

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	$\{true, false\}$	ring of booleans
Int	$\{\dots, -2, -1, 0, 1, 2, \dots\}$	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.

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

$$\emptyset \Vdash E \rightarrowtail v$$

$$E \rightarrowtail v$$

Auxiliary Semantic Function for simPL1

$\cdot \Vdash \cdot \rightarrow \cdot : \mathbf{Env} * \text{simPL1} \rightarrow \mathbf{EV}$

Auxiliary Semantic Function for simPL1 (cont'd)

$$\Delta \Vdash \text{true} \rightarrow \text{true}$$

$$n \rightarrow_{\mathbb{N}} i$$

$$\Delta \Vdash n \rightarrow i$$

$$\Delta \Vdash \text{false} \rightarrow \text{false}$$

$$\Delta \Vdash x \rightarrow \Delta(x)$$

Auxiliary Semantic Function for simPL1 (cont'd)

$$\Delta \Vdash E_1 \rightarrow v_1 \quad \Delta \Vdash E_2 \rightarrow v_2$$

$$\Delta \Vdash E_1 + E_2 \rightarrow v_1 + v_2$$

$$\Delta \Vdash E_1 \rightarrow v_1 \quad \Delta \Vdash E_2 \rightarrow v_2$$

$$\Delta \Vdash E_1 - E_2 \rightarrow v_1 - v_2$$

$$\Delta \Vdash E_1 \rightarrow v_1 \quad \Delta \Vdash E_2 \rightarrow v_2$$

$$\Delta \Vdash E_1 * E_2 \rightarrow v_1 \cdot v_2$$

Auxiliary Semantic Function for simPL1 (cont'd)

$$\Delta \Vdash E_1 \rightarrow v_1 \quad \Delta \Vdash E_2 \rightarrow v_2$$

$$\Delta \Vdash E_1 \& E_2 \rightarrow v_1 \wedge v_2$$

$$\Delta \Vdash E_1 \rightarrow v_1 \quad \Delta \Vdash E_2 \rightarrow v_2$$

$$\Delta \Vdash E_1 | E_2 \rightarrow v_1 \vee v_2$$

$$\Delta \Vdash E \rightarrow v$$

$$\Delta \Vdash E_1 \rightarrow v_1 \quad \Delta \Vdash E_2 \rightarrow v_2$$

$$\Delta \Vdash \setminus E \rightarrow \neg v$$

$$\Delta \Vdash E_1 = E_2 \rightarrow v_1 \equiv v_2$$

Auxiliary Semantic Function for simPL1 (cont'd)

$$\Delta \Vdash E_1 \rightarrow v_1 \quad \Delta \Vdash E_2 \rightarrow v_2$$

$$\Delta \Vdash E_1 > E_2 \rightarrow v_1 > v_2$$

$$\Delta \Vdash E_1 \rightarrow v_1 \quad \Delta \Vdash E_2 \rightarrow v_2$$

$$\Delta \Vdash E_1 < E_2 \rightarrow v_1 < v_2$$

Auxiliary Semantic Function for simPL1 (cont'd)

$$\Delta \Vdash E \rightarrow \text{true} \quad \Delta \Vdash E_1 \rightarrow v_1$$

$$\Delta \Vdash \text{if } E \text{ then } E_1 \text{ else } E_2 \text{ end} \rightarrow v_1$$

$$\Delta \Vdash E \rightarrow \text{false} \quad \Delta \Vdash E_2 \rightarrow v_2$$

$$\Delta \Vdash \text{if } E \text{ then } E_1 \text{ else } E_2 \text{ end} \rightarrow v_2$$

Auxiliary Semantic Function for simPL1 (cont'd)

$$\Delta[x_1 \leftarrow v_1] \cdots [x_n \leftarrow v_n] \Vdash E \rightarrow v \quad \Delta \Vdash E_1 \rightarrow v_1 \quad \cdots \quad \Delta \Vdash E_n \rightarrow v_n$$

$$\Delta \Vdash \text{let } x_1 = E_1 \cdots x_n = E_n \text{ in } E \text{ end} \rightarrow v$$

Example

...

$$\emptyset[\text{AboutPi} \leftarrow 3] \Vdash \text{AboutPi} + 2 \rightarrow 5$$

$$\emptyset \Vdash \text{let AboutPi} = 3 \text{ in AboutPi} + 2 \text{ end} \rightarrow 5$$

$$\text{let AboutPi} = 3 \text{ in AboutPi} + 2 \text{ end} \rightarrow 5$$

Syntactic Domain of simPL3

Add (non-recursive) function definition and application.

 E

 $\text{fun } \{ t \} \ x_1 \dots x_n \rightarrow E \ \text{end}$ $E \ E_1 \ \dots \ E_n$

 $(E \ E_1 \ \dots \ E_n)$

Semantic Domains of simPL3

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

Auxiliary Semantic Function (cont'd)

$$\Delta \Vdash \text{fun } \{ t \} x_1 \dots x_n \rightarrow E \text{ end} \rightarrow f$$

f is function such that

$f(y_1, \dots, y_n) = v$, where

$$\Delta[x_1 \leftarrow y_1] \cdots [x_n \leftarrow y_n] \Vdash E \rightarrow v$$

Evaluation of Function Application

$$\Delta \Vdash E \rightarrow f \quad \Delta \Vdash E_1 \rightarrow v_1 \quad \dots \quad \Delta \Vdash E_n \rightarrow v_n$$

$$\Delta \Vdash (E \ E_1 \ \dots \ E_n) \rightarrow f(v_1, \dots, v_n)$$

Syntactic Domain of simPL4

E

```
recfun f { t } x1...xn -> E end
```

Discussion simPL4

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

$$\Delta \Vdash \text{recfun } g \{ t \} x_1 \dots x_n \rightarrow E \text{ end} \rightarrow f$$

f is function such that

$f(y_1, \dots, y_n) = v$, where

$\Delta[x_1 \leftarrow y_1] \dots [x_n \leftarrow y_n]$

$[g \leftarrow f] \Vdash E \rightarrow v$

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

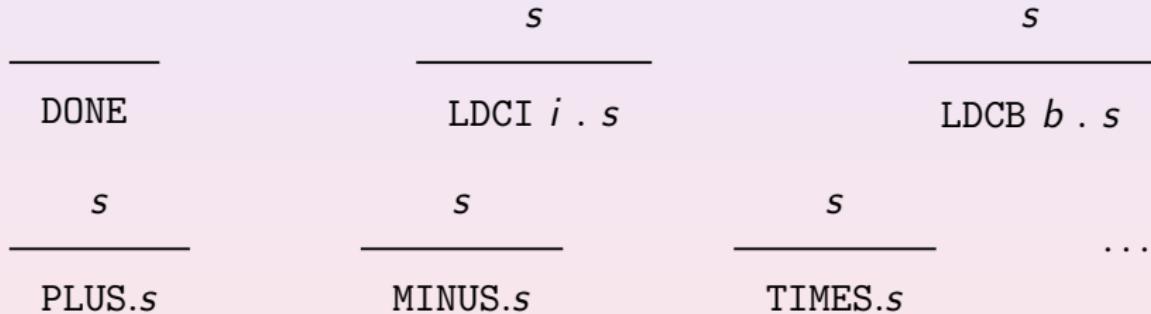
3 A Virtual Machine for simPL

- 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

$$\frac{}{n} \qquad \frac{}{\text{true}} \qquad \frac{}{\text{false}}$$
$$\frac{E_1 \quad E_2}{p \in \{|, \&, +, -, *, =, >, <\}. \quad p[E_1, E_2]}$$
$$\frac{E}{p \in \{\backslash\} \quad p[E]}$$

SVMLa



Compiling simPLa to SVMLa

The translation from simPLa to SVMLa is accomplished by a function

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

which appends the instruction DONE to the result of the auxiliary translation function \hookrightarrow .

$$E \hookrightarrow s$$

$$Es.\text{DONE}$$

Compiling, continued

$n \rightarrow_N i$

$n \hookrightarrow \text{LDCI } i$

$\text{true} \hookrightarrow \text{LDCB } \text{true}$

$\text{false} \hookrightarrow \text{LDCB } \text{false}$

$E_1 \hookrightarrow s_1$

$E_2 \hookrightarrow s_2$

$E_1 \hookrightarrow s_1$

$E_2 \hookrightarrow s_2$

$E_1 + E_2 \hookrightarrow s_1.s_2.\text{PLUS}$

$E_1 * E_2 \hookrightarrow s_1.s_2.\text{TIMES}$

Compiling, continued

$$E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2$$

$$E_1 \& E_2 \hookrightarrow s_1.s_2.\text{AND}$$

$$E \hookrightarrow s$$

$$\setminus E \hookrightarrow s.\text{NOT}$$

$$E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2$$

$$E_1 > E_2 \hookrightarrow s_1.s_2.\text{GREATER}$$

$$E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2$$

$$E_1 | E_2 \hookrightarrow s_1.s_2.\text{OR}$$

$$E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2$$

$$E_1 < E_2 \hookrightarrow s_1.s_2.\text{LESS}$$

$$E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2$$

$$E_1 = E_2 \hookrightarrow s_1.s_2.\text{EQUAL}$$

Executing SVMLa Code

- Given: SVMLa 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: $s(pc) = \text{DONE}$.
- Result:

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

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 SVMILb

$$s(pc) = \text{DIV}$$

$$(0.i_1.os, pc) \Rightarrow_s (\perp.os, |s| - 1)$$

$$s(pc) = \text{DIV}, i_2 \neq 0$$

$$(i_2.i_1.os, pc) \Rightarrow_s (i_1/i_2.os, pc + 1)$$

Implementation of SVMILb

```
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

$$E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2 \quad E_3 \hookrightarrow s_3$$

if E_1 then E_2 else E_3 end $\hookrightarrow s_1.\text{J0FR } |s_2| + 2.s_2.\text{GOTOR } |s_3| + 1.s_3$

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

$$s(pc) = \text{GOTOR } i$$

$$(os, pc) \Rightarrow_s (os, pc + i)$$

$$s(pc) = \text{J0FR } i$$

$$(true.os, pc) \Rightarrow_s (os, pc + 1)$$

$$s(pc) = \text{J0FR } i$$

$$(false.os, pc) \Rightarrow_s (os, pc + i)$$

Example

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

$$\begin{aligned} (\langle \rangle, 0) &\xrightarrow{s} (\langle 2 \rangle, 1) \xrightarrow{s} (\langle \text{true}, 2 \rangle, 2) \xrightarrow{s} (\langle \text{false}, \text{true}, 2 \rangle, 3) \xrightarrow{s} \\ &(\langle \text{true}, 2 \rangle, 4) \xrightarrow{s} (\langle 2 \rangle, 5) \xrightarrow{s} (\langle 1, 2 \rangle, 6) \xrightarrow{s} (\langle 2, 1, 2 \rangle, 7) \xrightarrow{s} \\ &(\langle 3, 2 \rangle, 8) \xrightarrow{s} (\langle 3, 2 \rangle, 12) \xrightarrow{s} (\langle 6 \rangle, 13) \end{aligned}$$

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 <i>true</i>	1
LDCB <i>false</i>	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

$$s(pc) = \text{GOTO } i$$

$$(os, pc) \Rightarrow_s (os, i)$$

$$s(pc) = \text{JOF } i$$

$$(true.os, pc) \Rightarrow_s (os, pc + 1)$$

$$s(pc) = \text{JOF } i$$

$$(false.os, pc) \Rightarrow_s (os, i)$$

Implementation of New Instructions

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

simPLd

x

$\text{fun } x_1 \dots x_n \rightarrow E \text{ end}$

E

$E \quad E_1 \quad \dots \quad E_n$

$(E \; E_1 \dots E_n)$

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: _____
 $x \hookrightarrow \text{LDS } x$

Execution of Identifiers

$$s(pc) = \text{LDS } x$$

$$(os, pc, e) \Rightarrow_s (e(x).os, pc + 1, e)$$

Compilation of Function Application

$$E \hookrightarrow s \quad E_1 \hookrightarrow s_1 \dots E_n \hookrightarrow s_n$$

$$(E \; E_1 \dots E_n) \hookrightarrow s.s_1\dots.s_n.\text{CALL } n$$

Compilation of Function Definition

$$E \hookrightarrow s$$

fun $x_1 \dots x_n \rightarrow E$ end \hookrightarrow LDFS $x_1 \dots x_n$.GOTOR $|s| + 2.s.RTN$

Execution of Function Definition

$$s(pc) = \text{LDFS } x_1 \dots x_n$$

$$(os, pc, e) \Rightarrow_s ((pc + 2, x_1 \dots x_n, e).os, pc + 1, e)$$

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

Execution of Function Application

How about this:

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

$$\begin{aligned} & (v_n \dots v_1.(address, x_1 \dots x_n, e').os, pc, e) \\ \Rightarrow_s & (os, address, e'[x_1 \leftarrow v_1] \dots [x_n \leftarrow v_n]) \end{aligned}$$

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

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

$$\begin{aligned} & (v_n \dots v_1.(address, x_1 \dots x_n, e').os, pc, e, rs) \Rightarrow_s \\ & (\langle \rangle, address, e'[x_1 \leftarrow v_1] \dots [x_n \leftarrow v_n], (pc + 1, os, e).rs) \end{aligned}$$

Returning from a Function

$$s(pc) = \text{RTN } n$$

$$(v.os, pc, e, (pc', os', e').rs) \Rightarrow_s (v.os', pc', e', rs)$$

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 NUMBEROFTARGETS;  
    public CALL(int noa) {  
        OPCODE = OPCODES.CALL;  
        NUMBEROFTARGETS = 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.NUMBEROFGARGUMENTS;
                      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

$$E \hookrightarrow s$$

```
recfun f x1 ... xn -> E end  $\hookrightarrow$ 
LDRFS f x1 ... xn.GOTOR |s| + 2.s.RTN
```

Execution of SVMLe

$$s(pc) = \text{LDRFS } f \ x_1 \dots x_n$$

$$(os, pc, e) \Rightarrow_s ((pc + 2, f, x_1 \dots x_n, e).os, pc + 1, e)$$

Execution of SVMLe, continued

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

$$\begin{aligned} & (v_n. \dots . v_1.(address, f, x_1 \dots x_n, e').os, pc, e, rs) \xrightarrow{s} \\ & (\langle \rangle, address, \\ & e'[f \quad \leftarrow \quad (address, f, x_1 \dots x_n, e')][x_1 \quad \leftarrow \\ & v_1] \dots [x_n \leftarrow v_n], \\ & (pc + 1, os, e).rs) \end{aligned}$$

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.NUMBEROFTARGETS;
                          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