AMPL, an abstract machine for MPL

June 15, 2017

1 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.

1.1 SAMPL Constructs

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 $\langle$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.

1.2 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.

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

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 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.

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

1.3 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.

2 Concurrent Abstract Machine for MPL (CAMPL)

CAMPL is the part of AMPL to which the concurrent MPL code compiles to. The design of CAMPL is one of the most important contributions of this thesis.

2.1 CAMPL Commands

In Table the CAMPL commands have been described. There is one CAMPL construct corresponding to every MPL construct. The concurrent MPL constructs have already been described in the first chapter of the thesis. The high level description of the CAMPL commands is the same as their corresponding CAMPL construct but in the below description we will mainly focus on the low level details of the CAMPL commands.

2.2 CAMPL Components

CAMPL consists of 2 main components:

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

2.2.1 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.

2.2.2 Process Mananger