A Virtual Machine for simPL

YSC3208: Programming Design & Language Implementation

Răzvan Voicu

Week 6, Feb 13 - 17, 2017

simPLa is ePL Without Division

$$\frac{}{n} \qquad \frac{}{\texttt{true}} \qquad \frac{}{\texttt{false}}$$

$$\frac{E_1 \quad E_2}{p[E_1, E_2]}\ p \in \{\texttt{|},\ \texttt{\&},\ \texttt{+},\ \texttt{-},\texttt{*},\texttt{=},\texttt{>},\texttt{<}\}. \qquad \frac{E}{p[E]}\ p \in \{\texttt{\\},\texttt{\~}\}$$

SVMLa

$$\frac{}{\texttt{DONE}} \qquad \frac{s}{\texttt{LDCI}\ i\ .\ s} \qquad \frac{s}{\texttt{LDCB}\ b\ .\ s}$$

$$\frac{s}{\texttt{PLUS}.s} \qquad \frac{s}{\texttt{MINUS}.s} \qquad \frac{s}{\texttt{TIMES}.s} \qquad \ldots$$

Compiling simPLa to SVMLa

The translation from simPLa to SVMLa is accomplished by a function

$$: \mathrm{simPLa} \rightarrow \mathrm{SVML}$$

which appends the instruction `DONE` to the result of the auxilary translation function $\hookrightarrow$.

$$\frac{E \hookrightarrow s}{E \twoheadrightarrow s.\mathtt{DONE}}$$

Compiling, continued

$$\frac{n \rightarrowtail_{\mathbf{N}} i}{n \hookrightarrow \texttt{LDCI}\ i}$$

$$\frac{}{\texttt{true} \hookrightarrow \texttt{LDCB}\ \mathit{true}}$$

$$\frac{}{\texttt{false} \hookrightarrow \texttt{LDCB}\ \mathit{false}}$$

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1{+}E_2 \hookrightarrow s_1.s_2.\texttt{PLUS}}$$

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1{*}E_2 \hookrightarrow s_1.s_2.\texttt{TIMES}}$$

Compiling, continued

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1 \& E_2 \hookrightarrow s_1.s_2.\mathtt{AND}}$$

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1 \,|\, E_2 \hookrightarrow s_1.s_2.\mathtt{OR}}$$

$$\frac{E \hookrightarrow s}{\backslash E \hookrightarrow s.\mathtt{NOT}}$$

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1 < E_2 \hookrightarrow s_1.s_2.\mathtt{LT}}$$

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1 > E_2 \hookrightarrow s_1.s_2.\mathtt{GT}}$$

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1 = E_2 \hookrightarrow s_1.s_2.\mathtt{EQ}}$$

Executing SVMLa Code

- Given: SVML program s
- To be defined: machine M_s
- M_s keeps two *registers*
    - Program counter pc
    - Operand stack os

Running the Machine

- Starting configuration: $(\langle\rangle, 0)$
- End configuration: $s(pc) = \texttt{DONE}$.
- Result:

$$R(M_s) = v, \text{ where } (\langle\rangle, 0) \rightrightarrows_s^* (\langle v.os\rangle, pc), \text{ and } s(pc) = \texttt{DONE}$$

simPLb

- simPLb adds division to simPLa
- Add $\perp$ as possible stack value
- Division by zero pushes $\perp$ on the stack, and jumps to `DONE`

Execution of SVMLb

$$\frac{s(pc) = \texttt{DIV}}{(0.i_1.os, pc) \rightrightarrows_s (\bot.os, |s| - 1)}$$

$$\frac{s(pc) = \texttt{DIV}, i_2 \neq 0}{(i_2.i_1.os, pc) \rightrightarrows_s (i_1/i_2.os, pc + 1)}$$

simPLc

- simPLc adds conditionals to simPLb
- Idea: introduce conditional and unconditional jump instructions
- How can we jump from one part of the SVML program to another?
- Answer: by setting the program counter to the address of the jump target

Translation of Conditionals

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2 \qquad E_3 \hookrightarrow s_3}{\begin{array}{c}\texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3\ \texttt{end} \hookrightarrow \\ s_1.\texttt{JOFR}\ |s_2| + 2.s_2.\texttt{GOTOR}\ |s_3| + 1.s_3\end{array}}$$

Example

```
2 *
if true | false
then 1+2
else 2+3
end
```

becomes

$[\mathit{LDI}\ 2,\ \mathit{LDCB}\ \mathit{true},\ \mathit{LDCB}\ \mathit{false},\ \mathit{OR},\ \mathit{JOFR}{+}5,\ \mathit{LDCI}\ 1,\ \mathit{LDCI}\ 2,\ \mathit{PLUS},\ \mathit{GOTOR}{+}4,\ \mathit{LDCI}\ 2,\ \mathit{LDCI}\ 3,\ \mathit{PLUS},\ \mathit{MUL},\ \mathit{DONE}]$

Execution of SVMLc

$$\frac{s(pc) = \texttt{GOTOR}\ i}{(os, pc) \rightrightarrows_s (os, pc + i)}$$

$$\frac{s(pc) = \texttt{JOFR}\ i}{(true.os, pc) \rightrightarrows_s (os, pc + 1)} \qquad \frac{s(pc) = \texttt{JOFR}\ i}{(false.os, pc) \rightrightarrows_s (os, pc + i)}$$

Example

$$[\mathit{LDI}\ 2,\ \mathit{LDCB}\ \mathit{true},\ \mathit{LDCB}\ \mathit{false},\ \mathit{OR},\ \mathit{JOFR}\ 5,\ \mathit{LDCI}\ 1,\ \mathit{LDCI}\ 2,\ \mathit{PLUS},\ \mathit{GOTOR}\ 4,\ \mathit{LDCI}\ 2,\ \mathit{LDCI}\ 3,\ \mathit{PLUS},\ \mathit{MUL},\ \mathit{DONE}]$$

$$(\langle\rangle, 0) \rightrightarrows_s (\langle 2\rangle, 1) \rightrightarrows_s (\langle \mathit{true}, 2\rangle, 2) \rightrightarrows_s (\langle \mathit{false}, \mathit{true}, 2\rangle, 3) \rightrightarrows_s (\langle \mathit{true}, 2\rangle, 4) \rightrightarrows_s (\langle 2\rangle, 5) \rightrightarrows_s (\langle 1, 2\rangle, 6) \rightrightarrows_s (\langle 2, 1, 2\rangle, 7) \rightrightarrows_s (\langle 3, 2\rangle, 8) \rightrightarrows_s (\langle 3, 2\rangle, 12) \rightrightarrows_s (\langle 6\rangle, 13)$$

Implementation of simPLc

- Compiler can generate and use symbolic labels.
- Translate later to use absolute jump addresses
- Corresponding instructions: `GOTO` and `JOF`

Using Symbolic Addresses

LABEL is a dummy instruction to capture the symbolic address of the next instruction.

$$\frac{\textit{fresh } l_1, l_2 \qquad E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2 \qquad E_3 \hookrightarrow s_3}{\begin{array}{l}\texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \texttt{ end} \\ \hookrightarrow s_1.\texttt{JOF } l_1.s_2.\texttt{GOTO } l_2.\texttt{LABEL } l_1.s_3.\texttt{LABEL } l_2\end{array}}$$

Example

```
2 *
if true | false
then 1+2
else 2+3
end
```

compiles to:

	[LDCI 2,LDCB true,LDCB false,OR,JOF l_0,
	LDCI 1,LDCI 2,PLUS,GOTO l_1,
LABEL l_0 :	LDCI 2,LDCI 3,PLUS,
LABEL l_1 :	TIMES,DONE]

Example

transforms (with address linking) to:

```
      [LDCI 2,LDCB true,LDCB false,OR,JOF 9,
       LDCI 1,LDCI 2,PLUS,GOTO 12,
 9 :   LDCI 2,LDCI 3,PLUS,
12 :   TIMES,DONE]
```

Rules for New Instructions (with absolute addr)

$$\frac{s(pc) = \texttt{GOTO}\ i}{(os, pc) \rightrightarrows_s (os, i)}$$

$$\frac{s(pc) = \texttt{JOF}\ i}{(true.os, pc) \rightrightarrows_s (os, pc+1)} \qquad \frac{s(pc) = \texttt{JOF}\ i}{(false.os, pc) \rightrightarrows_s (os, i)}$$

1. A Virtual Machine for simPL
   - simPLa: An Old Hat
   - simPLb: "Adding Division"
   - simPLc: Jumping Up and Down
   - simPLd: Getting Serious
   - simPLe: Getting Recursive
   - Tail Recursion: Getting Efficient

simPLd

$$\frac{}{x} \qquad \frac{E}{\texttt{fun } x_1 \cdots x_n \texttt{ -> } E \texttt{ end}} \qquad \frac{E \quad E_1 \quad \cdots \quad E_n}{\texttt{(}E\ E_1 \cdots E_n\texttt{)}}$$

Outline

- Compilation of identifiers
- Execution of identifiers
- Compilation of function application
- Compilation of function definition
- Execution of function definition
- Execution of function application
- Returning from a function

Compilation of Identifiers

- add register e (environment), mapping identifiers to denotable values.
- translation: $$\frac{}{x \hookrightarrow \texttt{LDS } x}$$
- Later, will find a mapping from identifier to integer offsets on environment stack.

Execution of Identifiers

$$\frac{s(pc) = \texttt{LDS}\ x}{(os, pc, e) \rightrightarrows_s (e(x).os, pc + 1, e)}$$

Compilation of Function Application

$$\frac{E \hookrightarrow s \qquad E_1 \hookrightarrow s_1 \ \cdots \ E_n \hookrightarrow s_n}{(E\ E_1 \cdots E_n) \hookrightarrow s_n.\ldots.s_1.s.\mathtt{CALL}\ n}$$

Note that arguments are compiled in reverse order. This allow us to support over-applications, if needed.

Compilation of Function Definition (without Labels)

$$\frac{E \hookrightarrow s \qquad vs = fv(E) - \{x_1 \cdots x_n\}}{\begin{array}{l}\texttt{fun}\ x_1 \ldots x_n \texttt{->} E\ \texttt{end} \hookrightarrow \\ \texttt{LDFS}\ [vs][x_1 \cdots x_n].\texttt{GOTOR}\ |s| + 1.s.\texttt{RTN}\end{array}}$$

What is *vs*? Answer : Variables from the environment when function was created.

Compilation using Symbolic Labels

$$\frac{E \hookrightarrow s \qquad vs = fv(E) - \{x_1 \cdots x_n\}}{\begin{array}{l}\texttt{fun } x_1 \ldots x_n \texttt{->} E \texttt{ end} \hookrightarrow \\ \texttt{LDFS } [vs][x_1 \cdots x_n].\texttt{GOTO } l.s.\texttt{RTN}.\texttt{LABEL } l:\end{array}}$$

Execution of Function Definition (Building a Closure)

$$\frac{s(pc) = \texttt{LDFS}[v_1 \cdots v_m][x_1 \cdots x_n]}{(os, pc, e) \rightrightarrows_s ((pc{+}2, x_1 \cdots x_n, e\#[v_1 \cdots v_m]).os, pc{+}1, e)}$$

We introduce triple (*address*, *formals*, e) to represent function value on the operand stack. Such a triple is called a *closure*.

Idea

- Add another machine register that can store the machine state to be re-installed after functions return.
- Since functions call other functions, we need a stack, to support the restoration of caller's state.
- This stack is called the *runtime stack*, denoted by rs.
- Stack entries are called *stack frames* and consist of $(r, address, e)$ where r denotes the number of residual arguments on operand stack.

Execution of Function Application

Over-Application when $m < n$
Exact-Application when $m = n$

$$\frac{s(pc) = \texttt{CALL}\ n \qquad 0 < m \leq n}{\begin{array}{l}((address, x_1 \cdots x_m, e').v_1.\ldots.v_m.os, pc, e, rs) \rightrightarrows_s \\ (os, address, e'[x_1 \leftarrow v_1]\cdots[x_m \leftarrow v_m], (n-m, pc+1, e).rs)\end{array}}$$

Examples of Applications

Over-Applied Applications:

```
(fun x -> (fun y -> x+y end) end  2 3)
```

Under-Applied Applications:

```
(fun x y -> x+y end  2)
```

Pre-processing ==> :

```
fun z -> (fun x y -> x+y end  2 z) end
```

Returning from a Function

Pop stack frame if there are no residual argument.

$$\frac{s(pc) = \texttt{RTN} \qquad r = 0}{(v.os, pc, e, (r, pc', e').rs) \rightrightarrows_s (v.os, pc', e', rs)}$$

Returning from a Function

Perform further application if there are some residual arguments.

$$\frac{s(pc) = \texttt{RTN} \qquad 0 < m \leq r}{\begin{array}{l}((address, x_1 \cdots x_m, e').v_1. \ldots .v_m.os, pc, e, (r, pc_2, e_2).rs) \rightrightarrows_s \\ (os, address, e'[x_1 \leftarrow v_1] \cdots [x_m \leftarrow v_m], (r - m, pc_2, e_2).rs)\end{array}}$$

Representation of Environments

We organize each environment as an array of values for both non-local arguments and local arguments.

Copying from Stack to Environments

During call application, arguments in operation stack are copied to environment and placed after the non-local variables.

Compilation of Application

Applications remember the number of arguments. The application may either by *fully-applied* or *over-applied* but never under-applied due to our pre-processing.

Exact Application Example

```
(fun x -> x + 1 end  2)
```

becomes

```
     [LDCI 2,LDF([],1,4),CALL 1,DONE,
4 :   LD (x,0),LDCI 1,PLUS,RTN]
```

Handling Non-Local Variables inside Closures

Why bind non-local variables inside the closure?
Reasons : closures may escape the scope of their non-local variables when under-applied.

```
let {int->int->int} f =
  (fun {int->int->int} x
      -> (fun {int->int} y -> x+y end) end) end
in {int->int} f 4 end
```

Inner function of `f` escapes the scope of its non-local variable `x`.
What is the disadvantage?

Partial -> Full Application

```
let {int->int->int} f =
  (fun {int->int->int} x
      -> (fun {int->int} y -> x+y end) end)
in {int->int} f 4 end
```

transforms to:

```
let {int->int->int} f =
  (fun {int->int->int} x
      -> (fun {int->int} y -> x+y end) end)
in {int->int} (fun {int->int} pa -> f 4 pa end) end
```

Representation of Closures

Compiled Code:

```
     [LDF([],1,11),LDF([],1,4),CALL 1,DONE,
4:    LDF([(f,0)],1,6),RTN,
6:    LD (pa,1),LDCI 4,LD (f,0),CALL 2,RTN,
10:   LDF([(x,0)],1,13),RTN,
13:   LD (x,0),LD (y,1),PLUS,RTN]
```

Note partial application removed by pre-processing.

Representing Runtime Stack Frames

Each stack frame comprises of (i) number of residual arguments that remain in the operand stack, (ii) address of method, and (iii) bindings for variables used by the method.

Translation of simPLe

$$\frac{E \hookrightarrow s \qquad v_1 \cdots v_m = fv(E) - \{f, x_1 \cdots x_n\}}{\begin{array}{c}\texttt{recfun}\ f\ x_1 \ldots x_n\ \texttt{->}\ E\ \texttt{end} \hookrightarrow \\ \texttt{LDFRS}([v_1 \cdots v_m], f, [x_1 \cdots x_n]).\texttt{GOTO}\ l.s.\texttt{RTN}.\texttt{LABEL}\ l\end{array}}$$

Execution of SVMLe

$$\frac{s(pc) = \texttt{LDFRS}([v_1 \cdots v_m], f, [x_1 \cdots x_n])}{(os, pc, e) \rightrightarrows_s (c{=}(pc+2, x_1 \cdots x_n, e\#[v_1 \cdots v_m][f \leftarrow c]).os, pc+1, e)}$$

Idea of Implementation

Use same call. Create a circular data structure.
Environment of recursive function value points to function itself.

Tail Recursion: Motivation

- Each function call creates a new stack frame.
- Function calls consume significant amount of memory.
- There are situations, where the creation of a new stack frame can be avoided.

Tail Recursion: Conditions

- last action in the body of a function is another function call
- calling function and the function to be called is the same recursive function

Terminology

- A recursive call, which appears in the body of a recursive function as the last instruction to be executed, is called *tail call*.
- A recursive function, in which all recursive calls are tail calls, is called *tail-recursive*.

Example

```
let {.}
   facloop = recfun  facloop {.} n acc ->
                 if n = 1 then acc
                 else (facloop (n-1) (acc*n)) end
                 end
in {int}
   let {.}
      fac = fun {.} n -> (facloop n 1) end
   in {int}
      (fac 4)
   end
end
```

Compilation for Tail Recursion

Replace

`CALL` n.`RTN`

by

`TAILCALL` n

if operator of the call is the function variable of the immediately surrounding recursive function definition

Example (normal calls)

```
      [LDFR([(facloop,1)],2,17),LDF([],1,4),CALL 1,DONE,
 4 :  LDF([(facloop,0)],1,12),LDF([],1,8),CALL 1,RTN,
 8 :  LDCI 4,LD (fac,0),CALL 1,RTN,
12 :  LDCI 1,LD (n,1),LD (facloop,0),CALL 2,RTN,
17 :  LD (n,1),LDCI 1,EQ,JOF 23,LD (acc,2),GOTO 31,
23 :  LD (acc,2),LD (n,1),TIMES,LD (n,1),LDCI 1,MINUS,
      LD (facloop,0),CALL 2,
31 :  RTN]
```

Example (tail calls)

```
     [LDFR([(facloop,1)],2,14),LDF([],1,4),CALL 1,DONE,
 4 : LDF([(facloop,0)],1,10),LDF([],1,7),TAILCALL 1,
 7 :  LDCI 4,LD (fac,0),TAILCALL 1,
10 :  LDCI 1,LD (n,1),LD (facloop,0),TAILCALL 2,
14 :  LD (n,1),LDCI 1,EQ,JOF 20,LD (acc,2),GOTO 28,
20 :  LD (acc,2),LD (n,1),TIMES,LD (n,1),LDCI 1,MINUS,
      LD (facloop,0),TAILCALL 2,
28 :  RTN]
```

Tail Call Semantics

$$\frac{S(pc) = \texttt{TAILCALL}\ n \qquad s \leq r+n}{\begin{array}{l}((addr, x_1 \cdots x_s, e_2).v_1 \ldots v_s.os, pc, e, (r, npc, ne).rs) \rightrightarrows_s \\ (os, addr, e_2[x_1 \leftarrow v_1] \cdots [x_s \leftarrow v_s], (r+n-s, npc, ne).rs)\end{array}}$$