CS4215—Programming Language Implementation

Martin Henz and Chin Wei Ngan

Sunday 8th January, 2017

Chapter 3

An Overture: The Language ePL

In this chapter, we are looking at a very simple "programming" language, ePL (**e**xpression **P**rogramming **L**anguage), which allows for calculating the results of arithmetic and boolean expressions. A typical "program" in ePL is the expression

```
10 * 2 > 21
```

The user expects that the programming system evaluates such expressions, and returns the result. Thus the execution of an ePL program results in a value, such as the boolean value `false` for the ePL program above. This chapter explores several ways in which such an execution can be precisely described (semantics), as well as ways to implement this execution in a programming system.

The language ePL is very simple; it does not allow for defining functions, data structures, loops etc. It is not possible to program algorithms that go beyond following the rules of expression evaluation. The language is not Turing-complete, which means it is not powerful enough to program arbitrary computational tasks. However, it gives us the opportunity to introduce dynamic and static semantics in Sections 3.2 and 3.3, denotational semantics in Section 3.4 and a semantics based on a virtual machine in Section 3.5, along with language implementations based on these different kinds of semantics.

3.1 The Syntax of ePL

The set of ePL programs is the least set that satisfies the following rules, where n ranges over the set of integers, p_1 ranges over the set of unary primitive operations $P_1 = \{\texttt{\char`\\}\}$, and p_2 ranges over the set of binary primitive operations $P_2 = \{\texttt{|},\texttt{\&},\texttt{+},\ \texttt{-},\texttt{*},\texttt{/},\ \texttt{=},\texttt{>},\texttt{<}\}$.

$$\frac{}{n} \qquad \frac{}{\texttt{true}} \qquad \frac{}{\texttt{false}} \qquad \frac{E}{p_1[E]} \qquad \frac{E_1 \quad E_2}{p_2[E_1, E_2]}$$

We shall give the following meaning to the primitive operators:

- binary boolean operators: `|` (boolean disjunction), `&` (boolean conjunction),
- unary boolean operator: `\` (boolean negation),
- unary arithmetic operator: `~` (integer negation),
- binary arithmetic operators: `+`, `-`, `*`, `/`,
- integer comparison operators: `=` (equality), `<` (less-than), `>` (greater-than).

Syntactic Conventions

We would like to use the following syntactic conventions:

- We can use parentheses in order to group expressions together.
- We use the usual infix and prefix notation for operators. The binary operators are left-associative and the usual precedence rules apply such that `1 + 2 * 3 > 10 - 4` stands for `>[+[1,*[2,3]],-[10,4]]` in prefix form. Also, unary operators have highest precedence such that `~ 1 + 2` stands for `+[[~1],2]` instead of `~[+[1,2]]` in prefix form.

Examples

Example 3.1 (Constants) *The following expression is an ePL program.*

```
42
```

Such programs are constants. Their meaning is the constant taken as arithmetic or boolean values. Other examples for constant expressions are `~333`, and `true`. Note that `~333` is an integer constant expression captured by the first rule above.

Example 3.2 (Arithmetic Expressions) *The following expression is an ePL program.*

```
~15 * 7 + 2
```

The "usual precedence rules" for the binary operators `*` and `+` prescribes that the multiplication is carried out before the addition. In this chapter, we are not concerned with the syntactic appearance of expressions; we assume that the precedence rules are always followed, and treat the expression above as `+[*[~15,7],2]`.

Example 3.3 (Boolean Expressions) *The following expression is an ePL program.*

```
\ false & true | false
```

Here, the conjunction operator `&` has a higher precedence than the disjunction operator `|`, which means that the conjunction is carried out before the disjunction. The equivalent expression in prefix form is `|[&[\false,true],false]`.

Example 3.4 (Mixed Expressions) *The following expression is an ePL program.*

```
17 < 20 - 4 & 10 = 4 + 11
```

Note that we allow the comparison operators `<`, `>` and `=` only between integer values, not between boolean values. Section 3.3 will formalize this restriction.

3.2 Dynamic Semantics of ePL

In order to define how programs are executed, we first use an approach that mimicks a human approach, taken by say a primary school student. The idea is to look for "things to do" and transform a given expression step-by-step until there is "nothing to do" any longer.

More formally, we define a relation that tells us how to carry out evaluation steps to execute ePL programs. This relation, we call *one-step evaluation.* This relation will then serve as the building block for an evaluation relation that defines the evaluation of programs.

3.2.1 Values

The goal of evaluating an expression is to reach a *value*, an expression that cannot be further evaluated. In ePL, a value is either an integer constant, or a boolean constant. In the following rules defining the contraction relation $>_{\mathrm{ePL}}$ for ePL, we denote values by v. That means rules in which v appears are restricted to values in the place of v.

3.2.2 Contraction

Before we get to one-step evaluation, we introduce an auxilary relation called *contraction*, denoted by $>_{\mathrm{ePL}}$, that directly captures the application of primitive operators to values.

For each primitive operation p and each set of values v_1, v_2 (we only have unary and binary operations) such that the result of applying p to v_1 and v_2 is a value v, we define a corresponding contraction rule.

$$\frac{}{p_1[v_1] >_{\mathrm{ePL}} v}\,[\mathrm{OpVals}_1] \qquad \frac{}{p_2[v_1, v_2] >_{\mathrm{ePL}} v}\,[\mathrm{OpVals}_2]$$

Note that this infinite set of rules defines the meaning of the primitive operators. For example, one instance of the second rule is:

$$\frac{}{+[1,1] \ >_{\mathrm{ePL}} 2}$$

3.2.3 One-Step Evaluation

Having defined contraction, we can now define one-step evaluation $\mapsto_{\mathrm{ePL}}$ inductively by the following rules.

$$\frac{E >_{\mathrm{ePL}} E'}{E \mapsto_{\mathrm{ePL}} E'}\text{[Contraction]}$$

The evaluation of sub-expressions of primitive operations is defined as follows.

$$\frac{E \mapsto_{\mathrm{ePL}} E'}{p_1[E] \mapsto_{\mathrm{ePL}} p_1[E']}\text{[OpArg}_1\text{]} \qquad \frac{E_1 \mapsto_{\mathrm{ePL}} E_1'}{p_2[E_1, E_2] \mapsto_{\mathrm{ePL}} p_2[E_1', E_2]}\text{[OpArg}_2\text{]}$$

$$\frac{E_2 \mapsto_{\mathrm{ePL}} E_2'}{p_2[E_1, E_2] \mapsto_{\mathrm{ePL}} p_2[E_1, E_2']}\text{[OpArg}_3\text{]}$$

Note that one-step evaluation does not prescribe the order in which a given ePL expression is evaluated. Both of the following statements hold:

`3 * 2 + 4 * 5` $\mapsto_{\mathrm{ePL}}$ `3 * 2 + 20`
`3 * 2 + 4 * 5` $\mapsto_{\mathrm{ePL}}$ `6 + 4 * 5`

This corresponds to our intuition about evaluation of expressions; we can work on any sub-expression, as long as the sub-expression is evaluated correctly. Note here that this approach makes the implicit assumption that the result of evaluation is independent of the evaluation order. We will study the issue of evaluation order in the next chapter in more detail.

3.2.4 Evaluation

The evaluation relation $\mapsto_{\mathrm{ePL}}{}^*$ is defined as the reflexive transitive closure of one-step evaluation $\mapsto_{\mathrm{ePL}}$, defined as follows:

$$\frac{E_1 \mapsto_{\text{ePL}} E_2}{E_1 \mapsto_{\text{ePL}}{}^* E_2}[\mapsto_{\text{ePL}}{}^*_B] \qquad \frac{}{E \mapsto_{\text{ePL}}{}^* E}[\mapsto_{\text{ePL}}{}^*_R]$$

$$\frac{E_1 \mapsto_{\text{ePL}}{}^* E_2 \quad E_2 \mapsto_{\text{ePL}}{}^* E_3}{E_1 \mapsto_{\text{ePL}}{}^* E_3}[\mapsto_{\text{ePL}}{}^*_T]$$

Using $\mapsto_{\text{ePL}}{}^*$, we can keep evaluating expressions until a value is reached. At that point, there is "nothing left to do", and that value is the result of the computation. An expression for which we can carry out one step according to $\mapsto_{\text{ePL}}$ is called *reducible*, whereas an expression with nothing left to do is called *irreducible*.

Evaluation defines how programs get executed. Such formal definitions of program execution are called a *dynamic semantics*. An important question to ask about a dynamic semantics based on evaluation is whether it is deterministic, i.e. whether for every expression E, there is at most one value v such that $E \mapsto_{\text{ePL}}{}^* v$. We will take a closer look at this question in the next chapter, when we study the dynamic semantics of simPL and its implementation.

3.2.5 Implementation in Java

First, we need to settle how to represent ePL programs in Java. Fortunately, we have defined the language ePL inductively, and therefore can use the technique presented in Chapter 2. The interface for ePL expressions looks like this.

```
public interface Expression {}
```

The three base cases in our inductive definition on page 4 are handled by the following two classes that implement Expression.

```
public class BoolConstant implements Expression {
    public String value;
    public BoolConstant(String v) {
        value = v;
    }
}
public class IntConstant implements Expression {
    public String value;
    public IntConstant(String v) {
        value = v;
    }
}
```

Finally, the rules

$$\frac{E}{p_1[E]} \qquad\qquad \frac{E_1 \quad E_2}{p_2[E_1, E_2]}$$

are represented by classes for unary and binary primitive operators.

```
public class UnaryPrimitiveApplication implements Expression {
    public String operator;
    public Expression argument;
    public UnaryPrimitiveApplication(String op, Expression a) {
      operator = op;
      argument = a;
   }
}
public class BinaryPrimitiveApplication implements Expression {
   public String operator;
   public Expression argument1,argument2;
   public BinaryPrimitiveApplication(String op, Expression a1,
                                     Expression a2) {
      operator = op;
      argument1 = a1;
      argument2 = a2;
   }
```

The dynamic semantics of ePL can be implemented by a function that checks if an expression is reducible. If so, it carries out one step, and checks again. if not, it returns the expression as result.

```
class Evaluator {
    static public Expression evaluate(Expression exp)  {
        return reducible(exp) ?
            evaluate(oneStep(exp))
            : exp;
    }
}
```

The implementation of one-step evaluation is left as an exercise.

3.2.6 Implementation in OCaml

It is very natural to capture ePL programs in OCaml. As language ePL is defined inductively, we can use a recursive data type to represent it, as illustrated below:

```
type op_id = string

type ePL_expr =
  | BoolConst of bool
  | IntConst of int
| UnaryPrimApp of op_id * ePL_expr
| BinaryPrimApp of op_id * ePL_expr * ePL_expr
```

The two base cases are captured by data constructors BoolConst and IntConst. Finally, the rules of

$$\frac{E}{p_1[E]} \qquad\qquad \frac{E_1 \quad E_2}{p_2[E_1, E_2]}$$

are represented by recursive data constructors UnaryPrimApp and BinaryPrimApp, respectively. The first constructor captures an operator identifier and an argument, while the second constructor captures an operator identifier and two arguments.

The dynamic semantics of ePL can be implemented by a function that checks if an expression is reducible. If so, it carries out one step, and checks again. if not, it returns the expression as result.

```
let reducible (e:ePL_expr) : bool =
  match e with
    | BoolConst _ | IntConst _ -> false
    | UnaryPrimApp _ | BinaryPrimApp _ -> true

let rec evaluate (e:ePL_expr): ePL_expr =
  if (reducible e) then evaluate (oneStep e)
  else e
```

The implementation of one-step evaluation is left as an exercise.

3.3 Static Semantics for ePL

Programming languages such as ML and Java (with certain restrictions) are said to be "safe" (or "type safe", or "strongly typed"). This means that certain kinds of errors cannot happen during execution. For example, it will never happen that an integer is mistaken for a boolean value at runtime. To this aim, the user (or the compiler) annotates the program with types. The compiler checks that all type annotations are correct. If things are set up in the right way, programs that are accepted by the compiler will never lead to type-related runtime errors.

The language ePL does not have type declarations for expressions, which means that we need to compute the types that correspond to each sub-expression. Section 3.3.1 defines the notion of well-typed programs by introducing a type system for ePL. This type system will allow us to compute a type for every sub-expression, if such a type exists. Section 3.3.2 then asks what properties we expect from well-typed programs. A precise answer to this question will be given in Chapter 6.

3.3.1 A Type System for ePL

Not all expressions in ePL make sense. For example,

```
true + 1
```

does not make sense, because `true` is a boolean expression, whereas the operator `+` to which `true` is passed as first argument, expects integers as arguments. We say that the expression is *ill-typed*, because a typing condition is not met. Expressions that meet all typing conditions are called *well-typed*.

Expressions in ePL either represent integer or boolean values. Thus, the set of well-typed expressions is defined by the binary *typing relation*

$$\text{“ : ” : ePL} \rightarrow \{\texttt{int}, \texttt{bool}\}$$

We use infix notation for “:”, writing $E : t$, which is read as “the expression E has type t.

Example 3.5 *We will define the typing relation “:” such that the following expressions hold:*

- `1+2` : *int*
- `false & true` : *bool*
- `10 < 17-8` : *bool*

but:

- `true + 1` : t
  does not hold, because in the expression, integer addition is applied to a boolean value.
- `3 + 1 * 5` : `bool`
  does not hold, because the expression has type `int`*, whereas* `bool` *is given after the* : *symbol.*

We define the typing relation inductively as follows.

Constants get their obvious type:

$$\frac{}{n : \texttt{int}}\,[\textbf{Num}] \qquad \frac{}{\texttt{true} : \texttt{bool}}\,[\textbf{True}]$$

$$\frac{}{\texttt{false} : \texttt{bool}}\,[\textbf{False}]$$

For the unary primitive operations `\` and `~` we have the following rules:

$$\frac{E : \texttt{bool}}{\backslash[E] : \texttt{bool}}\,[\textbf{Prim}_1] \qquad \frac{E : \texttt{int}}{\sim [E] : \texttt{int}}\,[\textbf{Prim}_2]$$

For each binary primitive operation p_2, we have a rule of the following form.

$$\frac{E_1 : t_1 \qquad E_2 : t_2}{p[E_1, E_2] : t}\,[\textbf{Prim}_2]$$

where the types t_1, t_2, t are given by the following table.

p	t_1	t_2	t
`+`	`int`	`int`	`int`
`-`	`int`	`int`	`int`
`*`	`int`	`int`	`int`
`/`	`int`	`int`	`int`
`&`	`bool`	`bool`	`bool`
`|`	`bool`	`bool`	`bool`
`=`	`int`	`int`	`bool`
`<`	`int`	`int`	`bool`
`>`	`int`	`int`	`bool`

This completes the definition of the typing relation. Now we can define what it means for an expression to be well-typed.

Definition 3.1 *An expression E is well-typed, if there is a type t such that $E : t$.*

Example 3.6 *The following proof shows that the typing relation holds for the expression* `2 * 3 > 7`: `bool`.

$$\frac{\dfrac{\dfrac{}{\texttt{2} : \texttt{int}}\,[\textbf{Num}] \qquad \dfrac{}{\texttt{3} : \texttt{int}}\,[\textbf{Num}]}{\texttt{2*3} : \texttt{int}}\,[\textbf{Prim}] \qquad \dfrac{}{\texttt{7} : \texttt{int}}\,[\textbf{Num}]}{\texttt{2*3>7} : \texttt{bool}}\,[\textbf{Prim}]$$

Definition 3.2 *The set of well-typed ePL expression is called* well-typed ePL*, denoted using bold font:* ***ePL****.*

In the rest of this chapter, we will only deal with well-typed ePL programs.

3.3.2 Type Safety

The dynamic semantics for ePL of Section 3.2 defines the evaluation of ePL programs. Section 3.3.1 described a way of classifying ePL programs as "well-typed". Such systems for classifying programs are called *static semantics*. Type safety is a property of a given language with a given static and dynamic semantics. It says that if a program of the languge is well-typed, certain problems are guaranteed not to occur at runtime.

What do we consider as "problems"? One kind of problem is that we would get stuck in the process of evaluation. That is the case when no evaluation rule applies to an expression, but the expression is not a value. We would like to be able to guarantee to make *progress* in evaluation. A second kind of problem is that the type changes as evaluation proceeds. This property is called *preservation*.

In Chapter 6, we shall formalize these concepts, leading to a notion of type-safety for the programming language simPL.

3.4 Denotational Semantics of ePL

The dynamic semantics that we have seen so far relies on the idea of contraction. An expression was evaluated by contracting subexpressions, until no further contraction was possible. This evaluation process constituted the "meaning" of programs. We treated the evaluation of expressions as a mere transformation of expressions to expressions. That means that we never left the syntactic realm. Evaluation of expressions was the game of transforming expressions. A minor nuisance was that we had infinitely many rules for the game. We list some major disadvantages of this approach to defining the semantics of a programming language.

- Primitive operations that are not total functions, such as division, can make the evaluation process get stuck. This means that evaluation fails to find a value. We would like to have a more explicit way of handling such a runtime error.

- Dynamic semantics cannot be extended easily to other language paradigms such as imperative programming. In Chapter 11, we shall define the semantics of a simple imperative language.

- Dynamic semantics here is based on a small-step semantics that operate on one execution at a time. Often, we would like to see the final answer from a given expression or code.

As a result of these difficulties, dynamic semantics are less preferred (by theorists[1]) for describing the meaning of computer programs.

The idea of denotational semantics is to directly assign a mathematical value as a meaning to an expression. Compared to dynamic semantics, there are two main advantages of this approach. Firstly, we can employ known mathematical concepts such as integers, booleans, functions etc to describe the meaning of

[1]A slightly more practical semantics is to use operational semantics that is a semantics revolved around some machine configuration. This more advanced topic is beyond the current scope of our course.

programs. Compare this option with the awkward construction of an infinite number of rules for defining simple arithmetic operators! Secondly, denotational semantics avoids the clumsy construction of evaluation as the transitive closure of one-step evaluation, which forced us to define erroneous programs as programs whose evaluation gets "stuck".

We follow the approach of [**?**], and define a denotational semantics as consisting of three parts:

- A description of the syntax of the language in question,
- a collection of semantic domains with associated algebraic operations and properties, and
- a collection of semantic functions which together describe the meaning of programs.

3.4.1 Decimal Numerals

Before we start with the denotational semantics of the language **ePL**, we shall concentrate on a small aspect of ePL, namely decimal numerals representing non-negative integers in ePL programs. The language **N** of decimal numerals contains non-empty strings of decimal digits. We can describe the language using the following twenty rules:

$$\frac{}{\texttt{0}} \quad \cdots \quad \frac{}{\texttt{9}} \quad \frac{n}{n\texttt{0}} \quad \cdots \quad \frac{n}{n\texttt{9}}$$

For example, the sequence of digits `12` and `987654321` are elements of the language **N** of decimal numerals. Such numerals occur in ePL programs such as `12 + 987654321`.

As semantic domain, we choose the integers, denoted by **Int**, taking for granted the ring properties of **Int** with respect to the operations of addition and multiplication.

The semantic function

$$\rightarrowtail_{\mathbf{N}} : \mathbf{N} \rightarrow \mathbf{Int}$$

describes the meaning of decimal numerals as their corresponding integer value. We use the usual notation of rules to describe $\rightarrowtail_{\mathbf{N}}$ as a relation.

$$\frac{}{\texttt{0} \rightarrowtail_{\mathbf{N}} 0}$$

Note that the `0` on the left hand side denotes an element of our language of decimal numerals, whereas the 0 on the right hand side denotes the integer value 0, the neutral element for addition in the ring of integers.

The other nineteen rules for $\rightarrowtail_{\mathbf{N}}$ are:

$$\frac{}{\mathtt{1} \rightarrowtail_{\mathbf{N}} 1} \quad \cdots \quad \frac{}{\mathtt{9} \rightarrowtail_{\mathbf{N}} 9}$$

$$\frac{n \rightarrowtail_{\mathbf{N}} i}{n\mathtt{0} \rightarrowtail_{\mathbf{N}} 10 \cdot i} \quad \frac{n \rightarrowtail_{\mathbf{N}} i}{n\mathtt{1} \rightarrowtail_{\mathbf{N}} 10 \cdot i + 1} \quad \cdots \quad \frac{n \rightarrowtail_{\mathbf{N}} i}{n\mathtt{9} \rightarrowtail_{\mathbf{N}} 10 \cdot i + 9}$$

Again, note the difference between the left and right hand side of $\rightarrowtail_{\mathbf{N}}$. In the last ten rules, the n on the left hand side denote elements of our language of decimal numerals, whereas the i on the right hand side denote integer values. The symbols $+$ and $\cdot$ denote addition and multiplication in the ring of integers.

The left hand sides of $\rightarrowtail_{\mathbf{N}}$ in the bottom of all four rules are mutually distinct. It is therefore easy to see that the relation defined by the rules is indeed a function.

Furthermore, it is not difficult to see that $\rightarrowtail_{\mathbf{N}}$ is a total function, since the rules defining $\rightarrowtail_{\mathbf{N}}$ cover all rules defining $\mathbf{N}$.

To demonstrate the usefulness of denotational semantics for proving properties of languages, let us prove that the "successor" operation on decimal numerals coincides with the successor function on integers.

We define the successor function $'$ on decimal numerals as follows:

$$\frac{}{\mathtt{0}' = \mathtt{1}} \quad \cdots \quad \frac{}{\mathtt{8}' = \mathtt{9}} \quad \frac{}{\mathtt{9}' = \mathtt{10}}$$

$$\frac{}{n\mathtt{0}' = n\mathtt{1}} \quad \cdots \quad \frac{}{n\mathtt{8}' = n\mathtt{9}} \quad \frac{n' = m}{n\mathtt{9}' = m\mathtt{0}}$$

Proposition 3.1 *For all $n \in \boldsymbol{N}$, if $n \rightarrowtail_{\boldsymbol{N}} i$, and $n' \rightarrowtail_{\boldsymbol{N}} j$, then $j = i + 1$.*

Proof: The cases for $\mathtt{0}, \ldots, \mathtt{9}$, and $n\mathtt{0}, \ldots, n\mathtt{8}$ are immediate. We prove by induction on the rules of $\rightarrowtail_{\mathbf{N}}$ that if $n\mathtt{9} \rightarrowtail_{\mathbf{N}} i$, and $n\mathtt{9}' \rightarrowtail_{\mathbf{N}} j$, then $j = i + 1$. For numerals of the form $n\mathtt{9}$, we have $n\mathtt{9}' = n'\mathtt{0}$ according to the definition of $'$. From the definition of $\rightarrowtail_{\mathbf{N}}$, we have $n'\mathtt{0} \rightarrowtail_{\mathbf{N}} 10 \cdot k$, where $n' \rightarrowtail_{\mathbf{N}} k$. From the induction hypothesis, we have: if $n \rightarrowtail_{\mathbf{N}} h$, then $k = h + 1$. Therefore, $10 \cdot k = 10 \cdot (h + 1) = 10 \cdot h + 9 + 1$. From the definition of $\rightarrowtail_{\mathbf{N}}$ and since $n \rightarrowtail_{\mathbf{N}} h$, we have $10 \cdot h + 9 + 1 = i + 1$, and thus $n\mathtt{9}' \rightarrowtail_{\mathbf{N}} j$, where $j = i + 1$. $\square$

3.4.2 Denotational Semantics for ePL without Division

We define the sublanguage **ePL0** of **ePL**, where division is not allowed. Thus, **ePL0** is the well-typed sub-language of ePL0, which is defined by the following rules.

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

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

The following semantic domains are suitable for this language.

Semantic domain	Definition	Explanation
Bool	$\{\mathit{true}, \mathit{false}\}$	ring of booleans
Int	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	ring of integer
EV	$\mathbf{Bool} + \mathbf{Int}$	expressible values

The ring of integers **Int** is already introduced in the previous section. The ring of booleans is the ring formed by the set $\{\mathit{true}, \mathit{false}\}$ with the operators disjunction, denoted by $\vee$, and conjunction, denoted by $\wedge$.

The symbol $+$ that we are using in the last line denotes *disjoint union.* Informally, disjoint union is a kind of union that preserves the origin of the values. That means from an element of $\mathbf{Int} + \mathbf{Bool}$ we can find out whether it came from **Int** or **Bool**, regardless of how integers and booleans are represented, i.e. even if boolean values are represented by integers such as 0 and 1.

Formally, disjoint union can be defined as follows:

$$S_1 + S_2 = \{(1, x_1) \mid x_1 \in S_1\} \cup \{(2, x_2) \mid x_2 \in S_2\}$$

We canonically extend the operations and properties of the component domains **Bool** and **Int** to the set $\mathbf{Int} + \mathbf{Bool}$. We choose the name **EV** (expressible values) for this set to indicate that its elements are the possible results of evaluating **ePL** expressions.

The semantic function

$$\cdot \rightarrowtail \cdot : \mathbf{ePL0} \to \mathbf{EV}$$

defined by the following rules, expresses the meaning of elements of **ePL0**, by defining the value of each element.

$$\frac{}{\texttt{true} \rightarrowtail \mathit{true}} \qquad \frac{}{\texttt{false} \rightarrowtail \mathit{false}} \qquad \frac{n \rightarrowtail_{\mathbf{N}} i}{n \rightarrowtail i}$$

Note that the last rule employs the denotational semantics of decimal numerals described in the previous section.

On the right hand sides of $\rightarrowtail$ in the following rules, we are making use of the operations of addition, subtraction and multiplication in the ring of integers.

$$\frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{+}E_2 \rightarrowtail v_1 + v_2} \qquad \frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{-}E_2 \rightarrowtail v_1 - v_2}$$

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

The following three rules make use of disjunction, conjunction and negation in the ring of booleans.

$$\frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{\&}E_2 \rightarrowtail v_1 \wedge v_2} \qquad \frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\,\texttt{|}\,E_2 \rightarrowtail v_1 \vee v_2} \qquad \frac{E \rightarrowtail v}{\texttt{\textbackslash}E \rightarrowtail \neg v}$$

The operation $\equiv$ in the following rule reifies the identity on integers to a boolean value. For example, $1 \equiv 2 = \mathit{false}$ and $3 \equiv 3 = \mathit{true}$.

$$\frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{=}E_2 \rightarrowtail v_1 \equiv v_2}$$

The operations $>$ and $<$ in the final two rules reflect the less-than and greater-than operations using the usual total ordering on integers.

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

Example 3.7 `1 + 2 > 3` $\rightarrowtail$ *false holds because* `1 + 2` $\rightarrowtail 3$ *and* $3 > 3$ *is false.*

3.4.3 Denotational Semantics for ePL

The language **ePL** adds division to **ePL0**.

$$\frac{E_1 \quad E_2}{E_1\texttt{/}E_2}$$

The difficulty lies in the fact that division on integers is a partial function, not being defined for 0 as second argument. In this section, we are more ambitious than in the previous one, and want to give meaning to programs, even if division by 0 occurs. For this purpose, we extend the definitions of semantic domains and functions as follows.

Semantic domain	Definition	Explanation
Bool	$\{\mathit{true}, \mathit{false}\}$	ring of booleans
Int	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	ring of integers
EV	**Bool** + **Int** + $\{\perp\}$	expressible values
DV	**Bool** + **Int**	denotable values

Note that we add the symbol $\perp$ to the set of expressible values. The meaning of expressions that execute a division by 0 will be $\perp$. The semantic function $\cdot \Vdash \cdot \rightarrowtail \cdot$ is modified to take the occurrence of the error value $\perp$ into account.

$$\frac{}{\texttt{true} \rightarrowtail \mathit{true}} \qquad \frac{}{\texttt{false} \rightarrowtail \mathit{false}} \qquad \frac{n \rightarrowtail_{\mathbf{N}} i}{n \rightarrowtail i}$$

Instead of having one single rule for each primitive operator, we now have three rules for each of the binary operators `+`, `-`, and `*`. The two additional rules in each case express that the meaning of an expression is $\perp$ if the meaning of one of the component expressions is $\perp$.

$$\frac{E_1 \rightarrowtail \perp}{E_1\texttt{+}E_2 \rightarrowtail \perp} \qquad \frac{E_2 \rightarrowtail \perp}{E_1\texttt{+}E_2 \rightarrowtail \perp} \qquad \frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{+}E_2 \rightarrowtail v_1 + v_2} \text{ if } v_1, v_2 \neq \perp$$

$$\frac{E_1 \rightarrowtail \perp}{E_1\texttt{-}E_2 \rightarrowtail \perp} \qquad \frac{E_2 \rightarrowtail \perp}{E_1\texttt{-}E_2 \rightarrowtail \perp}$$

$$\frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{-}E_2 \rightarrowtail v_1 - v_2} \text{ if } v_1, v_2 \neq \perp$$

$$\frac{E_1 \rightarrowtail \perp}{E_1\texttt{*}E_2 \rightarrowtail \perp} \qquad \frac{E_2 \rightarrowtail \perp}{E_1\texttt{*}E_2 \rightarrowtail \perp}$$

$$\frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{*}E_2 \rightarrowtail v_1 \cdot v_2} \text{ if } v_1, v_2 \neq \perp$$

The first three rules for division are similar. In the third rule / stands for integer division (with rounding towards 0).

$$\frac{E_1 \rightarrowtail \perp}{E_1\texttt{/}E_2 \rightarrowtail \perp} \qquad \frac{E_2 \rightarrowtail \perp}{E_1\texttt{/}E_2 \rightarrowtail \perp}$$

$$\frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{/}E_2 \rightarrowtail v_1 / v_2} \text{ if } v_1, v_2 \neq \perp \text{ and } v_2 \neq 0$$

The last rule for division covers the case that the meaning of the second argument of division is 0. Since division by 0 is not defined, the meaning of the entire expression is $\perp$.

$$\frac{E_2 \rightarrowtail 0}{E_1\texttt{/}E_2 \rightarrowtail \perp}$$

Equipped with this scheme of handling the error value, the remaining rules for **ePL** are not surprising.

$$\frac{E_1 \rightarrowtail \perp}{E_1\texttt{\&}E_2 \rightarrowtail \perp} \qquad \frac{E_2 \rightarrowtail \perp}{E_1\texttt{\&}E_2 \rightarrowtail \perp}$$

$$\frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{\&}E_2 \rightarrowtail v_1 \wedge v_2} \text{ if } v_1, v_2 \neq \perp$$

$$\frac{E_1 \rightarrowtail \perp}{E_1\texttt{|}E_2 \rightarrowtail \perp} \qquad \frac{E_2 \rightarrowtail \perp}{E_1\texttt{|}E_2 \rightarrowtail \perp}$$

$$\frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{|}E_2 \rightarrowtail v_1 \vee v_2} \text{ if } v_1, v_2 \neq \perp$$

$$\frac{E \rightarrowtail \perp}{\backslash E \rightarrowtail \perp} \qquad \frac{E \rightarrowtail v}{\backslash E \rightarrowtail \neg v} \text{ if } v \neq \perp$$

$$\frac{E_1 \rightarrowtail \perp}{E_1\texttt{=}E_2 \rightarrowtail \perp} \qquad \frac{E_2 \rightarrowtail \perp}{E_1\texttt{=}E_2 \rightarrowtail \perp}$$

$$\frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{=}E_2 \rightarrowtail v_1 \equiv v_2} \text{ if } v_1, v_2 \neq \perp$$

$$\frac{E_1 \rightarrowtail \perp}{E_1\texttt{>}E_2 \rightarrowtail \perp} \qquad \frac{E_2 \rightarrowtail \perp}{E_1\texttt{>}E_2 \rightarrowtail \perp}$$

$$\frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{>}E_2 \rightarrowtail v_1 > v_2} \text{ if } v_1, v_2 \neq \bot$$

$$\frac{E_1 \rightarrowtail \bot}{E_1\texttt{<}E_2 \rightarrowtail \bot} \qquad \frac{E_2 \rightarrowtail \bot}{E_1\texttt{<}E_2 \rightarrowtail \bot}$$

$$\frac{E_1 \rightarrowtail v_1 \qquad E_2 \rightarrowtail v_2}{E_1\texttt{<}E_2 \rightarrowtail v_1 < v_2} \text{ if } v_1, v_2 \neq \bot$$

Note that by introducing the error value, we achieve that $\rightarrowtail$ is still a total function although its component function / is not.

Example 3.8 `5+(3/0)` $\rightarrowtail \bot$, *since* `3/0` $\rightarrowtail \bot$.

Semantic rules that properly treat error values tend to be complex. (They took us a couple of pages for the simple language ePL.) In the following, we are therefore omitting the treatment of the error value for simplicity.

3.5 A Virtual Machine for ePL

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 **ePL** with respect to error handling was achieved by employing a considerable mathematical machinery, using one-step evaluation and evaluation for *dynamic semantics* and complex semantic domains for *denotational semantics*. In their implementation, we made heavy use of Java/OCaml written in an interpretive fashion. Such an approach is questionable; we explained the language **ePL** by using either a complex mathematical construction or another high-level programming language, Java/OCaml. 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 **ePL** 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 **ePL** to a machine language. We will formally specify a machine for executing machine language code, and describe its implementation in Java/OCaml.

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 **ePL** 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 *ePL virtual machine code.* The language containing all ePL virtual machine code programs is called *eVML.*

Section 3.5.1 describes the virtual machine language eVML. Section 3.5.4 presents the compilation process from **ePL** to eVML, Section 3.5.5 shows how the machine runs, and Section 3.5.6 asks whether we could prove that such an implementation is correct. Section 3.5.8 shows how eVML can be implemented in Java, and Section 3.5.9 describes the overall process of executing **ePL** programs using a virtual machine written in Java in terms of T-diagrams.

3.5.1 The Machine Language eVML

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

eVML 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 eVML 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 **ePL**.

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

$$\frac{s}{\texttt{AND}.s} \qquad \frac{s}{\texttt{OR}.s} \qquad \frac{s}{\texttt{NOT}.s}$$

$$\frac{s}{\texttt{LT}.s} \qquad \frac{s}{\texttt{GT}.s} \qquad \frac{s}{\texttt{EQ}.s}$$

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

Example 3.9 *The instruction sequence*

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

represents a valid eVML program.

3.5.2 Implementing eVML in Java

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 String VALUE;
   public LDCI(String 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 3.10 *Now, we can create the instruction sequence in Example 3.9 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();
```

3.5.3 Implementing eVML in OCaml

For an Java implementation, we can use a simple algebraic data type to to represent the various instructions:

```
type eVML_inst =
  | LDCI of int
  | LDCB of int (* 0 - false; 1 - true *)
  | PLUS | MINUS | TIMES | DIV | AND | NEG
  | NOT | OR | LT | GT | EQ | DONE

type eVML_prog = eVML_inst list
```

Note how we encode boolean values into integer (like the C language). This is to allow us to support our machine with a simple stack of only integers. Each `eVML` instruction is uniquely distinguished by a data constructor. Only two of the instructions, with constructors `LDCI` and `LDCB`, carries an argument that is meant to be loaded onto the top of its stack.

Example 3.11 *Now, we can create the instruction sequence* `iSeq` *and then placed it into an array* `iArr` *(to support fast retrieval) in Example 3.9 as follows:*

```
let iSeq : eVML_prog = [LDCI 1; LDCI 2; PLUS; DONE]

let iArr : eVML_inst array = Array.of_list i_seq
```

3.5.4 Compiling ePL to eVML

The translation from **ePL** to eVML is accomplished by a function

$$\twoheadrightarrow : \mathbf{ePL} \rightarrow \text{eVML}$$

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.

$$\frac{}{n \hookrightarrow \texttt{LDCI}\ n} \qquad \frac{}{\texttt{true} \hookrightarrow \texttt{LDCB true}} \qquad \frac{}{\texttt{false} \hookrightarrow \texttt{LDCB false}}$$

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

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

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1\texttt{|}E_2 \hookrightarrow s_1.s_2.\texttt{OR}} \qquad \frac{E \hookrightarrow s}{\backslash\ E \hookrightarrow s.\texttt{NOT}} \qquad \frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1\texttt{<}E_2 \hookrightarrow s_1.s_2.\texttt{LT}}$$

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1\texttt{>}E_2 \hookrightarrow s_1.s_2.\texttt{GT}} \qquad \frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1\texttt{=}E_2 \hookrightarrow s_1.s_2.\texttt{EQ}}$$

Example 3.12 *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 Reverse Polish Notation, in honor of its inventor, the Polish logician Jan Lukasiewicz, and is designed for efficient execution on automata.

Our compiler for **ePL** translates a given ePL expression—as usual represented by its syntax tree—to an `INSTRUCTION` array. In Java:

```
Expression epl=Parse.fromFileName(eplfile);
INSTRUCTION ia[] = Compile.compile(epl));
```

In OCaml:

```
let epl = Parse.fromFileName(eplfile)
let ia = Array.of_list (compile epl)
```

3.5.5 Executing eVML Code

The machine that we will use to execute eVML 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 eVML, 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 eVML 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 **ePL**'s operators. They pop their arguments from the operand stack, and push the result of the operation back onto the operand stack.

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

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

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.

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

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

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

$$\frac{s(pc) = \texttt{OR}}{(b_2.b_1.os, pc) \rightrightarrows_s (b_1 \vee b_2.os, pc+1)}$$

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

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

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

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

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

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 3.13 *The execution of the eVML program*

$$[\texttt{LDCI}\ 10, \texttt{LDCI}\ 20, \texttt{PLUS}, \texttt{LDCI}\ 6, \texttt{TIMES}, \texttt{DONE}]$$

is represented by the following sequence of states:

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

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

3.5.6 Correctness of the ePL Implementation

Having defined eVML and the compilation from **ePL** to eVML formally, we could ask the question of correctness of the compilation-based implementation of **ePL** as follows.

Let E be a well-typed expression in **ePL**, and v a value, such that $E \rightarrowtail v$. Does an instruction sequence s exist such that $E \twoheadrightarrow s$, and $R(M_s) = v$.

In Chapter 8, we shall answer this question for the virtual-machine based implementation of simPL, a superset of **ePL**.

3.5.7 Implementing a Virtual Machine for ePL 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.

3.5.8 Implementing a Virtual Machine for ePL in OCaml

The following OCaml program shows the structure of our machine. We wrote it in a functional style that uses a stack (of integer values) that is modified by its methods (for efficiency reasons). The machine's execution is carried out by a tail-recursive[2] method `execute` that repeatedly takes its next instruction for execution until `DONE` is encountered. The program counter is captured as a parameter, called `pc`, of the `execute` method. When `DONE` is encountered, the execution stops by returning the value on the top of the stack. Our execution would start with its `pc` parameter set to 0 using `execute 0`.

```
let eVML_mc (instArr:eVML_inst array) =
  let stk = Stack.create () in
  let rec execute pc =
    let c = Array.get instArr pc in
    match c with
      | DONE -> Stack.pop stk
      | _ -> (proc_inst stk c; execute (pc+1))
  in execute 0
```

For clarity, the execution of each instruction is implemented by a separate method `proc_inst`. Each instruction executes through side-effecting operations on its data stack. A fragment of this method is given below. The instructions `LDCI i` and `LDCB i` would push its only argument onto the top of the stack. The instruction `PLUS` would pop its two arguments from the top of the stack, adds them, before pushing its result back onto the stack.

```
let proc_inst stk c =
  match c with
    | LDCI i -> Stack.push i stk
    | LDCB i -> Stack.push i stk
    | PLUS ->
          let a2 = Stack.pop stk in
          let a1 = Stack.pop stk in
          Stack.push (a1+a2) stk
```

As OCaml supports an object-oriented programming style, we could also choose to implement our simple virtual machine using this style. Let us see how this can be done. We first create a class of `eVML` machine which takes a list of instruction `instSeq` as its parameter. Our machine has a mutable

[2]Note that tail-recursive method can be implemented using constant stack size, and would have the same execution performance as imperative while loops.

program counter, called `pc`, a stack, called `stk`, and an array `instArr` to hold the commands. For better modularity, we provide a method `finish` to check if the next instruction is `DONE` and a method `oneStep` to execute the next instruction before incrementing the program counter. The main execution is carried out by method `execute` which will execute each instruction until `DONE` is enountered.

```
class eVML instSeq =
   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 to execute till DONE encountered *)
     (* and return the value on top of its stack *)
     method execute () : int =
       if mc # finish () then Stack.pop stk
       else begin mc # oneStep(); mc # execute () end
   end;;
```

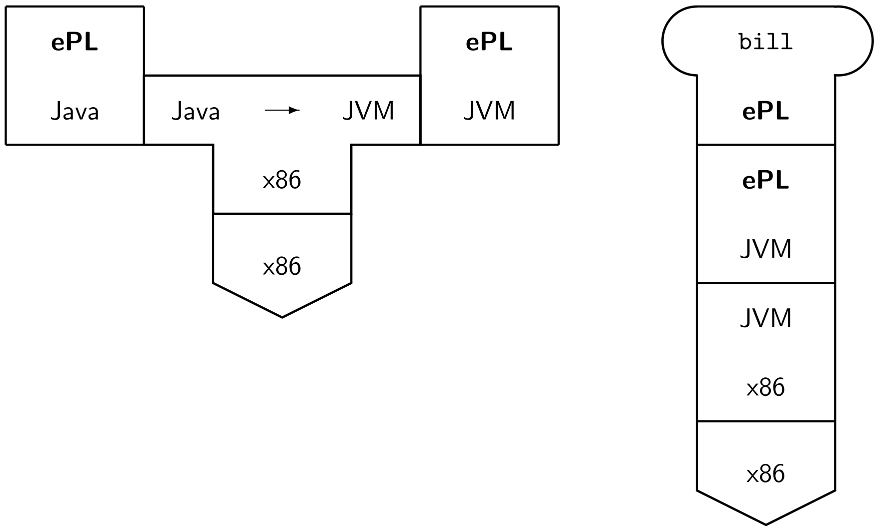
A command to create one eVML machine with a simple sequence of instruction loaded, and then executed, is shown below.

```
let iSeq : eVML_prog = [LDCI 1; LDCI 2; PLUS; DONE]
let oneMC = new eVML iSeq
let ans = oneMC # execute ()
```

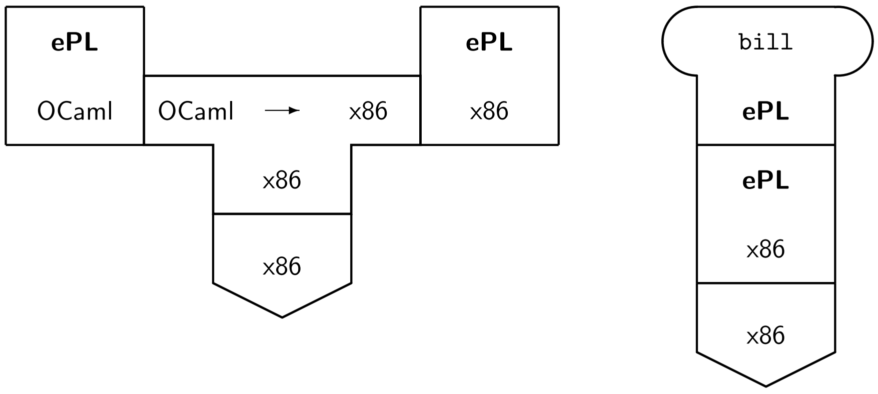
3.5.9 Compilation and Execution

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

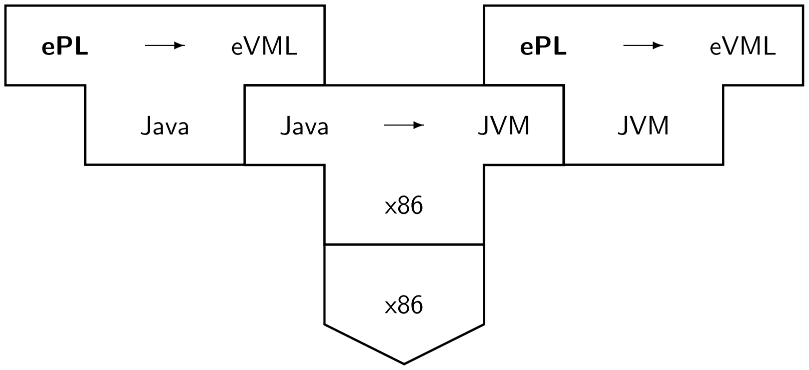
If we choose to directly execute the instructions stored in the instruction array, we can still view the entire execution of **ePL** program as an interpreter. The interpreter uses compilation, which is an internal detail of its implementation. According to this view, the corresponding T-diagrams are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



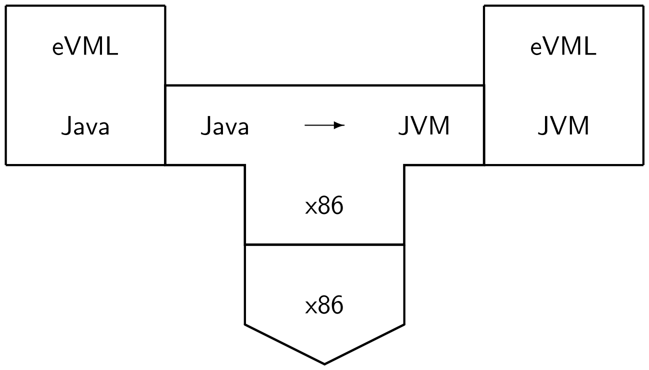
For the OCaml implementation, if we use the native compiler, we would have the following T-diagram set-up for our language processing.



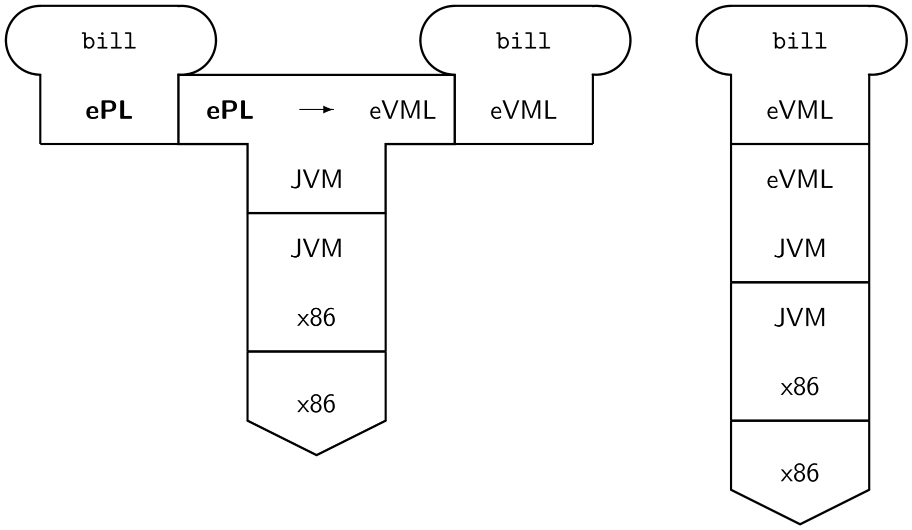
Instead of directly executing the instructions, we can instead store the instruction array in a file (in Java easily done using an `ObjectOutputStream`). This amounts to a **ePL** compiler, which translates **ePL** files to eVML files. For the Java implementation of the compiler is written in Java, we first need to translate it to JVM code, as summarised by the T-diagram below.



The machine loads a given eVML file and executes its eVML code. Thus the machine acts as an emulator for eVML. Since it is also implemented in Java, we translate it to JVM code as follows.



Finally, we are compiling and executing our **ePL** program `bill.epl`, as depicted in the following T-diagrams.



Example 3.14 *Using the compiler* `eplc` *and the emulator* `epl`*, both written in Java, we can execute a given* ***ePL*** *program* `bill.epl` *as follows:*

```
> javac eplc.java
> javac epl.java
> java eplc bill.epl
> ls bill.*
bill.epl bill.evml
> java epl bill
249
```

In the case of the corresponding OCaml implementation, you would be performing the following steps instead.

Example 3.15
```
> ocamlopt eplc.ml -o eplc
> ocamlopt eplm.ml -o eplm
> ./eplc bill.epl
> ls bill.*
bill.epl bill.evml
> ./eplm bill.evml
249
```

The first two commands generate an `eplc` *compiler, and an* `eplm` *virtual machine using the OCaml native compiler. After that, we use the* `eplc` *compiler on our simple* `bill.epl` *program into its corresponding virtual machine code,* `bill.evml`*. This is then executed on our virtual machine using* `eplm`*.*

Exercise 3.1 *As an exercise in T-diagram, draw a diagram corresponding to the steps used to obtain an* `ePL` *compiler and an* `eVML` *virtual machine using the native compiler of OCaml. Illustrate also the T-diagram for compiling and then executing the given* `bill.epl` *program.*