instruction waiting for a corresponding `split` instruction. When such corresponding `split` instruction arrives, this will split into two processes running c_1 and c_2.
- ch_1 `|=|` ch_2: identifies ch_1 and ch_2 as the same channel.
- `hput` i ch: puts the ith protocol handle on channel.
- `hcase` ch $[c_1, \ldots, c_n]$: suspends this process on ch, waiting for a corresponding protocol handle and executing the corresponding code c_i when such protocol arrives.
- `plug` c_1 c_2: spawns a new channel and executes c_1, c_2 in parallel.
- `race` (ch_1, c_1) (ch_2, c_2): races channels ch_1, ch_2 against each other. If ch_1 wins the race, then c_1 is executed and likewise for ch_2 and c_2. Ties are broken arbitrarily.

We now give the translation from core CaMPL to the bytecode language. Throughout this translation, we maintain the invariant for the calling convention to put arguments in the correct order on the stack, and results of an expression are returned at the top of the stack.

The translation is given by a recursive procedure with an environment for variables (the variables which may be accessed at a given node in the core language AST) and an

environment for channels (channels which may be accessed at a given node in the core language AST) which maps `CTerm` to a list of code as previously defined. We denote appending lists by $|$, and if lst is a list we denote the reverse list by $rev\ lst$. We denote this translation with $[\![\cdot]\!]_e$ where e is the environment for variables. Given a name v for a variable, we denote the index of v in the variable environment e by e_v, and similarly, given a channel ch we denote the index of ch in the channel environment by ec_{ch}.

The translation for a top level function declarations puts the arguments all in the context, translates the function body, then returns. When compiling all top level expressions, one would want to associate the current function name with this code but we decline to describe this implementation detail.

$$[\![\texttt{CFun}\ \ f\ \ gs\ \ as\ \ b]\!]_e = [\![b]\!]_{rev\ gs|rev\ as|e} \mid [\texttt{ret}]$$

Toplevel declarations of processes are translated similarly to top level functions with a few differences. Since processes have higher order processes and channels in the context, we must put those in the context as well. Moreover, process calls are all tail calls and do not need to return to a calling context.

$$[\![\texttt{CProc}\ \ f\ \ gs\ \ hs\ \ as\ \ cs\ \ b]\!]_e = [\![b]\!]_{rev\ hs|rev\ gs|rev\ as|rev\ cs|e}$$

When we have a variable named v in the core CaMPL language, at runtime this variable will be represented as a stack offset. Hence, we should look up in our variable context e which index we should access in order to access the variable.

$$[\![\texttt{CVar}\ \ v]\!]_e = [\texttt{access}\ \ e_v]$$

Constants in the core CaMPL language corresponds to constants in the bytecode language.

$$[\![\texttt{CCst}\ \ n]\!]_e = [\texttt{cst}\ \ n]$$

For constructors, we compile the arguments in reverse order to maintain the invariant that arguments are put in reverse order on the stack. Then, we follow all the arguments with a `cons` instruction to construct the constructor.

$$[\![\texttt{CCons}\ \ i\ \ [b_i]_{i=1\ldots q}]\!]_e = [\![b_q]\!]_e \mid \cdots \mid [\![b_1]\!]_e \mid [\texttt{cons}\ \ i\ \ q]$$

For the case expression, we compile the expression to be "cased on" first (i.e., we are essentially evaluating that expression first), then we follow that a case expression to discriminate the constructor. Again, note that in the translation of each b_i we need to reverse the argument lists.

$$[\![\texttt{CCase}\ \ \texttt{c}\ \ [(as_i, b_i)]_{i=1\ldots q}]\!]_e = [\![c]\!]_e \mid [\texttt{case}\ \ [[\![b_i]\!]_{rev\ \ as_i|e} \mid \texttt{ret}]_{i=1\ldots q}\ \ ]$$

A record is simply translated into the `rec` command. Again, note that the translation of the projections of a record reverses the arguments.

$$[\![\texttt{CRec}\ \ [(as_i, b_i)]_{i=1\ldots q}]\!]_e = [\texttt{rec}\ \ [[\![b_i]\!]_{rev\ \ as_i|e} \mid \texttt{ret}]_{i=1\ldots q}\ \ ]$$

A destructor is translates its arguments in reverse order first (i.e., we essentially evaluate the destructors in reverse order), then use the `dest` instruction.

$$[\![\texttt{CDest}\ \ i\ \ b_0\ \ [b_i]_{i=1\ldots q}]\!]_e = [\![b_q]\!]_e \mid \cdots \mid [\![b_1]\!]_e \mid [\![b_0]\!]_e \mid [\texttt{dest}\ \ i\ \ q]$$

The call translates the arguments in reverse order. Note that we put the higher order arguments on the stack, and call the function with $p + q$ arguments (i.e., the number of actual arguments plus the number of higher order arguments).

$$[\![\texttt{CCall}\ \ f\ \ [g_i]_{i=1\ldots p}\ \ [b_j]_{j=1\ldots q}]\!]_e = [\![b_q]\!]_e \mid \cdots \mid [\![b_1]\!]_e \mid [\texttt{cst}\ \ g_i]_{i=p\ldots 1} \mid [\texttt{call}\ \ f\ \ (p+q)]$$

The higher order function call is translated in a similar manner to a regular function call.

$$[\![\texttt{CHCall}\ \ f\ \ [b_i]_{i=1\ldots q}]\!]_e = [\![b_q]\!]_e \mid \cdots \mid [\![b_1]\!]_e \mid [\texttt{hcall}\ \ e_f\ \ q]$$

The run command is translated similarly to the call command. Again, note that we put arguments in backwards.

$$\left[\!\!\left[\begin{array}{lll}\texttt{CRun}\ \ f & [g_i]_{i=1\ldots p} & [h_j]_{j=1\ldots q} \\ & [b_k]_{k=1\ldots r} & [c_s]_{s=1\ldots m}\end{array}\right]\!\!\right]_e = \begin{array}{l}[\texttt{access}\ \ e_{c_s}]_{s=m\ldots 1} \mid [\![b_r]\!]_e \mid \cdots \mid [\![b_1]\!]_e \mid [\texttt{access}\ \ h_j]_{j=q\ldots 1} \\ \quad \mid [\texttt{access}\ \ g_i]_{i=p\ldots 1} \mid [\texttt{tailcall}\ \ f\ \ (p+q+r+m)]\end{array}$$

The higher order run command is similar to the previous run command.

$$[\![\texttt{CHRun}\ \ f\quad [b_i]_{i=1\ldots p}\quad [c_t]_{t=1\ldots q}]\!]_e = \begin{array}{c}[\![b_p]\!]_e \mid \cdots \mid [\![b_1]\!]_e \mid \\ [\texttt{cst}\ \ c_s]_{s=q\ldots 1} \mid [\texttt{htailcall}\ \ e_f\ \ (p+q)]\end{array}$$

The get command is translated literally to its corresponding command, and we note that we extend the environment with the variable which is bound to the value given by the channel.

$$[\![\texttt{CGet}\ \ a\ \ ch\ \ b]\!]_e = [\texttt{get}\ \ e_{ch}] \mid [\![b]\!]_{[a]\mid e}$$

The put command is translated to evaluate the expression first, then put that value on the channel, then finally continue as the expression to continue with.

$$[\![\texttt{CPut}\ \ a\ \ ch\ \ b]\!]_e = [\![a]\!]_e \mid [\texttt{put}\ \ e_{ch}] \mid [\![b]\!]_e$$

The plug command translates both of the terms l and r in the environment augmented with ch. Note that in the bytecode there is no longer any information about the channel ch retained.

$$[\![\texttt{CPlug}\ \ ch\ \ (l,r)]\!]_e = [\texttt{plug}\ \ [\![l]\!]_{ch\mid e}\ \ [\![r]\!]_{ch\mid e}\ \ ]$$

The remaining translations are fairly straightforward and employ a similar intuition as the `get` and `put` translations, so we will not comment on these translations.

$$[\![\texttt{CSplit}\ \ ch\ \ ch_0\ \ ch_1\ \ b]\!]_e = [\texttt{split}\ \ e_{ch}] \mid [\![b]\!]_{[ch_1]\mid[ch_0]\mid e}$$

$$[\![\texttt{CFork}\ \ ch\ \ \begin{array}{l}(ch_0, ws_0, b_0)\\(ch_1, ws_1, b_1)\end{array}\ \ ]\!]_e = [\texttt{fork}\ \ e_{ch}\ \ \begin{array}{l}(e_{ch_0}, [\![b_0]\!]_{[ch_0]\mid e})\\(e_{ch_1}, [\![b_1]\!]_{[ch_1]\mid e})\end{array}\ \ ]$$

$$[\![\texttt{CHCase}\ \ ch\ \ [b_i]_{i=1\ldots q}]\!]_e = [\texttt{hcase}\ \ e_{ch}\ \ [[\![b_i]\!]_e]_{i=1\ldots q}\ \ ]$$

$$[\![\texttt{CHPut}\ \ ch\ \ i\ \ b]\!]_e = [\texttt{hput}\ \ e_{ch}\ \ i\ \ ] \mid [\![b]\!]_e$$

$$[\![\texttt{CChEq}\ \ ch0\ \ ch1]\!]_e = [e_{ch0}\ \ \texttt{|=|}\ \ e_{ch1}\ \ ]$$

$$[\![\texttt{CRace } [(ch_i, b_i)]_{i=1\ldots q}]\!]_e = [\texttt{race } [(e_{ch_i}, [\![b_i]\!]_e)]_{i=1\ldots q} \ ]$$

$$[\![\texttt{CClose } ch \ b]\!]_e = [\texttt{close } e_{ch} \ ] \mid [\![b]\!]_e$$

$$[\![\texttt{CHalt } ch \ ]\!]_e = [\texttt{halt } e_{ch} \ ]$$

3. Operational Semantics of Bytecode

We discuss the execution semantics of the bytecode. We first need to give the state of the system, then we describe state transitions for the system. A running program consists of a set of *processes* $\mathcal{P}$ and a set of *channels* $\mathcal{C}$. If $\mathcal{P}$ is non empty, we write $\mathcal{P}', p$ to denote choosing an arbitrary element p out of the set $\mathcal{P}$ with $\mathcal{P}'$ denoting the rest of the set $\mathcal{P}$. We use a similar notation for the channels $\mathcal{C}$.

A process is 3-tuple (c, e, s) of code c, an environment e, and a stack s. The code is a stack of the bytecode described above. The environment is a stack consisting of either built-in values, constructors, destructors, function pointers, or channels. The stack consists of intermediate values for computations, so either built-in values, closures, constructors, or records. For the code c, environment e, and stack s, we write $a : as$ to denote that a is the top element of the stack and as is the rest of the stack. Moreover, we use [] to denote an empty stack for the code c, environment e, and stack s.

A channel is a 3-tuple (α, β, qs) where α, β are channel identifiers, and qs is a queue We call qs the channel queue of α. We write $q : qs$ to denote that q is the front of the queue and qs is the rest of the queue, and we write $qs : q$ to denote that we pushed q at the back of the queue. In the set of channels, we will maintain the invariant that $(\alpha, \beta, qs) \in \mathcal{C}$ if and only if $(\beta, \alpha, qs') \in \mathcal{C}$. The queue qs will contain modified concurrent process commands.

We describe the state transitions in following subsections. We first describe the starting and termination conditions of the abstract machine, then we discuss the "sequential" steps (steps which do not modify the channel manager), and then we discuss "concurrent" steps (steps which interact with the channel manager). For the following descriptions, since the program consists of both the sets $\mathcal{P}$ and $\mathcal{C}$ to execute a step, we write the following

$$\begin{array}{cc} \mathcal{P} & \mathcal{C} \\ \Longrightarrow & \\ \mathcal{P}' & \mathcal{C}' \end{array}$$

to denote a step from the top state to the bottom state provided we may write the system as given in the top state.

3.1. Starting Condition.

We discuss the starting condition. Assuming we have a distinguished function called *main* with its associated code c (from the previous translation), the machine starts in the state

$$\{(c, [], [])\} \quad \emptyset \tag{1}$$

where we note that the environment and stack are initially empty since there are no variables currently in the context, and no intermediate instructions to happen.

When there are no state transitions that apply, the system terminates.

3.2. **Sequential State Transitions.** We discuss the sequential steps of the system which do not interact with the channels $\mathcal{C}$.

The `access` instruction accesses the ith element in the environment and pushes it to the top of the stack. We denote the indexing operation with $e[i]$.

$$\boxed{\begin{array}{cc} \mathcal{P}, & \\ (\texttt{access}\ \ i : c, e, s) & \mathcal{C} \\ \Longrightarrow & \\ \mathcal{P}, & \\ (c, e, e[i] : s) & \mathcal{C} \end{array}} \tag{2}$$

The return instruction `ret` returns the top stack value, and continues to execute with the closure below it. Note that the closure below it had its own environment to run in, and hence we run the closure in its original environment.

$$\boxed{\begin{array}{cc} \mathcal{P}, & \\ (\texttt{ret} : c, e, v : \texttt{clos}\ \ c'\ \ e' : s) & \mathcal{C} \\ \Longrightarrow & \\ \mathcal{P}, & \\ (c', e', v : s) & \mathcal{C} \end{array}} \tag{3}$$

The `call` instruction execute the code given with the top n arguments in the stack creating the new environment. Moreover, this creates a closure of the code to return after the call finishes.

$$\boxed{\begin{array}{cc} \mathcal{P}, & \\ (\texttt{call}\ \ c'\ \ n : c, e, v_1 : \cdots : v_n : s) & \mathcal{C} \\ \Longrightarrow & \\ \mathcal{P}, & \\ (c', v_n : \cdots : v_1 : [], \texttt{clos}\ \ e\ \ c : s) & \mathcal{C} \end{array}} \tag{4}$$

The `tailcall` instruction execute the code given with the top n elements in the stack forming the new environment. This differs from the regular `call` in the sense that it does not create a closure to return to since it is the last instruction to execute in a function call and hence there is no need to return to code or generate a temporary closure with an environment to return to. We remark that the check that `tailcall` $c'\ \ n : []$ is the last instruction in the code is unnecessary and should be always true from the compilation step.

$$\boxed{\begin{array}{cc} \mathcal{P}, & \\ (\texttt{tailcall}\ \ c'\ \ n : cs, e, v_1 : \cdots : v_n : s) & \mathcal{C} \\ \Longrightarrow & \\ \mathcal{P}, & \\ (c', v_n : \cdots : v_1 : [], s) & \mathcal{C} \end{array}} \tag{5}$$

The `hcall` instruction calls the code corresponding to the label in the environment. Other than that, it is similar to the `call` instruction.

$$\boxed{\begin{array}{l} \mathcal{P}, \\ (\texttt{hcall}\ \ i\ \ n : c, e, v_1 : \cdots : v_n : s) \\ \Longrightarrow \\ \mathcal{P}, \\ (c', v_n : \cdots : v_1 : [], \texttt{clos}\ \ e\ \ c : s) \end{array} \begin{array}{l} \mathcal{C} \\ \\ \\ \mathcal{C} \end{array}} \tag{6}$$

where c' is the code for $e[i]$.

The `htailcall` instruction calls the code corresponding to the label in the environment. Other than that, it is similar to the `tailcall` instruction.

$$\boxed{\begin{array}{l} \mathcal{P}, \\ (\texttt{htailcall}\ \ c'\ \ n : [], e, v_1 : \ldots v_n : s) \\ \Longrightarrow \\ \mathcal{P}, \\ (c', v_n : \cdots : v_1 : [], s) \end{array} \begin{array}{l} \mathcal{C} \\ \\ \\ \mathcal{C} \end{array}} \tag{7}$$

where c' is the code for $e[i]$.

The `cst` instruction just pushes the constant at the top of the stack.

$$\boxed{\begin{array}{l} \mathcal{P}, \\ (\texttt{cst}\ \ v : c, e, s) \\ \Longrightarrow \\ \mathcal{P}, \\ (c, e, v : s) \end{array} \begin{array}{l} \mathcal{C} \\ \\ \\ \mathcal{C} \end{array}} \tag{8}$$

The `add` instruction adds the top two argments on the stack together. Other primitive instructions should be implemented similarly.

$$\boxed{\begin{array}{l} \mathcal{P}, \\ (\texttt{add} : c, e, v_1 : v_2 : s) \\ \Longrightarrow \\ \mathcal{P}, \\ (c, e, (v_1 + v_2) : s) \end{array} \begin{array}{l} \mathcal{C} \\ \\ \\ \mathcal{C} \end{array}} \tag{9}$$

The `cons` instruction grabs the top n elements fo the stack as arguments for the constructor.

$$\boxed{\begin{array}{l} \mathcal{P}, \\ (\texttt{cons}\ \ i\ \ n : c, e, v_1 : \cdots : v_n : s) \\ \Longrightarrow \\ \mathcal{P}, \\ (c, e, \texttt{cons}\ \ i\ \ [v_n, \ldots, v_1] : s) \end{array} \begin{array}{l} \mathcal{C} \\ \\ \\ \mathcal{C} \end{array}} \tag{10}$$

The `case` instruction pops the top of the stack to choose which should be a contructor to distringuish which case expression to continue with. We will leave it as an exercise to

write a tail recursive version of this as well.

$$\begin{array}{|ll|} \hline & \mathcal{P}, \\ & (\texttt{case}\ \ [c_1, \ldots, c_n] : c, e, \texttt{cons}\ \ i\ \ [v_n, \ldots, v_1] : s) \quad \mathcal{C} \\ \Longrightarrow & \\ & \mathcal{P}, \\ & (c_i, v_n : \cdots : v_1 : e, \texttt{clos}\ \ c\ \ e : s) \quad \mathcal{C} \\ \hline \end{array} \tag{11}$$

The `rec` instruction creates a record, and pushes this record at the top of the stack with the current environment. The reason why we need to drag the current environment with the record is because in the source language, the record may access variables outside of its immediate function call.

$$\begin{array}{|ll|} \hline & \mathcal{P}, \\ & (\texttt{rec}\ \ [c_1, \ldots, c_n] : c, e, s) \quad \mathcal{C} \\ \Longrightarrow & \\ & \mathcal{P}, \\ & (c, e, \texttt{rec}\ \ [c_1, \ldots, c_n]\ \ e : s) \quad \mathcal{C} \\ \hline \end{array} \tag{12}$$

The `dest` instruction peeks at the top of the stack for a record and the n arguments, and selects the corresponding code to execute under the modified environment with arguments.

$$\begin{array}{|ll|} \hline & \mathcal{P}, \\ & (\texttt{dest}\ \ i\ \ n : c, e, \texttt{rec}\ \ [c_1, \ldots, c_n]\ \ e' : v_1 : \cdots : v_n : s) \quad \mathcal{C} \\ \Longrightarrow & \\ & \mathcal{P}, \\ & (c_i, v_n : \cdots : v_1 : e', \texttt{clos}\ \ c\ \ e : s) \quad \mathcal{C} \\ \hline \end{array} \tag{13}$$

3.3. **Process Concurrent Steps.** The `plug` command generates fresh channel identifiers α, β with channels $(\alpha, \beta, [])$ and $(\beta, \alpha, [])$ in $\mathcal{C}$ with the intention that these channels should be communicating with each other. It then terminates this current process (the `plug` command should be the last process command), and puts the processes c_1 and c_2 in $\mathcal{P}$ for execution. Note that these processes have the same environment and stack as the original process, except for augmenting the environment of c_1 with α and c_2 wih β.

$$\begin{array}{|lll|} \hline & \mathcal{P}, & \\ & (\texttt{plug}\ \ c_1\ \ c_2 : [], e, s) & \mathcal{C} \\ \Longrightarrow & & \\ & \mathcal{P}, & \mathcal{C}, \\ & (c_1, \alpha : e, s), & (\alpha, \beta, []), \\ & (c_2, \beta : e, s) & (\beta, \alpha, []) \\ \hline \end{array} \tag{14}$$

where α, β are fresh channel identifiers.

In the following state transitions, we will see how processes will communicate with each other with the channels generated by the plug instruction. We will cover this in quite some detail for the `get` and `put` instructions ((15), (16), (17)), and the remaining instructions will follow a similar idea.

3.3.1. *Get/Put State Transitions.* The `get` command makes a process lookup the channel identifier $\alpha = e[i]$ from its environment, then pushes itself onto the back of the channel queue of α. When a process does this action of pushing itself to the back of a channel queue, we say that the process suspends itself on that channel. This captures the notion of this process waiting for a corresponding input.

$$\begin{array}{lll} & \mathcal{P}, & \mathcal{C}, \\ & (\texttt{get}\ \ ch : c, e, s) & (\alpha, \beta, qs) \\ \Longrightarrow & & \\ & \mathcal{P} & \mathcal{C}, \\ & & (\alpha, \beta, qs : (c, e, s)) \end{array} \tag{15}$$

where $\alpha = e[ch]$.

The `put` command is dual to the `get` command. We see that it pushes the value at the top of its stack to the back of the corresponding channel's queue. Note that this command does not force this process to suspend – this command allows the process to send its value at the top of the stack immediately and keep taking steps.

$$\begin{array}{lll} & \mathcal{P}, & \mathcal{C}, \\ & (\texttt{put}\ \ ch : c, e, v : s) & (\alpha, \beta, qs) \\ \Longrightarrow & & \\ & \mathcal{P}, & \mathcal{C}, \\ & (c, e, s) & (\alpha, \beta, qs : v) \end{array} \tag{16}$$

where $\alpha = e[ch]$.

Of course, we need some sort of interaction between the `get` and `put` commands in the channel manager. In [6], these descriptions were provided separately and called *channel manager steps*, but here we will group such steps and later discuss trade offs for how one may actually implement these steps. When we have channel $(\alpha, \beta, v : qs)$ and $(\beta, \alpha, (c, e, s) : qs')$ for which a value v is at the front of qs from a `put` command and a process (c, e, s) is suspended from the `get` at the front of qs', we may put the process $(c, v : e, s)$ in $\mathcal{P}$ for execution and pop both of the channel queues. Notationally, this is given by the following state transition.

$$\begin{array}{lll} & & \mathcal{C}, \\ & \mathcal{P} & (\alpha, \beta, v : qs), \\ & & (\beta, \alpha, (c, e, s) : qs') \\ \Longrightarrow & & \\ & \mathcal{P}, & \mathcal{C}, \\ & (c, v : e, s) & (\alpha, \beta, qs), \\ & & (\beta, \alpha, qs') \end{array} \tag{17}$$

3.3.2. *Split/Fork State Transitions.* The `split` command in a process generates fresh channel identifiers α_1, α_2 and β_1, β_2, and puts channels $(\alpha_1, \beta_1, [])$, $(\beta_1, \alpha_1, [])$ and $(\alpha_2, \beta_2, [])$, $(\beta_2, \alpha_2, [])$ in $\mathcal{C}$. Moreover, this command puts α_1, α_2 in the environment of this current process, and puts stores the datum β_1, β_2 at the back of the channel queue qs so a later `fork` command may put β_1, β_2 in its environment. Note that we put `split` β_1 β_2 in the channel manager

instead of `split` $ch\ \ \beta_1, \beta_2$ since ch serves no purpose in the channel manager. Notationally, this is given by the following state transition.

$$\begin{array}{|ccc|}\hline \multicolumn{3}{|c|}{\begin{array}{cc} \mathcal{P}, & \mathcal{C}, \\ (\texttt{split}\ \ ch\ \ ch_1, ch_2 : c, e, s) & (\alpha, \beta, qs) \end{array}} \\ \Longrightarrow & & \\ \multicolumn{3}{|c|}{\begin{array}{cc} \mathcal{P}, & \mathcal{C}, (\alpha, \beta, qs : \texttt{split}\ \ \beta_1\ \ \beta_2), \\ (c, \alpha_2 : \alpha_1 : e, s) & (\alpha_1, \beta_1, []), (\beta_1, \alpha_1, []), \\ & (\alpha_2, \beta_2, []), (\beta_2, \alpha_2, []) \end{array}} \\ \hline \end{array} \tag{18}$$

where $\alpha = e[ch]$, and $\alpha_1, \alpha_2, \beta_1, \beta_2$ are fresh channel identifiers.

The `fork` command is dual to the `split` command. When a process executes the `fork` command (which should be the last command), this process will suspend itself by pushing itself to the back of the corresponding channel queue of α.

$$\begin{array}{|ccc|}\hline \multicolumn{3}{|c|}{\begin{array}{cc} \mathcal{P}, & \mathcal{C}, \\ (\texttt{fork}\ \ ch\ \ \texttt{as}\ \ c_1\ \ c_2 : [], e, s) & (\alpha, \beta, qs) \end{array}} \\ \Longrightarrow & & \\ \multicolumn{3}{|c|}{\begin{array}{cc} \mathcal{P} & \mathcal{C}, \\ & (\alpha, \beta, qs : (\texttt{fork}\ \ c_1\ \ c_2 : [], e, s)) \end{array}} \\ \hline \end{array} \tag{19}$$

where $\alpha = e[ch]$.

Now, we discuss how the `split` and `fork` commands interact. Given channels $(\alpha, \beta, (\texttt{fork}\ \ c_1\ \ c_2 : [], e, s) : qs)$ and $(\beta, \alpha, \texttt{split}\ \ \beta_1\ \ \beta_2 : qs')$ in $\mathcal{C}$, we will pop both of the queues, and put $(c_1, \beta_1 : e, s)$ and $(c_2, \beta_2 : e, s)$ in $\mathcal{P}$ where we note that we augment the environments with channels β_1 and β_2. Notationally, this is given by the following state transition.

$$\begin{array}{|ccc|}\hline \multicolumn{3}{|c|}{\begin{array}{cc} & \mathcal{C}, \\ \mathcal{P} & (\alpha, \beta, (\texttt{fork}\ \ c_1\ \ c_2 : [], e, s) : qs), \\ & (\beta, \alpha, \texttt{split}\ \ \beta_1\ \ \beta_2 : qs') \end{array}} \\ \Longrightarrow & & \\ \multicolumn{3}{|c|}{\begin{array}{cc} \mathcal{P}, & \mathcal{C}, \\ (c_1, \beta_1 : e, s), & (\alpha, \beta, qs), \\ (c_2, \beta_2 : e, s) & (\beta, \alpha, qs') \end{array}} \\ \hline \end{array} \tag{20}$$

3.3.3. *Close/Halt State Transitions.* The `close` command pushes the `close` command onto the back of the corresponding channel's queue.

$$\begin{array}{|ccc|}\hline \multicolumn{3}{|c|}{\begin{array}{cc} \mathcal{P}, & \mathcal{C}, \\ (\texttt{close}\ \ ch : c, e, s) & (\alpha, \beta, qs) \end{array}} \\ \Longrightarrow & & \\ \multicolumn{3}{|c|}{\begin{array}{cc} \mathcal{P}, & \mathcal{C}, \\ (c, e, s) & (\alpha, \beta, qs : \texttt{close}) \end{array}} \\ \hline \end{array} \tag{21}$$

where $\alpha = e[ch]$.

The `halt` terminates the current process, and pushes the `close` command onto the back of the corresponding channel's queue. One may ponder why we would not push the `halt`

command to the back of the corresponding channel's queue, but the reason is simple – this saves us some writing for the transitions since the `halt` is treated no different as `close` when in the channel manager. Note that the `halt` command should be the last instruction in the code.

$$\begin{array}{lll} & \mathcal{P}, & \mathcal{C}, \\ (\texttt{halt}\ \ ch : [], e, s) & & (\alpha, \beta, qs) \\ \Longrightarrow & & \\ & \mathcal{P} & \mathcal{C}, \\ & & (\alpha, \beta, qs : \texttt{close}) \end{array} \tag{22}$$

where $\alpha = e[ch]$.

Now, we discuss the state transitions for the `close` command in the channel manager. The `close` command in the channel manager simply just removes both of the channels. Note that in a well-formed program we may assert that the channel queues are empty after the `close` command.

$$\begin{array}{lll} & & \mathcal{C}, \\ & \mathcal{P} & (\alpha, \beta, \texttt{close} : []), \\ & & (\beta, \alpha, \texttt{close} : []) \\ \Longrightarrow & & \\ & \mathcal{P} & \mathcal{C} \end{array} \tag{23}$$

3.3.4. *Channel Identification State Transitions.* The `|=|` command identifies two channels to be the same channel. So given a process with ch_1 `|=|` ch_2 and writing $\alpha_1 = e[ch_1]$ and $\beta_2 = e[ch_2]$; this command will regard α_1 and β_2 as the same channel for the system. We discuss exactly how this mechanism works in more detail. Since this should be the last instruction in the code, the corresponding channels in the channel manager $(\alpha_1, \gamma, ps), (\gamma, \alpha_1, qs)$ and $(\beta_2, \delta, ps'), (\delta, \beta_2, qs')$ must necessarily have empty queues for ps and ps'. Then, this means that there will be no more pending commands to be put on ps and ps', so to regard α_1 and β_2 as the same channel, this amounts to allowing replacing the four channels with $(\gamma, \delta, qs), (\delta, \gamma, qs')$. Notationally, this is given by the following state transition.

$$\begin{array}{lll} & \mathcal{P}, & \mathcal{C}, \\ (ch_1\ \ \texttt{|=|}\ \ ch_2 : [], e, s) & & (\alpha_1, \gamma, []), (\gamma, \alpha_1, qs), \\ & & (\beta_2, \delta, []), (\delta, \beta_2, qs') \\ \Longrightarrow & & \\ & \mathcal{P} & \mathcal{C}, \\ & & (\gamma, \delta, qs), \\ & & (\delta, \gamma, qs') \end{array} \tag{24}$$

where $\alpha_1 = e[ch_1]$ and $\beta_2 = e[ch_2]$.

3.3.5. *Hput and Hcase State Transitions.* The command `hput` puts the ith protocol handle on the corresponding channel's queue, and this process proceeds its execution.

$$\begin{array}{lll} & \mathcal{P}, & \mathcal{C}, \\ & (\texttt{hput}\ \ i\ \ ch : c, e, s) & (\alpha, \beta, qs) \\ \Longrightarrow & & \\ & \mathcal{P}, & \mathcal{C}, \\ & (c, e, s) & (\alpha, \beta, qs : \texttt{hput}\ \ i) \end{array} \tag{25}$$

where $\alpha = e[ch]$.

The command `hcase` is dual to the `hput` command. Note that this should be the last instruction in the code, and this similar to `get` suspends itself at the back of the channel queue.

$$\begin{array}{lll} & \mathcal{P}, & \mathcal{C}, \\ & (\texttt{hcase}\ \ ch\ \ [c_1, \ldots, c_n] : [], e, s) & (\alpha, \beta, qs) \\ \Longrightarrow & & \\ & \mathcal{P} & \mathcal{C}, \\ & & (\alpha, \beta, qs : (\texttt{hcase}\ \ [c_1, \ldots, c_n], e, s)) \end{array} \tag{26}$$

where $\alpha = e[ch]$.

Now, we discuss how the `hcase` and `hput` commands interact in the channel manager. In essence, this creates a new process which runs the code taht is selected by the `hput` command.

$$\begin{array}{lll} & & \mathcal{C}, \\ & \mathcal{P} & (\alpha, \beta, (\texttt{hcase}\ \ [c_1, \ldots, c_n], e, s) : qs), \\ & & (\beta, \alpha, \texttt{hput}\ \ i : qs') \\ \Longrightarrow & & \\ & \mathcal{P}, & \mathcal{C}, \\ & (c_i, e, s) & (\alpha, \beta, qs), \\ & & (\beta, \alpha, qs') \end{array} \tag{27}$$

3.3.6. *Race State Transition.* The `race` command will choose which code to execute depending on the first opposing queue to send a command back. We will decline to write this in our state transition notation, and instead describe how such a translation should occur. So, given a process $(\texttt{race}\ \ (ch_1, c_1)\ \ (ch_2, c_2) : [], e, s)$, we should look up the channels $\alpha_1 = e[ch_1], \alpha_2 = e[ch_2]$. Then, we should push $(\texttt{race}\ \ (\alpha_1, c_1)\ \ (\alpha_2, c_2) : [], e, s)$ on the back of both of the queues corresponding to α_1 and α_2 i.e., $(\alpha_1, \beta_1, qs_1)$ transitions to $(\alpha_1, \beta, qs_1 : (\texttt{race}\ \ (\alpha_1, c_1)\ \ (\alpha_2, c_2) : [], e, s))$ and $(\alpha_2, \beta_2, qs_2)$ transitions to $(\alpha_2, \beta, qs_2 : (\texttt{race}\ \ (\alpha_1, c_1)\ \ (\alpha_2, c_2) : [], e, s))$. Then, whichever $(\texttt{race}\ \ (\alpha_1, c_1)\ \ (\alpha_2, c_2) : [], e, s)$ reaches the front of its queue first such that the channel queue of either β_1 or β_2 are non empty, will proceed to execute its code c_1 and c_2 with the same stack and environment while terminating the other $(\texttt{race}\ \ (\alpha_1, c_1)\ \ (\alpha_2, c_2) : [], e, s)$ in the other channel queue.

3.4. **Implementation Difficulties.** In this section we discuss ideas when implementing this machine on stock hardware. As the nomenclature suggests, one would most likely want the set $\mathcal{P}$ of processes to execute each process on a separate thread. If there are more

processes available than threads on the machine, a suggestion could be to use a thread pool (a well known technique) to manage the tasks of the various processes. Then, one would have to resolve the scheduling of when and how long each process should execute, and there are many trade offs between various approaches. For example, some languages (e.g. Haskell [9]) do a round robin scheduler for fairness.

This is a rather natural way to get easy concurrency since these processes will essentially run independently of each other with their own memory with the exception of the channel manager which is where most implementation complications arise. With this presentation, it's clear that the channels would be simply pointers to a queue for which a process may interact with the queue via a standard `front`, `push`, and `pop` queue interface. But with this view, when a process suspends itself (e.g. with the `get`, `fork`, and `hcase` commands) there is a question of when that process will wake up and who will wake the process up. More explicitly put, since the channels are just queues which are interacted to by a process, a channel has no notion of knowing when itself should wake up. In the following paragraphs, we will discuss how to tackle this in the context of (17) (the `get`/`put` command) but a similar idea applies to other processes which suspend.

One way to tackle this is to make the process that suspends with a `get` to busy loop until it reaches the front of the queue and sees a corresponding `put` in the other queue , then this process should wake itself up. Busy looping for an event to occur is a waste of CPU time, but thankfully this is well studied in the operating system world and using a condition variables [1] could help mitigate this. Alternatively, if the machine is being implemented in a language like Haskell, one could use software transactional [9] to force the process to block until such condition is true and the Haskell run time system will manage the busy looping. Of course, one could also implement their own scheduling system for suspending processes as well.

The other alternative is to make both the processes which `get` and `put` responsible for waking themselves up i.e., we share the responsibility of waking up the process which `get`s on both the process which executes `get` and the process which executes `put`. This is partially inspired by a well known technique from operating systems that use a ready queue to suspend processes which are waiting for user input. In particular, the given a process ($\texttt{get}\ \ ch : c, e, s$) with channels (α, β, qs) (i.e., $\alpha = e[ch]$) and (β, α, qs') will execute either of the steps atomically:

- If qs empty and there is a `put` value on qs', then pop the front of qs' and immediately start executing itself with its environment augmented with popped value of qs'.
- Otherwise, suspend itself at the back of qs.

In the case when the process which `get` suspends, this forces the responsibility of waking up the process which executes `get` on the process which executes the `put` command. So, if given a process ($\texttt{put}\ \ v\ \ ch : c, e, s$) with $\beta = e[ch]$, this process must execute the either of the following steps atomically.

- If qs' empty and if there is a process suspending on qs, then execute wake up the process which is suspending on qs (augmenting the environment with v).
- Otherwise, simply push v to the back of the queue qs'.

In the above state transitions, this idea is a slightly "stricter" notion of forcing the machine to wake up a process at a particular time instead of having (17) which allowed processes to wake up at an arbitrary time.

Moving away from processes which suspend, we discuss implementing the `race` command. As we described, the race command races two processes waiting for input on a queue and the winner is the only process which executes. Here, we discuss the scenario for racing n processes against each other. The main difficulty here is knowing which process actually won the race, and removing the processes which do not win the race. But this can be done by giving each process that races a region of shared memory which starts at 0, and when a process thinks that they have won a race (a process knows it has won the race either by the opposing process waking this process up, or immediately realizing that it can wake itself up when suspending) they atomically increment the region of shared memory and return the previous value (i.e., a fetch and add instruction). If the value they return is 0, they know the process knows it has won the race and can thus continue its execution. Otherwise, the process knows it has lost the race and can then kill itself. We will remark that this gives a means of referencing counting this region of shared memory that can be deallocated when the nth process decides to kill itself.

4. Conclusion

In this document, we have discussed an alternative presentation of CaMPL's abstract machine and considerations to make for a concurrent implementation. Further work on this should include discussion of memory management and garage collection of this language. In particular, adapting the work [5, 10] which discusses parallel garbage collection for which processes have local heaps and messages are put on a reference counted global heap could be applied to allow here. Such an approach would avoid a global stop-the-world mark step that traditional garbage collectors have. After, all that would remain is to implement the system which would hopefully yield a fast and efficient abstract machine for this system on stock hardware.

References

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 1.00. Arpaci-Dusseau Books, Aug. 2018.

[2] J.R.B Cockett and Craig Pastro. "The logic of message-passing". eng. In: *Science of computer programming* 74.8 (2009), pp. 498–533. ISSN: 0167-6423.

[3] Robin Cockett. *Notes on Evaluating Lambda-Calculus Terms and Abstract Machines*. `https://pages.cpsc.ucalgary.ca/~robin/class/521/lectures_lambda/abstract-machines.pdf`. Accessed: 2022-4-10. 2016.

[4] Stephan Diehl, Pieter Hartel, and Peter Sestoft. "Abstract Machines for Programming Language Implementation". eng. In: *Future generation computer systems* 16.7 (2000), pp. 739–751. ISSN: 0167-739X.

[5] Erik Johansson, Konstantinos Sagonas, and Jesper Wilhelmsson. "Heap Architectures for Concurrent Languages Using Message Passing". In: *SIGPLAN Not.* 38.2 supplement (June 2002), pp. 88–99. ISSN: 0362-1340. DOI: `10.1145/773039.512440`. URL: `https://doi.org/10.1145/773039.512440`.

[6] Prashant Kumar. *Implementation of Message Passing Language*. 2018. DOI: 10.11575/PRISM/10182. URL: https://prism.ucalgary.ca/handle/1880/106402.

[7] Xavier Leroy. *Functional programming languages. Part II: Abstract Machines*. https://xavierleroy.org/mpri/2-4/machines.pdf. Accessed: 2022-4-10. 2016.

[8] Reginald Lybbert. "Progress for the Message Passing Logic". Undergraduate thesis. 2019.

[9] Simon Marlow. *Parallel and Concurrent Programming in Haskell*. eng. O'Reilly Media, 2013. ISBN: 1449335934.

[10] Robert Virding. "A Garbage Collector for the Concurrent Real-Time Language Erlang". eng. In: *Memory Management*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 343–354. ISBN: 9783540603689.