CS4215—Programming Language Implementation

Martin Henz and Chin Wei Ngan

Sunday 8th January, 2017

Chapter 8

Virtual Machines

8.1 Motivation

The semantic frameworks that we have seen so far suffer from two drawbacks. Firstly, they rely on complex mathematical formalism, and secondly, they do not properly account for the space and time complexity of programs.

Complex mathematical formalism Our improved understanding of the language simPL with respect to parameter passing, identifier scoping and error handling was achieved by employing a considerable mathematical machinery, using substitution for *dynamic semantics* and complex semantic domains for *denotational semantics*. In their implementation, we made heavy use of Java/OCaml. We used classes with member functions, functions, recursion etc. Such an approach is questionable; we explained the high-level language simPL by using either a complex mathematical construction or another high-level programming language, Java/OCaml. Worse: in our denotational semantics, we use conditionals in Java/OCaml in order to define conditionals in simPL, recursion in Java in order to define recursion in simPL etc. How are we going to explain Java/OCaml? By reduction to another high-level programming language?

Lack of realism The substitution operation that we employed in *dynamic semantics* is far away from what happens in real programming systems. We can therefore not hope to properly account for the space and time complexity of programs using dynamic semantics. The aim in *denotational semantics* is to describe the meaning of programs as mathematical values, and not as a process, and therefore denotational semantics would have to be significantly modified to account for the resources that executing programs consume.

In this chapter, we are aiming for a simpler, lower-level description of the meaning of simPL programs, which will allow us to realistically capture the runtime of programs and some aspects of their space consumption. To this aim,

we are going to translate simPL to a machine language. We will formally specify a machine for executing machine language code, and describe its implementation in Java (or OCaml).

In order to explain the virtual-machine-based implementation of simPL, we are taking an approach similar to the previous chapter, introducing the machine step-by-step for sublanguages of simPL. This allows us to concentrate on the individual constructs and not get lost in the complexity of the resulting machine for full simPL.

- simPLa is a calculator language similar to simPL0 (Sections 8.2 through 8.5.1);
- simPLb adds division (Section 8.6);
- simPLc adds conditionals (Section 8.7);
- simPLd adds function definition and application (Sections 8.8 and 8.8.2);
- simPLe adds recursive function definition (Section 8.9).

Section 8.10 gives an alternative meaning to some recursive function calls that optimizes on the re-use of call stack. In each of these sections, we describe the concepts using mathematical notation, as well as in terms of two implementations using Java and OCaml, respectively.

Finally, Section 8.11 describes the overall process of executing simPL programs using a virtual machine in terms of T-diagrams.

8.2 The Language simPLa

The language simPLa is defined by the following rules.

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

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

$$\frac{E}{p[E]} \text{ if } p \in \{\texttt{\textbackslash}, \texttt{\~{}}\}.$$

So far, our semantic frameworks relied on the ability to call functions. That allowed us to define the semantics of addition by equations of the form

$$\frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{\texttt{+}[E_1, E_2] \rightarrowtail v_1 + v_2}$$

More specifically, we relied on the ability to remember to evaluate E_2 after evaluating E_1, and then to add the results together. Our high-level notation hid these details.

The goal of this chapter is to present a framework, in which a simple machine suffices to execute programs, which will force us to make explicit how we remember things.

In order to implement simPL in such a low-level setting, we first compile the given expression to a form that is amenable to the machine. We call the result of the compilation *simPL virtual machine code*. The language containing all simPL virtual machine code programs is called *SVML*.

For each of the sublanguages simPLa through simPLe, we will introduce a corresponding machine language SVMLa through SVMLe, respectively.

8.3 The Machine Language SVMLa

SVMLa programs consist of sequences of machine instructions, terminated by the special instruction `DONE`.

SVMLa is defined by the rules of this section.

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

The first rule states that `DONE` is a valid SVML program. The operator . in the second and third rules denotes the concatenation of instruction sequences. In the second and third rules, i stands for elements of the ring of integers, and b stands for elements of the ring of booleans, respectively. The letters `LDCI` in the machine instruction `LDCI` n stand for "LoaD Constant Integer". The letters `LDCB` in the machine instruction `LDCB` b stand for "LoaD Constant Boolean".

The remaining ten rules introduce machine instructions corresponding to each of the operators in simPLa.

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

$$\frac{s}{\texttt{OR}.s} \qquad \frac{s}{\texttt{NOT}.s} \qquad \frac{s}{\texttt{NEG}.s} \qquad \frac{s}{\texttt{LT}.s} \qquad \frac{s}{\texttt{GT}.s} \qquad \frac{s}{\texttt{EQ}.s}$$

To clarify that we are dealing with SVML programs, we are separating instructions with semi-colons and enclosing instruction sequences in brackets.

Example 8.1 *The instruction sequence*

$$[\textit{LDCI}\ 1; \textit{LDCI}\ 2; \textit{PLUS}; \textit{DONE}]$$

represents a valid SVMLa program.

8.3.1 Java Implementation

In our Java implementation, we represent instructions as instances of classes, which implement the `INSTRUCTION` interface.

```
public class INSTRUCTION implements Serializable {
   public int OPCODE;
}
```

Each `INSTRUCTION` carries an `OPCODE` that uniquely identifies its class. For example, the class `LDCI` looks like this.

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

For convenience, we store the opcodes in a class `OPCODES`.

```
public class OPCODES {
  public static final byte
      LDCI     = 1,
      LDCB     = 2,
      ...
}
```

Example 8.2 *Now, we can create the instruction sequence in Example 8.1 as follows:*

```
INSTRUCTION[] ia = new INSTRUCTION[4];
ia[0] = new LDCI(1);
ia[1] = new LDCI(2);
ia[2] = new PLUS();
ia[3] = new DONE();
```

8.3.2 OCaml Implementation

In the OCaml implementation, our instructions are represented using:

```
type sVML_inst =
  | LDCI of int
  | LDCB of bool
  | PLUS | MINUS | TIMES
  | DIV | AND | NEG | NOT
  | OR | LT | GT | EQ | DONE
```

Example 8.3 *We can create the instruction sequence in Example 8.1 in OCaml as follows:*

```
let ia = Array.of_list [LDCI 1;LDCI 2;PLUS;DONE]
```

8.4 Compiling simPLa to SVMLa

The translation from simPLa to SVMLa is accomplished by a function

$$\twoheadrightarrow\colon \text{simPLa} \to \text{SVML}$$

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

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

The auxiliary translation function $\hookrightarrow$ is defined by the following rules.

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

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

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1\texttt{\&}E_2 \hookrightarrow s_1.s_2.\texttt{AND}} \qquad \frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1\,\texttt{|}\,E_2 \hookrightarrow s_1.s_2.\texttt{OR}} \qquad \frac{E \hookrightarrow s}{\backslash\ E \hookrightarrow s.\texttt{NOT}}$$

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1\texttt{<}E_2 \hookrightarrow s_1.s_2.\texttt{LT}} \qquad \frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1\texttt{>}E_2 \hookrightarrow s_1.s_2.\texttt{GT}}$$

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1\texttt{=}E_2 \hookrightarrow s_1.s_2.\texttt{EQ}} \qquad \frac{E \hookrightarrow s}{\sim E \hookrightarrow s.\texttt{NEG}}$$

Example 8.4 *Using the usual derivation trees, we can show* `(1 + 2) * 3` $\twoheadrightarrow$ [*LDCI 1;LDCI 2;PLUS;LDCI 3;TIMES;DONE*]*, and* `1 + (2 * 3)` $\twoheadrightarrow$ [*LDCI 1;LDCI 2;LDCI 3;TIMES;PLUS;DONE*].

Observe that the machine code places the operator of an arithmetic expression after its arguments. This way of writing expressions is called postfix

notation, because the operators are placed after their arguments. It is also called Reverse Polish Notation, because it is the reverse of the prefix notation, which is also called Polish Notation in honor of its inventor, the Polish logician Jan Lukasiewicz.

Our compiler for simPL translates a given simPL expression—as usual represented by its syntax tree—to an `INSTRUCTION` array.

```
Expression simpl=Parse.fromFileName(simplfile);
INSTRUCTION ia[] = Compile.compile(simpl));
```

For OCaml implementation, we can first parse the file, perform type-checking (with extra type information added), followed by pre-processing, prior to compilation to machine instructions, as shown below.

```
let (s,p) = parse_file simplfile in
let (v,np) = type_infer [] p in
let np = trans_exp np in
let ia = compile np in
  ...
```

8.5 Executing SVMLa Code

The machine that we will use to execute SVMLa programs is a variation of a *push-down automaton*. Let us fix a specific program s. The machine M_s that executes s is given as an automaton that transforms a given machine state to another state. The machine state is represented by so-called registers. In the case of SVMLa, we need two registers, called *program counter*—denoted by the symbol *pc*—and *operand stack* —denoted by the symbol *os*.

The program counter is used to point to a specific instruction in s, starting from position 0. For example, if $pc = 2$, and s is the program [`LDCI 1`; `LDCI 2`; `PLUS`; `LDCI 3`; `TIMES`; `DONE`], then $s(pc) =$ `PLUS`.

The operand stack is a sequence of values from **Int** + **Bool**. We will use angle brackets for operand stacks to differentiate them from SVML programs. For example, $os = \langle 10, 20, \mathit{true}\rangle$ represents an operand stack with 10 on top, followed by 20, followed by *true*.

Now, we can describe the behavior of the machine M_s as a transition function $\rightrightarrows_s$, which transforms machine states to machine states, and which is defined by the following twelve rules.

$$\frac{s(pc) = \texttt{LDCI}\ i}{(os, pc) \rightrightarrows_s (i.os, pc + 1)} \qquad \frac{s(pc) = \texttt{LDCB}\ b}{(os, pc) \rightrightarrows_s (b.os, pc + 1)}$$

These load instructions simply push their value on the operand stack. The remaining rules implement the instructions corresponding to simPLa's operators. They pop their arguments from the operand stack, and push the result of the operation back onto the operand stack.

$$\frac{s(pc) = \texttt{PLUS}}{(i_2.i_1.os, pc) \rightrightarrows_s (i_1 + i_2.os, pc + 1)} \qquad \frac{s(pc) = \texttt{MINUS}}{(i_2.i_1.os, pc) \rightrightarrows_s (i_1 - i_2.os, pc + 1)}$$

Note that the `MINUS` instruction subtracts the top element of the stack from the element below, because the subtrahend will be the most recently computed value and therefore appears on top of the stack, whereas the minuend has been computed before the subtrahend, and thus appears below it on the stack.

With this in mind, the remaining rules are straightforward.

$$\frac{s(pc) = \texttt{TIMES}}{(i_2.i_1.os, pc) \rightrightarrows_s (i_1 \cdot i_2.os, pc + 1)} \qquad \frac{s(pc) = \texttt{AND}}{(b_2.b_1.os, pc) \rightrightarrows_s (b_1 \wedge b_2.os, pc + 1)}$$

$$\frac{s(pc) = \texttt{OR}}{(b_2.b_1.os, pc) \rightrightarrows_s (b_1 \vee b_2.os, pc + 1)} \qquad \frac{s(pc) = \texttt{NOT}}{(b.os, pc) \rightrightarrows_s (\neg b.os, pc + 1)}$$

$$\frac{s(pc) = \texttt{LT}}{(i_2.i_1.os, pc) \rightrightarrows_s (i_1 < i_2.os, pc + 1)} \qquad \frac{s(pc) = \texttt{GT}}{(i_2.i_1.os, pc) \rightrightarrows_s (i_1 > i_2.os, pc + 1)}$$

$$\frac{s(pc) = \texttt{EQ}}{(i_2.i_1.os, pc) \rightrightarrows_s (i_1 \equiv i_2.os, pc + 1)} \qquad \frac{s(pc) = \texttt{NEG}}{(b.os, pc) \rightrightarrows_s (0 - b.os, pc + 1)}$$

Note that the behavior of the transition function is entirely determined by the instruction, to which pc points. Like the dynamic semantics $\rightarrowtail$ of simPL, the evaluation gets stuck if none of the rules apply.

The starting configuration of the machine is the pair $(\langle\rangle, 0)$, where $\langle\rangle$ is the empty operand stack. The end configuration of the machine is reached, when $s(pc) = \texttt{DONE}$. The result of the computation can be found on top of the operand stack of the end configuration. The result of a computation of machine M_s is denoted by $R(M_s)$ and formally defined as

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

Example 8.5 *The following sequence of states represents the execution of the SVML program* [`LDCI` 10; `LDCI` 20; `PLUS`, `LDCI` 6; `TIMES`; `DONE`].

$$(\langle\rangle, 0) \rightrightarrows (\langle 10\rangle, 1) \rightrightarrows (\langle 20, 10\rangle, 2) \rightrightarrows (\langle 30\rangle, 3) \rightrightarrows (\langle 6, 30\rangle, 4) \rightrightarrows (\langle 180\rangle, 5)$$

At this point, the machine has reached an end configuration, because $s(5) =$ `DONE`. *The result of the computation is therefore* 180.

8.5.1 Implementing a VM for simPLa in Java

The following Java program shows the general structure of our machine. It consists of a `while` loop, which contains a `switch` statement for executing instructions. The registers *pc* and *os* are represented by Java variables `pc` and `os` to which the interpreter loop has access.

```
public class VM {
   public static Value run(INSTRUCTION[] instructionArray) {
      int pc = 0;
      Stack os = new Stack();
      loop:
      while (true) {
         INSTRUCTION i = instructionArray[pc];
         switch (i.OPCODE) {
            case OPCODES.LDCI:    os.push(new IntValue(i.VALUE));
                                   pc++;
                                  break;
            case OPCODES.PLUS:    os.push(new IntValue(
                                          os.pop().value +
                                          os.pop().value));
                                   pc++;
                                  break;
            case OPCODES.DONE:    break loop;
         }
      }
      return os.pop();
  }
}
```

The instruction `DONE` breaks the loop, after which the top of the operand stack is returned as the result of the program.

8.5.2 Implementing a VM for simPLa in OCaml

The following OCaml program shows the general structure of our virtual machine. It consists of three components: (i) a program counter, named `pc`, a data stack, `stk` and an array of instruction `instArr`. Our main procedure, called `execute`, would repeatedly fetch the next instruction to execute, until `DONE` is encountered.

```
class sVML (instSeq:sVML_inst_mc list) =
object (mc)
  val mutable pc = 0
  val stk = Stack.create ()
  val instArr = Array.of_list instSeq
  (* method to check if next inst is DONE *)
  method finish : bool =
```

```
    let c = Array.get instArr pc in
    c == DONE
  (* method to execute one step *)
  method oneStep : unit =
    let c = Array.get instArr pc in
    proc_inst stk c;
    pc <- pc+1
  method execute : value_mc =
    if mc # finish then Stack.pop stk
    else begin mc # oneStep; mc # execute end
end;;
```

The oneStep method uses a match instruction to perform a wide range of machine instructions, as illustrated below:

```
let proc_inst stk (c:sVML_inst_mc) : unit =
  match c with
    | LDCI i -> Stack.push (VInt i) stk
    | LDCB b -> Stack.push (VBool b) stk
    | PLUS | MINUS | TIMES | AND
    | OR | GT | LT | EQ ->
       let a2 = Stack.pop stk in
       let a1 = Stack.pop stk in
       Stack.push (binary_operate c a1 a2) stk
    | NEG | NOT ->
       let a1 = Stack.pop stk in
       Stack.push (unary_operate c a1) stk
```

We group the instructions into either loading instructions, binary operations or unary operations. Loading instructions pushes a value on the stack, while binary instructions pop two values from the stack before performing its binary operations whose result it pushed back on the operand stack. Unary instructions do the same for single-operand operations.

8.6 A Virtual Machine for simPLb

The language simPLb adds the primitive operator division to the language. In order to handle division by zero, we add $\perp$ as possible stack value. Division by zero will then push $\perp$ on the stack, and jump to `DONE`.

Observe that `DONE` is always at the end of a given program s. In other words, $s(|s|-1) = \texttt{DONE}$. Thus, we can formulate the rules for division as follows:

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

In our Java implementation, we can just break the loop when encountering the divisor 0. The method `run` will then return the `Error` value.

```
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;
                      }
```

A corresponding OCaml implementation is illustrated below: When a zero divisor is encountered, we push an error value, denoted by `BOT` into the stack, prior to an exception that will break our main machine's execution loop.

```
| DIV ->
      begin
      let a2 = Stack.pop stk in
      match a2 with
        | VInt 0 ->
              Stack.push BOT stk;
              failwith "Divide by Zero"
        | _ ->
              let a1 = Stack.pop stk in
              Stack.push (binary_operate c a1 a2) stk
      end
```

8.7 A Virtual Machine for simPLc

The language simPLc adds conditionals to simPLb. Conditionals involve jumping from one part of the program to another. How can we jump in our machine? The obvious answer: by setting the program counter to the index of the jump target. Indices pointing to instructions inthe instruction sequence are called *addresses*. The actual addresses would typically be integer values that capture the actual location of the code. To facilitate compilation, we shall also use an intermediate form that uses strings as address labels. To obtain unique address labels, we also provide a unique name generator for labels. We also add a dummy instruction that can capture the symbolic location of some given program point. In OCaml, this is written as LABEL~l where l is a symbolic representation in string form of an address.

In order to implement conditionals, we add the instructions `GOTO` ("GOTO") and `JOF` (Jump On False) to our instruction set. Both instructions carry with

them an address, by which the program counter is to be set to. The instruction `GOTO` l changes the program counter by the address of l, whereas `JOF` l changes the program counter by l only if the top of the operand stack (that will also be popped out) is *false*.

The translation of conditionals is as follows.

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

Example 8.6 *The translation function translates the simPLc expression*

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

to the following instruction sequence.

[*LDCI* 2; *LDCB true*; *LDCB false*; *OR*; *JOF* l_1; *LDCI* 1; *LDCI* 2; *PLUS*; *GOTO* l_2; *LABEL* l_1; *LDCI* 2; *LDCI* 3; *PLUS*; *TIMES*; *LABEL* l_2; *DONE*]

The symbolic addresses will be eventually translated into integer form at the end of compilation. This is done by collecting the absolute locations of the dummy label instructions. These dummy labels may be removed, prior to code execution. For our example, assuming that the code fragment starts from location 0, we can initially determine that labels l_1 and l_2 are at locations 9 and 13, respectively.

Example 8.7 *Using the newly computed absolute locations in integer form, the earlier code in Example 8.6 can be transformed to the following instruction sequence, where the code addresses are indicated for clarity.*

[*LDCI* 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*
TIMES	*12*
DONE]	*13*

The executions of `JOF` and `GOTO` are defined as follows.

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

Example 8.8 *The following state sequence represents the execution of the program in the previous example.*

$(\langle\rangle, 0) \rightrightarrows_s (\langle 2\rangle, 1) \rightrightarrows_s (\langle true, 2\rangle, 2) \rightrightarrows_s (\langle false, true, 2\rangle, 3) \rightrightarrows_s (\langle true, 2\rangle, 4) \rightrightarrows_s (\langle 2\rangle, 5) \rightrightarrows_s (\langle 1, 2\rangle, 6) \rightrightarrows_s (\langle 2, 1, 2\rangle, 7) \rightrightarrows_s (\langle 3, 2\rangle, 8) \rightrightarrows_s (\langle 3, 2\rangle, 12) \rightrightarrows_s (\langle 6\rangle, 13)$
The last state is an end configuration, and thus the result is 6.

We add the following two cases to the Java implementation of the virtual machine.

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

A corresponding OCaml implementation at the point where the machine instructions are dispatched would be the following:

```
| GOTO i -> pc<-i
| JOF i -> let a = Stack.pop stk in
           let b = get_bool a in
           if b then pc<-pc+1
           else pc<-i
```

8.8 A Virtual Machine for simPLd

The language simPLd adds identifiers, function definition and application to simPLc. Note that we revert to reducing the `let` construct to function definition and application and thus avoid its treatment here.

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

These constructs are by far the most challenging aspects of simPL from the point of view of the virtual machine. We shall describe the compilation of these constructs and the execution of the corresponding SVML instructions step by step in the following paragraphs. Along the way, we shall introduce the instructions of the corresponding machine language SVMLd. Section 8.8.2 discusses our virtual machine implementation.

Compilation of Identifiers Similar to the approach of the previous chapter, we implement identifiers by environments. To this aim, we add a register e to the machine state. Register e represents the environment with respect to which the identifiers are executed. As usual, environments map identifiers to

denotable values. Thus an environment e, in which `x` refers to the integer 1 can be accessed by applying e to `x`, $e(\texttt{x}) = 1$.

Occurrences of identifiers in simPLd are translated to instructions `LDS` x (LoaD Symbolic).

$$\frac{}{x \hookrightarrow \texttt{LDS}\ x}$$

Execution of Identifiers The execution of identifier occurrences pushes the value to which the identifier refers on the operand stack. Thus, the rule specifying the behavior of `LDS` x is as follows.

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

Note that the state of our machine now has an additional component, the environment e. Recall that this is a mapping of identifier to values. In order to reduce the cost of lookup for identifiers in this environment, we will later compile each identifier to its corresponding relative locations of the current environment. This current environment is sometimes also referred to as the activation record of the method call. As we are using a higher-order programming language, we shall be capturing free (non-local) variables of each method into its corresponding activation record.

Compilation of Function Application A function application is translated by translating operator and operands, and adding a new instruction `CALL` n, which remembers the number of arguments n of the application.

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

Take note that we are compiling the arguments in reverse order, and will be placing the code s for the function E at the top of the stack, after its operands are placed into the stack. Thus, the instruction `CALL` n will always find the operator (function) and then its operands on the topmost portion of the operand stack.

As we support functions that returns functions as results, there may be scenarios where a function application has more arguments than is actually needed. An example of this is illustrated below:

```
fun {int -> int -> int -> int} x y ->
   fun {int -> int} z -> y + z - x end
end 7 8 9
```

Though the outer function has only two parameters, the application itself, that will be compiled into CALL 3, would actually be having three arguments.

We can detect such over-applied applications by looking at the arity of the operator, and then taking note of the number of residuals arguments left in the operand stack. These residual arguments would have to be applied to the function result that are returned by our earlier over-applied application. We shall handle such residual applications through the return instruction for method definitions, which will be described next.

Compilation of Function Definition Function definition needs to create a function value, which will have a reference to the code to which the function body is translated. In addition, the function definition needs to remember the names of its formal parameters and also the non-local variables used in the method in order to support lexical scoping. The function definition is represented in SVML code by the instruction `LDFS` (LoaD Function Symbolic), and translated as follows.

$$\frac{E \hookrightarrow s \qquad v_1 \cdots v_m = fv(E) - \{x_1 \cdots x_n\}}{\texttt{fun}\ x_1 \ldots x_n\ \texttt{->}\ E\ \texttt{end} \hookrightarrow \texttt{LDFS}\ [v_1 \cdots v_m][x_1 \cdots x_n].\texttt{GOTO}\ l.s.\texttt{RTN.LABEL}\ l}$$

Execution of the instruction `LDFS` will push a function value (that includes an environment for the non-local variables used) and then jump to the code after the function body. In between is the code of the function body, followed by a `RTN` instruction, which indicates that the called function ReTurNs to the caller.

Execution of Function Definition According to static (or lexical) scoping, the function body needs to be executed with respect to the environment of the function definition for its non-local variables. Thus, function definition needs to push a function value, which remembers the code address of the body, the formal parameters and the environment.

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

The operation $e\#[v_1 \cdots v_m]$ will select a subset of the environment for just the non-local variables $v_1 \cdots v_m$. Such a triple $(address, formals, env)$ is called a *closure* in the context of virtual machines.

Execution of Function Application According to the translation of function application, the instruction `CALL` n will find its operator, followed by its arguments on the operand stack, followed by the operator, which—according the the previous paragraph—is represented by a closure. To implment static scoping, the machine must take the environment of the closure, and extend it by a binding of the formal parameters to the actual arguments. Thus, the following rule is our first attempt to describe the execution of `CALL` n.

$$\frac{s(pc) = \texttt{CALL}\ n}{\begin{array}{l}((address, x_1 \cdots x_n, e').v_1 \ldots . v_n.os, pc, e) \quad \rightrightarrows_s \\ (os, address, e'[x_1 \leftarrow v_1] \cdots [x_n \leftarrow v_n])\end{array}}$$

There are, however, two major difficulties with this rule. Firstly, what if the closure requires fewer parameters that the application itself. Secondly, what should the machine state be after a function returns? In other words, what should the machine state be when it encounters the instruction `RTN` after executing the function body? In particular, what should be the program counter, operand stack and environment be after returning from a function? Of course, the program counter, operand stack and environment must be restored to their state before the function call.

In order to keep program execution in a simple loop, we need to make this return information explicit. Since functions can call other functions before returning, the natural data structure for this return information is another stack. We call this stack the *runtime stack*, as opposed to the *operand stack* that we have got used to. The runtime stack, denoted by *rs*, will be the fourth and last register that we add to our machine state. Each entry in the runtime stack contains the *address* of the instruction to return to, the environment e, and an integer r which denotes the number of residual arguments that are still left behind in the operand the stack *os*, where further applications are to be applied after the current function returns. Such a triplet $(r, address, e)$ is called *runtime stack frame*, or simply *stack frame*.

Function application pushes a new stack frame on the runtime stack, in addition to the actions described in the first attempt above. Thus, the actual rule for `CALL` n is as follows.

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

Note that $n - m$ denotes that residual arguments (from over-application) that are left on the operand stack that are to be applied when the current method returns. In the case of exact application when $n = m$, there would be no further arguments left on the operand stack.

Returning from a function Now, the instruction `RTN` would do two possible things. It can either return properly from a function by popping a stack frame from the runtime stack and re-installing its content in the other machine registers This can occur when the application has the same number of arguments as its function operator.

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

Alternatively, it could be expected to perform further applications, as below, using the residual arguments that were left behind in the operand stack.

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

We can check this by systematically tracking the number of residual parameters that are being kept the operand stack. This is represented by r that is being kept in the runtime stack (as shown in the above execution rule).

Of course, all other instructions also need to be extended to include a runtime stack, which is not changed by the instruction. For example, the rule for `LDS` becomes:

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

8.8.1 Implementing a VM for simPLd in OCaml

Let us now look at some aspects of an OCanl implementation of the virtual machine for simPLd. We will do that in stages, and also modify the format of the instruction slightly.

Compilation of identifiers Instead of using lookup tables that map identifiers to values, our actual compiler predicts the place where the identifier can be found in the environment. Identifiers are therefore translated to load instructions `LD` (instead of `LDS`) that carry the index where the identifier is expected in the environment. Thus, our load instruction would be implemented, as follows:

```
| LD of string * int
```

For ease of debugging, we also kept the original identifier name that was in string form. As an example, the instruction `LD (x,0)` denotes the identifier `x` that is currently at location `0` in the environment.

Compilation of function definition To avoid the `GOTO` instruction after `LDFS` (see page 16), the code for function bodies can be placed in a different part of the instruction array. Thus, the instruction corresponding to function definition needs to remember the address of the first instruction of the corresponding body. On the other hand, since the compiler predicts all environment

positions of identifiers, there is no need to remember the names of formal parameters. The corresponding `LDF` (LoaD Function) instruction has a simpler format:

```
| LDF of 'sym list * int * 'label
      (* global vars * arity * address *)
```

We keep track of the non-local variables, number of parameters and also the address of the method definition,

Compilation of application Applications remember their number of arguments. The corresponding instruction has the format:

```
| CALL of int
```

Example 8.9 *To illustrate the compilation, let us consider the following simPLd expression.*

```
(fun {int->int->int} x y -> x + y end 3 4)
```

This expression gets translated to the instruction sequence

$$\begin{array}{ll}
[\mathit{LDCI}\ 4 & \mathit{0} \\
\ \mathit{LDCI}\ 3 & \mathit{1} \\
\ \mathit{LDF}([], 2, 5) & \mathit{2} \\
\ \mathit{CALL}\ 2 & \mathit{3} \\
\ \mathit{DONE} & \mathit{4} \\
\ \mathit{LD}(x, 0) & \mathit{5} \\
\ \mathit{LD}(y, 1) & \mathit{6} \\
\ \mathit{PLUS} & \mathit{7} \\
\ \mathit{RTN}] & \mathit{8}
\end{array}$$

Note that the compiler avoids the `GOTO` instruction after `LDF` by placing the code for the body after the `DONE` instruction. `LDF` carries ([],2,5) where [] denotes no non-local variables, 2 denotes it takes two local parameters and 5 denotes the start address of the method body.

Representation of environments Instead of using lookup tables that map identifiers to values, our actual compiler predicts the place where the identifier can be found in the environment. Thus, a vector mapping integers to `Value`s represents environments. The `CALL` instruction needs to fill the closure's environment by as many new slots as the called function has arguments, which is done by the following method which pops the arguments from the operand stack and then placing it at enviroment from position m..(r−1) (after the non-local variables).

```
let pop_2_venv stk env m r =
  let rec aux m =
```

```
    if m=r then ()
    else
      let v = Stack.pop stk in
      let _ = Array.set env m v in
      aux (m+1)
  in aux m
```

The environment register e is represented by an additional OCaml variable venv we introduce for our virtual machine to which the emulation loop has access to.

Execution of identifiers The `LD` instruction simply looks up the `val` stored in the environment under its `index`.

```
  | LD (_,index) ->
         let val = Array.get venv index in
         Stack.push val stk; pc<-pc+1
```

Representation of closures Function definitions must—in addition to the body of the function—keep track of the environment in which the definition was executed. To this aim, we add another case for values, using `CLS` to denote closures, as follows:

```
type value_mc =
  | BOT (* denotes an error *)
  | VInt of int
  | VBool of bool
  | CLS of int * int * var_env
     (* addr * arity * global var values *)
```

Each closure value `CLS(addr,a,e)` captures the address `addr` of method body, the arity `a` on the number of parameters, and environment `e` that captures the non-local variables used.

Execution of function definition At runtime, `LDF` simply puts together a closure data structure, consisting of the address of the function, its arity and the current environment; and pushes it on the operand stack.

```
  | LDF(gvs,a,l) ->
      let gvs_len = List.length gvs in
      let e = Array.init (gvs_len+a) ..
      Stack.push (CLS (l,a,e)) stk;
      pc <- pc+1
```

Representing runtime stack frames The instructions `CALL` and `RTN` form a pair. In order to be able to return from function application, the number of residual parameters, current program counter and the current environment, and the need for them to be saved in a runtime stack frame.

```
(ref int * int * (value_mc array) Stack.t
```

The runtime stack register *rs* is represented by the additional OCaml mutable stack value `rs` to which the interpreter loop has access.

Execution of application The instruction `CALL` takes the callee function's environment out of its closure and extends it by bindings of the arguments. Then it pushes a new frame on the runtime stack, saving the current register values (after incrementing *pc* by 1 to make it point to the next instruction) for the return from the function. Finally it sets the registers for the execution of the function body.

```
| CALL n ->
   begin
    match Stack.pop stk with
     | CLS(addr,s,e) ->
        let m = Array.length e in
        let r = m+s in
        let e2= Array.init r (fun i -> if i<m then
                  Array.get e i else BOT) in
        let _ = pop_2_venv stk e2 m r in
        let _ = Stack.push (ref (n-s),pc+1,venv) rs in
        let _ = venv <- e2 in
        pc <- addr
     | _ -> failwith "CALL : not a function"
   end
```

Take note that it expects a closure on the top of the operand stack whose arity `a` may be less the number of arguments of the call. (After our pre-processing, re-call that we allow over-applied applications but never under-applied applications.) As only `a` parameters are popped from the operand stack, we are left with `n-a` as the number `m` of residual arguments left on the operand stack. We use a mutable reference ref(n−a) to mark the residual arguments that would later be consumed through the `RTN` instruction. Mutable reference is being used for efficient local updates to this number of residual arguments, without the need to pop it from the run-time stack.

Returning from a function The `RTN` instruction is meant to pop the most recently saved frame from the runtime stack and reinstalls its components in the respective registers.

```
| RTN ->
   let (n_ref,npc,ne) = Stack.top rs in
    let n = !n_ref in
    if n=0 then (Stack.pop rs; pc<-npc; venv<-ne)
    else
     begin
```

```
      match Stack.pop stk with
       | CLS(addr,s,e) ->
          let m = Array.length e in
          let r = m+s in
          let e2= Array.init r (fun i -> if i<m then
                Array.get e i else BOT) in
          let _ = pop_2_venv stk e2 m r in
          let _ = n_ref := !n_ref-s in
          let _ = venv <- e2 in
          pc <- addr
       | _ -> failwith "RTN 0<s<=n : not a closure"
     end
```

However, as we may have residual arguments on the operand stack, we may have to perform further applications when n>0. When n=0, the `RTN` instruction pops the previous program counter and the previous environment back into our machine's registers.

8.8.2 Implementing a VM for simPLd in Java

Similar to the previous section, we are going to look at the implementation aspects of the virtual machine for simPLd step by step.

Compilation of identifiers Instead of using lookup tables that map identifiers to values, our actual compiler predicts the place where the identifier can be found in the environment. Identifiers are therefore translated to load instructions `LD` that carry the index where the identifier is expected in the environment.

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

Compilation of function definition To avoid the `GOTO` instruction after `LDF` (see page 16), the code for function bodies is placed to a different part of the `INSTRUCTION` array. Thus, the instruction corresponding to function definition needs to remember the address of the first instruction of the corresponding body. On the other hand, since the compiler predicts all environment positions of identifiers, there is no need to remember the names of formal parameters. The corresponding `LDF` (LoaD Function) class follows.

```
public class LDF extends INSTRUCTION {
   public int ADDRESS;
   public LDF(int address) {
```

```
      OPCODE = OPCODES.LDF;
      ADDRESS = address;
   }
}
```

Compilation of application Applications remember their number of arguments. The corresponding class follows.

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

Example 8.10 *To illustrate the compilation, let us consider the following simPLd expression.*

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

This expression gets translated to the instruction sequence

$$\begin{array}{ll} [\textit{LDF}\ 4 & \textit{0} \\ \textit{LDCI}\ 2 & \textit{1} \\ \textit{CALL}\ 1 & \textit{2} \\ \textit{DONE} & \textit{3} \\ \textit{LD}\ 0 & \textit{4} \\ \textit{LDCI}\ 1 & \textit{5} \\ \textit{PLUS} & \textit{6} \\ \textit{RTN}] & \textit{7} \end{array}$$

Note that the compiler avoids the `GOTO` *instruction after* `LDF` *by placing the code for the body after the* `DONE` *instruction.*

Representation of environments Instead of using lookup tables that map identifiers to values, our actual compiler predicts the place where the identifier can be found in the environment. Thus, a vector mapping integers to `Value`s represents environments. The `CALL` instruction needs to extend the closure's environment by as many new slots as the called function has arguments, which is done by `Environment`'s `extends` method.

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

The environment register e is represented by the additional Java variable `e` to which the interpreter loop has access.

Execution of identifiers The `LD` instruction simply looks up the `Value` stored in the environment under its `INDEX`.

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

Representation of closures Function definitions must—in addition to the body of the function—keep track of the environment in which the definition was executed. To this aim, we add another `Value` class.

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

Execution of function definition At runtime, `LDF` simply puts together a `Closure` data structure, consisting of the current environment and the address of the function, and pushes it on the operand stack.

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

Representing runtime stack frames The instructions `CALL` and `RTN` form a pair. In order to be able to return from function application, the current environment, the current operand stack and the current program counter need to be saved in a runtime stack frame.

```
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;
  }
}
```

The runtime stack register *rs* is represented by the additional Java variable `rs` to which the interpreter loop has access.

Execution of application The instruction `CALL` takes the callee function's environment out of its closure and extends it by bindings of the arguments. Then it pushes a new frame on the runtime stack, saving the current register values (after incrementing *pc* by 1 to make it point ot the next instruction) for the return from the function. Finally it sets the registers for the execution of the function body.

```
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 The `RTN` instruction pops the most recently saved frame from the runtime stack and reinstalls its components in the respective registers.

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

The `RTN` instruction pops the return value from the old `os` and pushes it onto the new `os`.

8.9 A Virtual Machine for simPLe

The call of recursive functions needs to extend the function environment by a binding of the function variable to the function value. Recursive function definition therefore needs to remember the function variable in the corresponding instruction `LDFRS` (LoaD Recursive Function Symbolic).

$$E \hookrightarrow s \qquad v_1 \cdots v_m = fv(E) - \{f, x_1 \cdots x_n\}$$

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

The execution of LDFRS creates a closure, which contains the name of the function variable, and thus consists of a quadruplet of the form (*address*, *funvar*, *formals*, e).

Correspondingly, we add the following rules for definition and application of recursive functions.

$$s(pc) = \texttt{LDFRS}([v_1 \cdots v_m], f, [x_1 \cdots x_n])$$

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

In our actual implementation, which relies on the compiler to compute the correct environment indices for all identifier occurrences, we employ a trick to achieve a similar effect. Recursive function definition extends the given environment by an entry containing the function itself. That means that the resulting closure is a circular data structure. The compiler generates an LDFR n (LoaD Recursive Function) instruction similar to LDF.

```
| LDFR(gvs,f,vs_len,l) ->
   let gvs_len = List.length gvs in
   let e = Array.init (gvs_len+1)
    (fun i-> if i<gvs_len
     then let g = snd(List.nth gvs i) in
          Array.get venv g
     else BOT) in
   let cls = CLS (l,vs_len,e) in
   Array.set e gvs_len cls; (* circular closure *)
   Stack.push cls stk;
   pc <- pc+1
```

Using this trick, there is no need to change the execution of the CALL instruction. Function calls do not need to distinguish between functions and recursive functions.

For Java implementation, we may have the following codes:

```
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;
```

8.10 Tail Recursion

Each function call creates a new stack frame and pushes it on the runtime stack. Function calls therefore consume a significant amount of memory. There are situations, where the creation of a new stack frame can be avoided.

If the last action in the body of a function is another function call, then the environment, program counter and operand stack of the calling function invocation is not going to be needed upon returning from the function to be called. The function to be called can return to wherever the calling function needs to return.

Furthermore, if the calling function and the function to be called is the same recursive function, then the environment needed by the called invocation is almost identical to the environment of the calling invocation. The difference is the binding of the formal parameters to arguments, which of course can change between recursive calls.

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

Example 8.11 *Consider the following implementation of the factorial function.*

```
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 (fac 4) end
end
```

The recursive call in the body of `facloop` *is the last instruction to be executed by the body. It is a tail call. Therefore, we can re-use the environment of the calling invocation and do not need to push a new stack frame. Since the tail call is the only recursive call of* `facloop`*, the function is tail-recursive.*

We change our compiler so that the instruction sequence [`CALL` n; `RTN`] is replaced by [`TAILCALL` n]. This tail-call optimization can be done in general. However, when the operator of the call is a recursive variable f, the immediately surrounding function definition is recursive and what we will have achieved for tail-recursive optimization is a run-time behaviour that is similar to that of iterative loops.

Without tail-call optimization, our compiler would have generated the following code (with the addresses for labels displayed on the left):

```
      [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]
```

With tail-call optimization, we would have generated the following shorter and more efficient code.

```
      [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]
```

Take note that the last pair of `CALL 2,RTN` at address 30 in the original code is translated to `TAILCALL 2,RTN`, rather than just `TAILCALL 2`, since there is a jump to location 28 from an earlier location.

Tail calls do not change the size of the runtime stack. Instead they replace the current values of the argument variables with the arguments of the new call for the current environment. Unlike the `CALL` instruction which extends the run-time stack, `TAILCALL` does not need to save the current stack frame into the run-time stack. The function that is being called will therefore return directly to the function that called the calling function. As the old environment e is no longer needed, the environment extension operation can be done destructively. As there may be residual arguments from a prior application in the operand stack, our implementation of `TAILCALL` may make a slight change on the number of residual arguments that remains from r to $r + n - s$. Because of the need for this tracking, it is better to use a mutable type for storing this number of residual arguments. Formally, we can capture its semantics, as follows:

$$s(pc) = \texttt{TAILCALL}\ n \qquad s{\leq}r{+}n$$

$$((addr, x_1 \cdots x_s, e_2).v_1 \ldots v_s.os, pc, e, (r, npc, ne).rs) \rightrightarrows_s$$
$$(os, addr, e_2[x_1{\leftarrow}v_1] \cdots [x_s{\leftarrow}v_s], (r{+}n{-}s, npc, ne).rs)$$

The implementation of `TAILCALL` is simpler and much more efficient than the implementation of `CALL` (compare with page 25).

In the Java implementation, this is achieved by:

```
case OPCODES.TAILCALL: { int n = i.NUMBEROFARGUMENTS;
                         Closure closure
                             = os.elementAt(os.size()
                                            -n-1);
```

```
                                int s = e.size();
                                int k = s - n;
                                int j = s - 1;
                                while (j >= k)
                                   e.setElementAt(os.pop(),j--);
                                os.pop();
                                pc = closure.ADDRESS;
                                break;
                              }
```

In the OCaml implementation, this is done using:

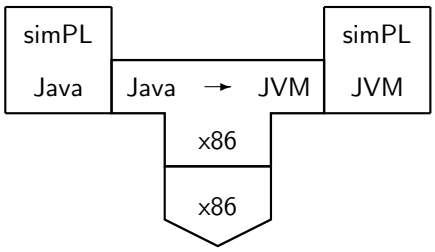
```
| TAILCALL n ->
  begin
    let (n_ref,_,_) = Stack.top rs in
    match Stack.pop stk with
    | CLS(addr,s,e) ->
       let m = Array.length e in
       let r = m+s in
       let e2 = if Array.length venv>=r then
                   (Array.blit e 0 venv 0 m; venv)
                else Array.init r (fun i -> if i<m then
                      Array.get e i else BOT) in
       let _ = pop_2_venv stk e2 m r in
       let _ = n_ref := !n_ref+n-s in
       let _ = venv <- e2 in
       pc <- addr
  end
```

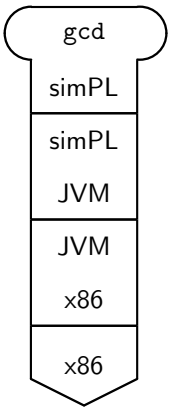
Note that we re-use the current enviroment `venv` if there is sufficient space on it. Our current implementation can be further optimized by using a contiguous array to store all stack frames. This is left as an optimization exercise.

8.11 Compilation and Execution

In our virtual machine based implementation of simPL, we now have two distinct phases, namely compilation to SVML code, and execution of the SVML code by a virtual machine.

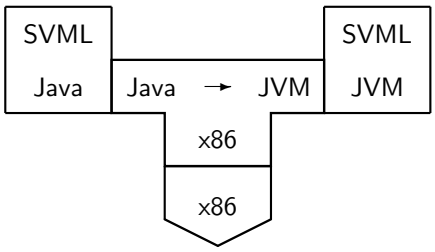
If we choose to directly execute the instructions stored in the instruction array, we can still view the entire execution of simPL program as an interpreter. The interpreter uses compilation, which is an internal detail of its implementation. According to this view, the corresponding T-diagrams are as follows.

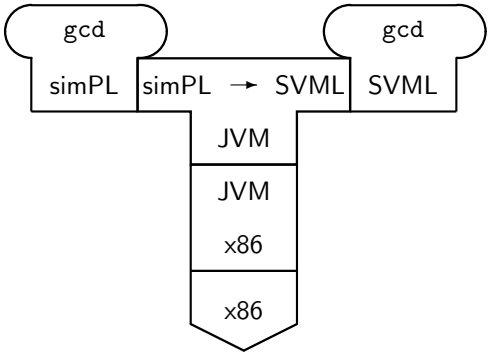


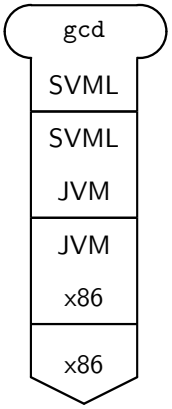


Instead of directly executing the instructions, we can instead store the instruction array in a file (in Java easily done using an `ObjectOutputStream`). This amounts to a simPL compiler, which translates simPL files to SVML files.

The machine loads a given SVML file and executes its SVML code. Thus the machine acts as an emulator for SVML. Since it is implemented in Java, it is running on top of the Java Virtual Machine, as depicted in the following T-diagrams.







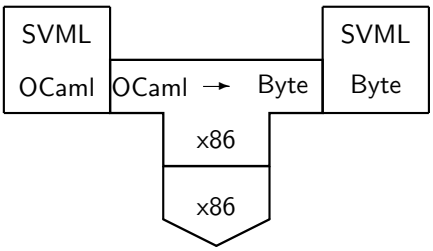
Example 8.12 *Using the compiler* `simplc` *and the emulator* `simpl`*, both written in Java, we can execute a given simPL program* `gcd.simpl` *as follows:*

```
> java simplc gcd.simpl
> ls gcd.*
gcd.simpl gcd.svml
> java simpl gcd
2
```

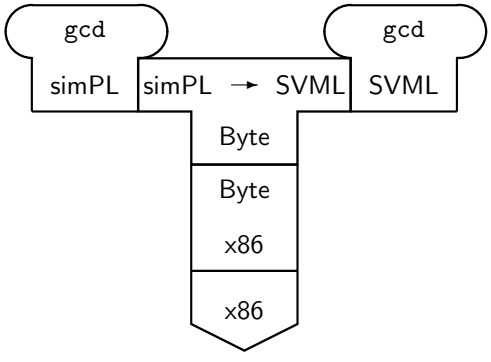
In the OCaml implementation of the SimPL language, we also have two phases:

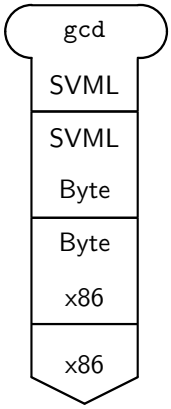
- A compiler, called `splc`, which compiles SimPL programs to corresponding SVML code.
- A virtual machine emulator, called `svm`, that can load instruction in SVML for execution.

We compile the virtual machine, written in OCaml, to an executable bytecode, as shown below:



We can also use a compiler from simPL to SVML, that is written in OCaml but compiled to a bytecode executable. The two phase procedure would first compile to SVML code, and then have it executed by our virtual machine, as illustrated below for a GCD program written in simPL.





Example 8.13 *This two phase procedure on compiling a simPL program (using a compile named splc), and then executing it in a virtual machine emulator, called svm, both written in OCaml, can be used to execute a given simPL program* `gcd.spl` *as follows:*

```
> ./splc gcd.spl
> ls gcd.*
gcd.spl gcd.svm
> ./svm gcd
2
```