05—A Virtual Machine for simPL

CS4215: Programming Language Implementation

Martin Henz

February 10, 2012

Generated on Friday 10 February, 2012, 00:05

1 Variable Scoping

2 Two Previous Implementations

3 A Virtual Machine for simPL

Example in Java

```
class myObject extends yourObject {
    static int x;
    int y;
    int f(int z) {
        int w;
        while (z != 0) {
            int v;
            v = w + z--;
        }
        return w:
}}
```

Another Example in Java

```
public class A {
    int y;
    public class B {
        int x;
        void f () {}
} }
```

An Example in C

```
int a;
int main() {
  int b;
  {
    int c;
    ...
  }
  ...
  exit (0);
}
```

An Example in JavaScript

```
<script language="javascript" >
function outer(data) {
    var operand1 = data;
    function inner(operand2) {
        alert(operand1 + operand2)
    }
}
</script>
```

A Common Theme

How to find the variable declaration?

Go from the variable occurrence *outwards* and stop at the first matching declaring construct

Declaring Constructs in Java

- local variables in blocks
- fields (static and non-static) in classes
- method declarations
- class declaration

Declaring Constructs in C

- local variables in blocks
- global variables
- function declarations

Declaring Constructs in simPL

- function declarations
  `fun {...} x -> ... end`
- recursive function declarations
  `recfun f {...} x -> ... end`
- let-expressions

1 Variable Scoping

2 Two Previous Implementations

3 A Virtual Machine for simPL

First attempt: Small-step interpreter

Implementation of scoping

was done by explicitly substituting expressions for variables

Result

Extremely inefficient and cumbersome

Second attempt: Environments

Idea

Keep track of the binding of an identifier to a value in a data structure called *environment*

Entering a scope

amounts to *extending* the environment by a binding of formal parameters to actual argument values

Semantic Domains of simPL1

Semantic domain	Definition	Explanation
Bool	$\{\mathit{true}, \mathit{false}\}$	ring of booleans
Int	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	ring of integers
EV	**Bool + Int**	expressible values
DV	**Bool + Int**	denotable values
Id	alphanumeric string	identifiers
Env	**Id** $\rightsquigarrow$ **DV**	environments

Operations on Environments

We introduce an operation $\Delta[x \leftarrow v]$,
which denotes an environment that works like Δ,
except that $\Delta[x \leftarrow v](x) = v$

Semantic Functions for simPL1

Define $\cdot \rightarrowtail \cdot$ using auxiliary semantic function $\cdot \Vdash \cdot \rightarrowtail \cdot$ that gets an environment as additional argument.

$$\cdot \rightarrowtail \cdot : \textbf{simPL1} \rightarrow \textbf{EV}$$

$$\frac{\emptyset \Vdash E \rightarrowtail v}{E \rightarrowtail v}$$

Auxiliary Semantic Function for simPL1

$$\cdot \Vdash \cdot \rightarrowtail \cdot : \textbf{Env} * \mathrm{simPL1} \rightarrow \textbf{EV}$$

Auxiliary Semantic Function for simPL1 (cont'd)

$$\frac{}{\Delta \Vdash \texttt{true} \rightarrowtail \mathit{true}} \qquad \frac{}{\Delta \Vdash \texttt{false} \rightarrowtail \mathit{false}}$$

$$\frac{n \rightarrowtail_{\mathbf{N}} i}{\Delta \Vdash n \rightarrowtail i} \qquad \frac{}{\Delta \Vdash x \rightarrowtail \Delta(x)}$$

Auxiliary Semantic Function for simPL1 (cont'd)

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 + E_2 \rightarrowtail v_1 + v_2}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 - E_2 \rightarrowtail v_1 - v_2}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 * E_2 \rightarrowtail v_1 \cdot v_2}$$

Auxiliary Semantic Function for simPL1 (cont'd)

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 \& E_2 \rightarrowtail v_1 \wedge v_2}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 \mid E_2 \rightarrowtail v_1 \vee v_2}$$

$$\frac{\Delta \Vdash E \rightarrowtail v}{\Delta \Vdash \backslash E \rightarrowtail \neg v} \qquad \frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 = E_2 \rightarrowtail v_1 \equiv v_2}$$

Auxiliary Semantic Function for simPL1 (cont'd)

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 \texttt{>} E_2 \rightarrowtail v_1 > v_2}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1 \texttt{<} E_2 \rightarrowtail v_1 < v_2}$$

Auxiliary Semantic Function for simPL1 (cont'd)

$$\frac{\Delta \Vdash E \rightarrowtail \mathit{true} \quad \Delta \Vdash E_1 \rightarrowtail v_1}{\Delta \Vdash \texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ end} \rightarrowtail v_1}$$

$$\frac{\Delta \Vdash E \rightarrowtail \mathit{false} \quad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash \texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ end} \rightarrowtail v_2}$$

Auxiliary Semantic Function for simPL1 (cont'd)

$$\frac{\Delta[x_1 \leftarrow v_1]\cdots[x_n \leftarrow v_n] \Vdash E \rightarrowtail v \quad \Delta \Vdash E_1 \rightarrowtail v_1 \quad \cdots \quad \Delta \Vdash E_n \rightarrowtail v_n}{\Delta \Vdash \texttt{let}\ x_1 = E_1 \cdots x_n = E_n\ \texttt{in}\ E\ \texttt{end} \rightarrowtail v}$$

Example

$$\frac{\dfrac{\dfrac{\ldots}{\emptyset[\texttt{AboutPi} \leftarrow 3] \Vdash \texttt{AboutPi + 2} \rightarrowtail \texttt{5}}}{\emptyset \Vdash \texttt{let AboutPi = 3 in AboutPi + 2 end} \rightarrowtail \texttt{5}}}{\texttt{let AboutPi = 3 in AboutPi + 2 end} \rightarrowtail \texttt{5}}$$

Syntactic Domain of simPL3

Add (non-recursive) function definition and application.

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

Semantic Domains of simPL3

Domain name	Definition	Explanation
Bool	$\{\mathit{true}, \mathit{false}\}$	ring of booleans
Int	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	ring of integers
EV	$\mathbf{Bool} + \mathbf{Int} + \{\perp\} + \mathbf{Fun}$	expressible values
DV	$\mathbf{Bool} + \mathbf{Int} + \mathbf{Fun}$	denotable values
Id	alphanumeric string	identifiers
Env	$\mathbf{Id} \rightsquigarrow \mathbf{DV}$	environments
Fun	$\mathbf{DV} * \cdots * \mathbf{DV} \rightsquigarrow \mathbf{EV}$	function values

Auxiliary Semantic Function (cont'd)

$$\Delta \Vdash \texttt{fun}\ \{\ t\ \}\ x_1 \ldots x_n\ \texttt{->}\ E\ \texttt{end} \rightarrowtail f$$

f is function such that
$f(y_1, \ldots, y_n) = v$, where
$\Delta[x_1 \leftarrow y_1] \cdots [x_n \leftarrow y_n] \Vdash E \rightarrowtail v$

Evaluation of Function Application

$$\frac{\Delta \Vdash E \rightarrowtail f \quad \Delta \Vdash E_1 \rightarrowtail v_1 \ \cdots \ \Delta \Vdash E_n \rightarrowtail v_n}{\Delta \Vdash (E \ E_1 \ \ldots \ E_n) \ \rightarrowtail f(v_1, \ldots, v_n)}$$

Syntactic Domain of simPL4

$$\frac{E}{\texttt{recfun}\ f\ \{\ t\ \}\ x_1 \ldots x_n\ \texttt{->}\ E\ \texttt{end}}$$

Discussion simPL4

We would like to add the following rule to our definition of $\rightarrowtail$.

$$\Delta \Vdash \texttt{recfun}\ g\ \{\ t\ \}\ x_1 \ldots x_n\ \texttt{->}\ E\ \texttt{end} \rightarrowtail f$$

f is function such that
$f(y_1, \ldots, y_n) = v$, where
$\Delta[x_1 \leftarrow y_1] \cdots [x_n \leftarrow y_n]$
$[g \leftarrow f] \Vdash E \rightarrowtail v$

Problem

The symbol f occurs on the right hand side of the definition of f. How do we know that such a function f exists? Some expressions have unique solutions for f, others have multiple solutions. Example:

```
(recfun f {int -> int} x -> (f x) end 0)
```

Theory of fixpoints ...

Overall Structure of Interpreter

- Interpreter calls `eval` on expressions
- `eval` transforms expressions into expressible values

Evaluating Constants

```
public class BoolConstant implements Expression {
    public boolean value;
    ...
    public Value eval(Environment e) {
        return new BoolValue(value);
    }
}
```

Environments

```
public class Environment
        extends Hashtable<String,Value> {
    public Environment extend(String v, Value d) {
        Environment e = (Environment)clone();
        e.put(v,d);
        return e;
    }
```

Closures

```
public class Fun implements Expression {
 public Type funType;
 public Vector<String> formals;
 public Expression body;
 public Fun(Type t,Vector<String> xs,Expression b){
    funType = t;
    formals = xs;
    body = b;
 }
 public Value eval(Environment e) {
    return new FunValue(e, formals, body);
}   }
```

1. Variable Scoping
2. Two Previous Implementations
3. A Virtual Machine for simPL
   - simPLa: An Old Hat
   - simPLb: "Adding Division"
   - simPLc: Jumping Up and Down
   - Implementation of simPLc
   - simPLd: Getting Serious
   - simPLe: Getting Recursive
   - Tail Recursion: Getting Efficient

simPLa is ePL Without Division

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

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

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

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

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

$$\frac{E \hookrightarrow s}{Es.\texttt{DONE}}$$

Compiling, continued

$$\frac{n \rightarrowtail_{\mathbf{N}} i}{n \hookrightarrow \texttt{LDCI}\ i} \qquad \frac{}{\texttt{true} \hookrightarrow \texttt{LDCB}\ \mathit{true}} \qquad \frac{}{\texttt{false} \hookrightarrow \texttt{LDCB}\ \mathit{false}}$$

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

Compiling, continued

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

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

$$\frac{E \hookrightarrow s}{\texttt{\textbackslash}E \hookrightarrow s.\texttt{NOT}}$$

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1\texttt{<}E_2 \hookrightarrow s_1.s_2.\texttt{LESS}}$$

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1\texttt{>}E_2 \hookrightarrow s_1.s_2.\texttt{GREATER}}$$

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1\texttt{=}E_2 \hookrightarrow s_1.s_2.\texttt{EQUAL}}$$

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

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

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

Implementation of SVMLb

```
case OPCODES.DIV:     int divisor = os.pop();
                      if (divisor == 0) {
                         os.push(new Error());
                         break loop;
                      } else {
                         os.push(new IntValue(
                                        os.pop().value
                                        / divisor));
                         pc++;
                         break;}
```

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

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

Example

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

becomes

[*LDI* 2, *LDCB true*, *LDCB false*, *OR*, *JOFR* 5, *LDCI* 1, *LDCI* 2, *PLUS*, *GOTOR* 4, *LDCI* 2, *LDCI* 3, *PLUS*, *MUL*, *DONE*]

Execution of SVMLc

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

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

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 computes absolute jump addresses
- Corresponding instructions: `GOTO` and `JOF`

Example

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

becomes

```
[LDI 2          0
LDCB true       1
LDCB false      2
OR              3
JOF 9           4
LDCI 1          5
LDCI 2          6
PLUS            7
GOTO 12         8
LDCI 2          9
LDCI 3         10
PLUS           11
MUL            12
DONE]          13
```

New Rules for New Instructions

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

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

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

Implementation of New Instructions

```
case OPCODES.GOTO:      pc = i.ADDRESS;
                        break;

case OPCODES.JOF:       pc = (os.pop().value)
                             ? pc+1
                             : i.ADDRESS;
                        break;
```

simPLd

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

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: $\dfrac{}{x \hookrightarrow \texttt{LDS}\ x}$

Execution of Identifiers

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

Compilation of Function Application

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

Compilation of Function Definition

$$\frac{E \hookrightarrow s}{\texttt{fun } x_1 \ldots x_n \texttt{ -> } E \texttt{ end} \hookrightarrow \texttt{LDFS } x_1 \ \cdots x_n\texttt{.GOTOR } |s| + 2\texttt{.}s\texttt{.RTN}}$$

Execution of Function Definition

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

Such a triple $(address, formals, e)$ is called a *closure*.

Execution of Function Application

How about this:

$$s(pc) = \texttt{CALL}\ n$$

$$(v_n. \ldots . v_1.(address, x_1 \cdots x_n, e').os, pc, e)$$
$$\rightrightarrows_s (os, address, e'[x_1 \leftarrow v_1] \cdots [x_n \leftarrow v_n])$$

Problem

- What should be the program counter, operand stack and environment after returning from a function?
- What should the machine do when it encounters `RTN`?

Idea

- Add another machine register that can store the machine state to be reinstalled after functions return.
- Since functions call other functions, we need a stack.
- This stack is called the *runtime stack*, denoted by *rs*.
- Stack entries are called *stack frames* and consist of $(address, os, e)$

Execution of Function Application: Second Attempt

$$s(pc) = \texttt{CALL}\ n$$

$$(v_n. \ldots v_1.(address, x_1 \cdots x_n, e').os, pc, e, rs) \rightrightarrows_s$$
$$(\langle\rangle, address, e'[x_1 \leftarrow v_1] \cdots [x_n \leftarrow v_n], (pc + 1, os, e).rs)$$

Returning from a Function

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

Representation of Environments

Compiler predicts the place where the identifier can be found in the environment.

```
public class Environment extends Vector {
   public Environment extend(int numberOfSlots) {
      Environment newEnv = (Environment) clone();
      newEnv.setSize(newEnv.size() + numberOfSlots);
      return newEnv;
   }
}
```

Implementing SVMLd

LD instruction uses index computed by compiler.

```
public class LD extends INSTRUCTION {
    public int INDEX;
    public LD(int i) {
        OPCODE = OPCODES.LD;
        INDEX = i;
    }
}
```

Compilation of Function Definition

```
public class LDF extends INSTRUCTION {
    public int ADDRESS;
    public LDF(int address) {
        OPCODE = OPCODES.LDF;
        ADDRESS = address;
    }
}
```

Compilation of Application

```
public class CALL extends INSTRUCTION {
  public int NUMBEROFARGUMENTS;
  public CALL(int noa) {
     OPCODE = OPCODES.CALL;
     NUMBEROFARGUMENTS = noa;
  }
}
```

Example

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

becomes

```
[LDF 4    0
LDCI 2    1
CALL 1    2
DONE      3
LD 0      4
LDCI 1    5
PLUS      6
RTN]      7
```

Execution of Identifiers

```
case OPCODES.LD:      os.push(e.elementAt(i.INDEX));
                      pc++;
                      break;
```

Representation of Closures

```
public class Closure implements Value {
  public Environment environment;
  public int ADDRESS;
  Closure(Environment e, int a) {
    environment = e;
    ADDRESS = a;
  }
}
```

Execution of Function Definition

```
case OPCODES.LDF:   Environment env = e;
                    os.push(new Closure(env,i.ADDRESS));
                    pc++;
                    break;
```

Representing Runtime Stack Frames

```
public class StackFrame {
  public int pc;
  public Environment environment;
  public Stack operandStack;
  public StackFrame(int p, Environment e, Stack os) {
     pc = p;
     environment = e;
     operandStack = os;
  }
}
```

Execution of Application

```
case OPCODES.CALL: {int n = i.NUMBEROFARGUMENTS;
                    Closure closure
                      = os.elementAt(os.size()-n-1);
                    Environment newEnv
                      = closure.environment.extend(n);
                    int s = newEnv.size();
      for (int j = s - 1; j >= s-n; --j)
         newEnv.setElementAt(os.pop(),j);
                    os.pop(); // function value
                    rs.push(new StackFrame(pc+1,e,os));
                    pc = closure.ADDRESS;
                    e = newEnv; os = new Stack();
                    break;}
```

Returning from a Function

```
case OPCODES.RTN:     Value returnValue = os.pop();
                      StackFrame f = rs.pop();
                      pc = f.pc;
                      e = f.environment;
                      os = f.operandStack;
                      os.push(returnValue);
                      break;
```

Translation of simPLe

$$\frac{E \hookrightarrow s}{\texttt{recfun}\ f\ x_1 \ldots x_n\ \texttt{->}\ E\ \texttt{end} \hookrightarrow \texttt{LDRFS}\ f\ x_1\ \cdots x_n.\texttt{GOTOR}\ |s| + 2.s.\texttt{RTN}}$$

Execution of SVMLe

$$\frac{s(pc) = \texttt{LDRFS}\ f\ x_1 \cdots x_n}{(os, pc, e) \rightrightarrows_s ((pc + 2, f, x_1 \cdots x_n, e).os, pc + 1, e)}$$

Execution of SVMLe, continued

$$s(pc) = \texttt{CALL}\ n$$

$$(v_n.\ldots.v_1.(address, f, x_1 \cdots x_n, e').os, pc, e, rs) \rightrightarrows_s$$
$$(\langle\rangle, address,$$
$$e'[f \leftarrow (address, f, x_1 \cdots x_n, e')][x_1 \leftarrow$$
$$v_1] \cdots [x_n \leftarrow v_n],$$
$$(pc + 1, os, e).rs)$$

Idea of Implementation

Create a circular data structure:
Environment of recursive function value points to function itself.

Implementation of SVMLe

```
case OPCODES.LDRF:    Environment envr = e.extend(1);
                      Value fv = new
                            Closure(envr,i.ADDRESS);
                      envr.setElementAt(fv,e.size());
                      os.push(fv);
                      pc++;
                      break;
```

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
in
   let
      fac = fun n -> (facloop n 1)
   in
      (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

Implementation of Tail Recursion

```
case OPCODES.TAILCALL: { int n = i.NUMBEROFARGUMENTS;
                         Closure closure
                             = os.elementAt(os.size()
                                            -n-1);
                         int s = e.size();
 for (int j = s - 1; j >= s-n; --j)
    e.setElementAt(os.pop(),j);
                         os.pop();
                         pc = closure.ADDRESS;
                         break;
                       }
```

Overview of Next Lecture

- Memory allocation for programs
- A heap memory model
- SVML with heap
- Implementing a heap
- Memory management techniques