CS4215—Programming Language Implementation

Martin Henz

Friday 10 February, 2012

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. We used classes with member 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. Worse: in our denotational semantics, we use conditionals in Java 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? 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.

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 8.6);
- simPLb adds division (Section 8.7);
- simPLc adds conditionals (Section 8.8);
- simPLd adds function definition and application (Sections 8.9 and 8.10);
- simPLe adds recursive function definition (Section 8.11).

Section 8.12 gives an alternative meaning to some recursive function calls. In each of these sections, we describe the concepts using mathematical notation, as well as in terms of an implementation in Java.

Finally, Section 8.13 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.

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

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

$$\frac{E}{p[E]} \text{ if } p \in \{\backslash\}.$$

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.

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

$$\frac{s}{\texttt{OR}.s} \qquad \frac{s}{\texttt{NOT}.s} \qquad \frac{s}{\texttt{LESS}.s} \qquad \frac{s}{\texttt{GREATER}.s} \qquad \frac{s}{\texttt{EQUAL}.s}$$

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

Example 8.1 *The instruction sequence*

$$[\mathit{LDCI}\ 1, \mathit{LDCI}\ 2, \mathit{PLUS}, \mathit{DONE}]$$

represents a valid SVMLa program.

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();
```

8.4 Compiling simPLa to SVMLa

The translation from simPLa to SVMLa is accomplished by a function

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

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.

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

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

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

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

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

Example 8.3 *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));
```

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 $s(pc) = \texttt{PLUS}$.

The operand stack is a sequence of values from $\mathbf{Int} + \mathbf{Bool}$. 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.

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

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

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

$$\frac{s(pc) = \texttt{EQUAL}}{(i_2.i_1.os, pc) \rightrightarrows_s (i_1 \equiv i_2.os, pc + 1)}$$

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 8.4 *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.

8.6 Implementing a Virtual Machine 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.

8.7 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;
                      }
```

8.8 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 in the instruction sequence are called *addresses*.

In order to implement conditionals, we add the instructions `GOTOR` ("GOTO" Relative) and `JOFR` (Jump On False Relative) to our instruction set. Both instructions carry with them an offset, by which the program counter is incremented. The instruction `GOTOR` i always increments the program counter by i, whereas `JOFR` i increments the program counter by i only if the top of the operand stack is *false*.

The translation of conditionals is as follows.

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

Example 8.5 *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*, *JOFR* 5, *LDCI* 1, *LDCI* 2, *PLUS*, *GOTOR* 4, *LDCI* 2, *LDCI* 3, *PLUS*, *TIMES*, *DONE*]

The execution of `JOFR` and `GOTOR` is defined as follows.

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

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

Example 8.6 *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.

The compiler in our Java implementation is able to compute absolute jump addresses instead of letting the machine do the computation. The corresponding instructions are called `GOTO` and `JOF` (Jump On False).

Example 8.7 *Using these new jump instructions, the expression in Example 8.5 is compiled 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 new instructions are defined by the following rules.

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

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

We add the following two cases to the loop of the virtual machine.

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

8.9 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 8.10 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.

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.

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

Thus, the instruction `CALL` n will find the operands of the application in reverse order, followed by the operator, on the operand stack.

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. The function definition is represented in SVML code by the instruction `LDFS` (LoaD Function Symbolic), and translated as follows.

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

Execution of the instruction `LDFS` will push a function value 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 scoping, the function body needs to be executed with respect to the environment of the function definition. Thus, function definition needs to push a function value, which remembers the code address of the body, the formal parameters and the environment.

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

Such a triple $(address, formals, e)$ is called a *closure* in the context of virtual machines.

Execution of Function Application According to the translation of function application, the instruction `CALL` n will find its arguments in reverse order 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.

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

There is, however, a major difficulty with this rule. What should happen when a function returns? In other words, what should the machine do when it encounters the instruction `RTN` after executing the function body? In particular, what should be the program counter, operand stack and environment 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 a stack. We call this stack the *runtime stack*. The runtime stack, denoted by *rs*, will be the forth 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, and the operand

stack os and environment e to be reinstalled after the function call. Such a triplet $(address, os, e)$ 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.

$$\frac{s(pc) = \texttt{CALL}\ n}{\begin{array}{l}(v_n \ldots v_1.(address, x_1 \cdots x_n, e').os, pc, e, rs) \\ \rightrightarrows_s (\langle\rangle, address, e'[x_1 \leftarrow v_1] \cdots [x_n \leftarrow v_n], (pc+1, os, e).rs)\end{array}}$$

Returning from a function Now, the instruction `RTN` can return from a function by popping a stack frame from the runtime stack and re-installing its content in the other machine registers.

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

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

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

8.10 Implementing a Virtual Machine 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 `GOTOR` instruction after `LDFS` (see page 14), 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 8.8 *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

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

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`.

8.11 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 `LDRFS` (LoaD Recursive Function Symbolic).

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

The execution of `LDRFS` 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.

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

$$\frac{s(pc) = \texttt{CALL}\ n}{\begin{array}{l}(v_n \ldots . v_1.(address, f, x_1 \cdots x_n, e').os, pc, e, rs) \rightrightarrows_s \\ (\langle\rangle, address, e'[f \leftarrow (address, f, x_1 \cdots x_n, e')][x_1 \leftarrow v_1] \cdots [x_n \leftarrow v_n], (pc + 1, os, e).rs)\end{array}}$$

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 `LDRF` n (LoaD Recursive Function) instruction similar to `LDF`, which is executed as follows.

```
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;
```

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.

8.12 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 call*. A recursive function, in which all recursive calls are tail calls, is called *tail-recursive*.

Example 8.9 *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
   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, when the operator of the call is a variable f, the immediately surrounding function definition is recursive and has the function variable f.

Example 8.10 *Our modified compiler generates the following instructions for the body of* `facloop`*.*

[*LD* 1	*4*
LDCI 1	*5*
EQUAL	*6*
JOF 10	*7*
LD 2	*8*
RTN	*9*
LD 0	*10*
LD 1	*11*
LDCI 1	*12*
MINUS	*13*
LD 2	*14*
LD 1	*15*
TIMES	*16*
TAILCALL 2]	*17*

Tail calls do not need to manipulate the runtime stack. Instead they replace the current values of the argument variables with the arguments of the new call.

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

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

Note that `TAILCALL` does not save any stack frame. The function that is being called will therefore return directly to the function that called the calling function. The old environment e is not needed any longer. The environment extension operation can therefore be destructive.

The implementation of `TAILCALL` is simpler and much more efficient than the implementation of `CALL` (compare with page 18).

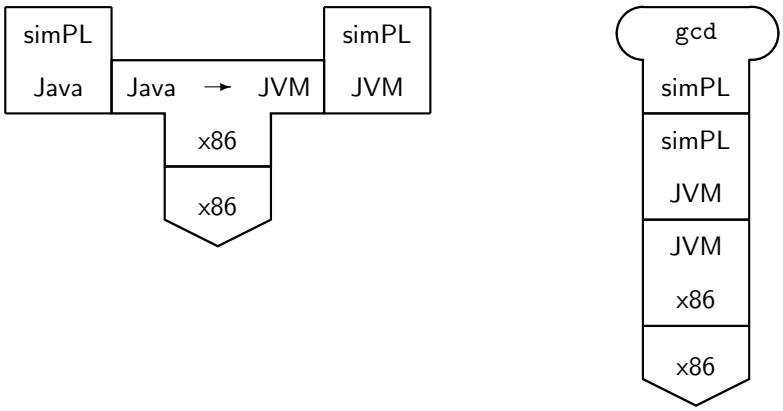
```
case OPCODES.TAILCALL: { int n = i.NUMBEROFARGUMENTS;
                         Closure closure
                             = os.elementAt(os.size()
                                            -n-1);
                         int s = e.size();
                         int k = s - n;
             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

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

8.13 Compilation and Execution

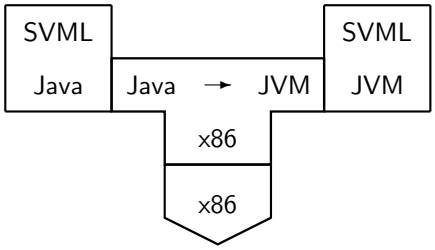
In our virtual machine based implementation of simPL, we now have two distinct phases, namely compilation to SVML code, and execution of the SVML code by a virtual machine.

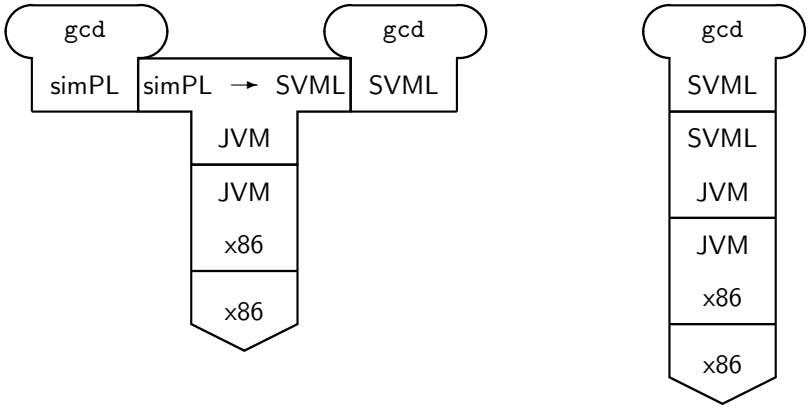
If we choose to directly execute the instructions stored in the instruction array, we can still view the entire execution of simPL program as an interpreter. The interpreter uses compilation, which is an internal detail of its implementation. According to this view, the corresponding T-diagrams are as 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 8.11 *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
```