CS4215—Programming Language Implementation

Martin Henz

Monday 10th February, 2020

Chapter 7

Virtual Machines

7.1 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 Source 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 JavaScript. We used functions, recursion etc. Such an approach is questionable; we explained the high-level language Source by using either a complex mathematical construction or another high-level programming language, JavaScript. Worse: in our denotational semantics, we use conditionals in JavaScript in order to define conditionals in Source, recursion in JavaScript in order to define recursion in Source etc. How are we going to explain JavaScript? 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 Source 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 Source to a machine language. We will formally specify a machine for executing machine language code, and describe its implementation in JavaScript.

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

- SourceA is a calculator language similar to Source0 (Sections 7.2 through 7.7);
- SourceB adds division (Section 7.8);
- SourceC adds conditionals (Section 7.9);
- SourceD adds function definition and application (Sections 7.10 and 7.11); (Section **??**).

Section 7.12 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 JavaScript. Section 7.11 employs the language Java in order to illustrate the dynamic allocation of data structures.

Finally, Section 7.13 describes the overall process of executing Source programs using a virtual machine in terms of T-diagrams.

7.2 The Language SourceA

The language SourceA 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 Source 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 *Source virtual machine code*. The language containing all Source virtual machine code programs is called *SVML*.

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

7.3 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.

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

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 domain of numbers, and b stands for elements of the ring of booleans, respectively. The letters `LDCN` in the machine instruction `LDCN` n stand for "LoaD Constant Number". 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 SourceA.

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

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

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

Example 7.1 *The instruction sequence*

$$[\textit{LDCN}\ 1, \textit{LDCN}\ 2, \textit{PLUS}, \textit{DONE}]$$

represents a valid SVMLa program.

In our JavaScript implementation, we represent instruction sequences as arrays of numbers, where each instruction has a unique number, its "op-code". Some instructions such as `LDCN` have an argument, in which case we place the argument immediately after the op-code. For example the instruction sequence above might be represented by the array

```
[ 17, 1, 17, 2, 18, 19 ]
```

if the op-codes for `LDCN`, `PLUS` and `DONE` are 17, 18 and 19, respectively.

7.4 Compiling SourceA to SVMLa

The translation from SourceA to SVMLa is accomplished by a function

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

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.

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

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

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

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

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

Example 7.2 *Using the usual derivation trees, we can show*
`(1 + 2) * 3` $\twoheadrightarrow$ [*LDCN 1, LDCN 2, PLUS, LDCN 3, TIMES, DONE*], *and*
`1 + (2 * 3)` $\twoheadrightarrow$ [*LDCN 1, LDCN 2, LDCN 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 Source translates a given Source expression—as usual represented by its syntax tree—to an `INSTRUCTION` array.

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

7.5 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 [`LDCN 1`, `LDCN 2`, `PLUS`, `LDCN 3`, `TIMES`, `DONE`], then $s(pc) =$ `PLUS`.

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

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

These load instructions simply push their value on the operand stack. The remaining rules implement the instructions corresponding to SourceA'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 Source, 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 7.3 *The following sequence of states represents the execution of the SVML program* [`LDCN` 10, `LDCN` 20, `PLUS`, `LDCN` 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*.*

7.6 Correctness of the SourceA Implementation

Before we can prove the correctness of our SourceA implementation, we show a result that is more general (but easier to prove). In the following propositions and theorems, we consider well-typed expressions in SourceA.

Proposition 7.1 *Let E be a well-typed expression in SourceA. There exists a value v, and an SVML instruction sequence s such that $E \rightarrowtail v$, $E \hookrightarrow s$, and for any instruction sequence s' and any operand stack os, we have $(os, 0) \rightrightarrows^*_{s.s'} (v.os, |s|)$.*

Proof: (Sketch) We prove the proposition by rule induction on the rules defining SourceA. □

The following theorem states the correctness of the virtual machine implementation of SourceA.

Theorem 7.1 *For every well-typed expression $E \in$ **SourceA**, if $E \rightarrowtail v$, $E \twoheadrightarrow s$, then $R(M_s) = v$.*

Proof: When $E \twoheadrightarrow s$, then there is a sequence t with $s = t.\texttt{DONE}$ and $E \hookrightarrow t$. The theorem follows directly form the previous proposition by setting $s' = \texttt{DONE}$ and $os = \langle\rangle$. □

Corollary 7.1 *For a well-typed expression $E \in$ **SourceA**, the corresponding machine M_s, where $E \twoheadrightarrow s$, will never get stuck.*

We define the runtime of a given SVML program as the number of transitions needed to reach an end configuration. The length of a given sequence of instructions s is denoted by $|s|$.

Lemma 7.1 *The runtime of an SVML program s obtained by compiling a well-typed SourceA program is $|s| - 1$.*

Proof: At each step, the program counter is increased by 1. According to the previous corollary, the `DONE` instruction will be reached. Since $s(|s|-1) = \texttt{DONE}$, the runtime is $|s| - 1$. □

7.7 Implementing a Virtual Machine for SourceA in JavaScript

The following JavaScript 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 JavaScript variables `pc` and `os` to which the interpreter loop has access.

```
// PC is program counter: index of the next instruction
let PC = 0;
// OS is array representing the operand stack of the machine
let OS = [];
// operand stack is initially empty, so we start out with -1
let TOP = -1;
// temporary value, used by the machine instructions
let TMP = 0;

// PUSH and POP are convenient subroutines that operate on
// the operand stack OS, its OS_TOP address and OS_TMP.
// PUSH expects its argument in OS_TMP.
function PUSH() {
    TOP = TOP + 1;
    OS[TOP] = TMP;
}
// POP puts the top-most value into OS_TMP.
function POP() {
    TMP = OS[TOP];
    TOP = TOP - 1;
}

// some registers for intermediate results
let A = 0;
let B = 0;
let C = 0;

const M = [];

M[LDCN] = () =>    { TMP = P[PC + 1];
                     PUSH();
                     PC = PC + 2;
                   };

M[PLUS] = () =>    { POP();
                     A = TMP;
                     POP();
                     TMP = TMP + A;
                     PUSH();
                     PC = PC + 1;
                   };

...

// register that says if machine is running
let RUNNING = true;
```

```
M[DONE] = () =>    { RUNNING = false;
                   };

while (RUNNING) {
    M[P[PC]]();
}
display(OS[TOP]);
```

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

7.8 A Virtual Machine for SourceB

The language SourceB 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 JavaScript implementation, we can just break the loop when encountering the divisor 0. The method `run` will then return the `Error` value.

```
...
let RUNNING = true;
const NORMAL = 0;
const ERROR = 1;
let STATE = NORMAL;

M[DONE] = () =>    { RUNNING = false;
                   };

M[DIV] = () =>     { POP();
                     A = TMP;
                     POP();
                     TMP = TMP / A;
                     PUSH();
                     PC = PC + 1;
                     A = A === 0;
                     if (A) { STATE = ERROR; } else {}
                     if (A) { RUNNING = false; } else {}
                   };
```

```
function run() {
    while (RUNNING) {
        M[P[PC]]();
    }
    if (STATE === ERROR) {
        error(OS[TOP], "execution aborted:");
    } else {
        display(OS[TOP], "result:");
    }
}
```

7.9 A Virtual Machine for SourceC

The language SourceC adds conditionals to SourceB. 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.

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2 \qquad E_3 \hookrightarrow s_3}{E_1 \texttt{ ? } E_2 \texttt{ : } E_3 \hookrightarrow s_1.\texttt{JOFR}\ |s_2| + 2.s_2.\texttt{GOTOR}\ |s_3| + 1.s_3}$$

Example 7.4 *The translation function translates the SourceC expression*

```
2 * (true || false ? 1 + 2 : 2 + 3)
```

to the following instruction sequence.

> [*LDCN* 2, *LDCB true*, *LDCB false*, *OR*, *JOFR* 5, *LDCN* 1, *LDCN* 2, *PLUS*, *GOTOR* 4, *LDCN* 2, *LDCN* 3, *PLUS*, *TIMES*, *DONE*]

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

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

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

Example 7.5 *The following state sequence represents the execution of the program in the previous example.*

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

The compiler in our JavaScript 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 7.6 *Using these new jump instructions, the expression in Example 7.4 is compiled to the following instruction sequence, where the code addresses are indicated for clarity.*

[*LDCN* 2	*0*
LDCB true	*1*
LDCB false	*2*
OR	*3*
JOF 9	*4*
LDCN 1	*5*
LDCN 2	*6*
PLUS	*7*
GOTO 12	*8*
LDCN 2	*9*
LDCN 3	*10*
PLUS	*11*
TIMES	*12*
DONE]	*13*

The new instructions are defined by the following rules.

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

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

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

```
M[JOF] = () =>     { POP();
                     if (!TMP) { PC = P[PC + 1]; } else {}
                     if (TMP) { PC = PC + 2; } else {}
                   };

M[GOTO] = () =>    { PC = P[PC + 1];
                   };
```

7.10 A Virtual Machine for SourceD

The language SourceD adds names, constant declarations, function definition and application to SourceC. These constructs are by far the most challenging aspects of Source 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.

Compilation of Names Similar to the approach of the previous chapter, we implement names by environments. To this aim, we add a register e to the machine state. Register e represents the environment with respect to which the names are executed. Environments map names to denotable values. Thus an environment e, in which `x` refers to the number 1 can be accessed by applying e to `x`, $e(\texttt{x}) = 1$.

Occurrences of names in SourceD are translated to instructions `LDS` x (LoaD Symbolic).

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

Execution of Names The execution of name occurrences pushes the value to which the name 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.

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

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.

$$\frac{E \hookrightarrow s}{(x_1, \ldots, x_n) \texttt{ => \{ } S \texttt{ \}} \hookrightarrow \texttt{LDFS } x_1 \cdots x_n.\texttt{GOTOR } |s| + 2.s.\texttt{RTN}}$$

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 implement 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.

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

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.

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

Returning from a function 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.

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

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:

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

7.11 Implementing a Virtual Machine for SourceD

Similar to the previous section, we are going to look at the implementation aspects of the virtual machine for SourceD step by step. The implementation involves the dynamic creation of data structures for closures, operand stacks, environments and runtime stack frames. To clarify these data structures, we shall use the object-oriented features of the language Java. However note that in a realistic implementation of a virtual machine, all these data structures are residing on a heap which is under the full control of the virtual machine.

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 =         LD;
      INDEX = i;
   }
}
```

In a realistic implementation of a virtual machine, these `LD` instructions and the instructions `LDF` and `CALL` in the next paragraphs are embedded in the virtual machine code array, as shown in the previous sections.

Compilation of function definition To avoid the `GOTOR` instruction after `LDFS` (see page 15), 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 =         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 =         CALL;
     NUMBEROFARGUMENTS = noa;
  }
}
```

Example 7.7 *To illustrate the compilation, let us consider the following SourceD expression.*

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

This expression gets translated to the instruction sequence

[*LDF* 4	*0*
LDCN 2	*1*
CALL 1	*2*
DONE	*3*
LD 0	*4*
LDCN 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.

These environment data structures reside on the machine's heap, and can be thought of as instances of the following class.

```
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 JavaScript 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         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, called `Closure`. These closure data structures reside on the machine's heap, and can be thought of as instances of the following 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         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. These stack frame data structures reside on the machine's heap, and can be thought of as instances of the following class.

```
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 JavaScript 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          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          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`.

7.12 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 7.8 *Consider the following implementation of the factorial function.*

```
function factorial(n) {
    function facloop(n, acc) {
        return n === 1 ? acc
            ? facloop(n - 1, acc * n);
    }
    return facloop(n, 1);
}
factorial(4);
```

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

[*LD* 1	*4*
LDCN 1	*5*
EQUAL	*6*
JOF 10	*7*
LD 2	*8*
RTN	*9*
LD 0	*10*
LD 1	*11*
LDCN 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.

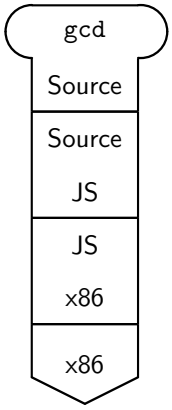
The implementation of `TAILCALL` is simpler and much more efficient than the implementation of `CALL` (compare with page 20).

```
case         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;
                       }
```

7.13 Compilation and Execution

In our virtual machine based implementation of Source, 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 Source 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, assuming that we interpret Source in JavaScript (JS).



Instead of directly executing the instructions, we can instead store the instruction array in a file. This amounts to a Source compiler, which translates Source 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 JavaScript, it is running on top of the JavaScript engine, as depicted in the following T-diagrams.