CS4215—Programming Language Implementation

Martin Henz

Thursday 19 January, 2012
Chapter 3

An Overture: The Language ePL

In this chapter, we are looking at a very simple “programming” language, ePL (expression Pogramming Language), which allows for calculating the results of arithmetic and boolean expressions. A typical “program” in ePL is the expression

\[ 10 \times 2 > 21 \]

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 \texttt{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 \( P_1 = \{\backslash\} \), and \( p_2 \) ranges over the set of binary primitive operations \( P_2 = \{1,\&,+, -,\times,\div, =,>,<\} \).
We shall give the following meaning to the primitive operators:

- binary boolean operators: $\|$ (boolean disjunction), $\&$ (boolean conjunction),
- unary boolean operator: \ (boolean negation),
- binary arithmetic operators: $+$, $-$, $\ast$, $/$,
- 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 $1 + 2 \ast 3 > 10 - 4$ stands for $> \ast \lbrack 1, \ast \lbrack 2, 3 \rbrack, - \lbrack 10, 4 \rbrack$.

**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 $\text{true}$. 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.

$-15 \ast 7 + 2$

The “usual precedence rules” for the binary operators $\ast$ 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 $+ \ast \lbrack -15, \ast \lbrack 7, 2 \rbrack$.

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

$\text{false} \& \text{true} \mid \text{false}$
Here, the conjunction operator \& has a higher precedence than the disjunction operator |, which means that the conjunction is carried out before the disjunction.

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 mimics 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 \(\>_{ePL}\) 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 auxiliary relation called contraction, denoted by \(\>_{ePL}\), 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.

\[
\begin{align*}
\text{[OpVals_1]} & \\
\text{\(p_1[v_1] >_{ePL} v\)} & \quad \text{[OpVals_2]} \\
\text{\(p_2[v_1, v_2] >_{ePL} v\)}
\end{align*}
\]
CHAPTER 3. THE LANGUAGE EPL

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

\[ [+1, 1] \xrightarrow{ePL} 2 \]

3.2.3 One-Step Evaluation

Having defined contraction, we can now define one-step evaluation \( \xrightarrow{ePL} \) inductively by the following rules.

\[
E \xrightarrow{ePL} E'
\]

[Contraction]

\( E \xrightarrow{ePL} E' \)

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

\[
p_1[E] \xrightarrow{ePL} p_1[E'] \quad p_2[E_1, E_2] \xrightarrow{ePL} p_2[E'_1, E'_2]
\]

[OpArg1]

[OpArg2]

\[
p_2[E_1, E_2] \xrightarrow{ePL} p_2[E'_1, E'_2]
\]

[OpArg3]

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

\[
3 \times 2 + 4 \times 5 \xrightarrow{ePL} 3 \times 2 + 20
\]

\[
3 \times 2 + 4 \times 5 \xrightarrow{ePL} 6 + 4 \times 5
\]

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 \( \xrightarrow{ePL}^* \) is defined as the reflexive transitive closure of one-step evaluation \( \xrightarrow{ePL} \), defined as follows:
Using $\Rightarrow_{ePL}$, 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 $\Rightarrow_{ePL}$ 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 $E \Rightarrow_{ePL} v$. 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

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

$$
\begin{align*}
E_1 \Rightarrow_{ePL} E_2 & \quad \Rightarrow_{ePL}^* E_2 & [\Rightarrow_{ePL_R}^*] \\
E_1 \Rightarrow_{ePL}^* E_2 & \quad E \Rightarrow_{ePL}^* E & [\Rightarrow_{ePL_R}^*]
\end{align*}
$$

$$
\begin{align*}
E_1 \Rightarrow_{ePL}^* E_2 & \quad E_2 \Rightarrow_{ePL}^* E_3 & [\Rightarrow_{ePL_T}^*] \\
E_1 \Rightarrow_{ePL}^* E_3
\end{align*}
$$
are represented by classes for unary and binary primitive operators.

```java
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.

```java
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.

### 3.3 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,

\[ \text{true + 1} \]

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

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

\[
\text{";" : ePL} \rightarrow \{\text{int, bool}\}
\]

We use infix notation for \text{";"}, writing \(E : t\), which is read as \text{“the expression } E \text{ has type } t\.

**Example 3.5** We will define the typing relation \text{";"} such that the following expressions hold:

- \(1 + 2 : \text{int}\)
- \(\text{false} \& \text{true} : \text{bool}\)
- \(10 < 17 - 8 : \text{bool}\)

but:

- \(\text{true + 1 : } t\)
  does not hold, because in the expression, integer addition is applied to a boolean value.

- \(3 + 1 * 5 : \text{bool}\)
  does not hold, because the expression has type \text{int}, whereas \text{bool} is given after the \text{;} symbol.

We define the typing relation inductively as follows.

Constants get their obvious type:

\[
\begin{align*}
\text{[Num]} & : \text{int} \\
\text{[True]} & : \text{bool}
\end{align*}
\]
CHAPTER 3. THE LANGUAGE EPL

---

false : bool

For the unary primitive operation \ we have the following rule.

\[ E : \text{bool} \]

\[ \text{\textbackslash}[E] : \text{bool} \]

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

\[ E_1 : t_1 \]

\[ E_2 : t_2 \]

\[ p[E_1, E_2] : t \]

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

<table>
<thead>
<tr>
<th>( p )</th>
<th>( t_1 )</th>
<th>( t_2 )</th>
<th>( t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>int</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>-</td>
<td>int</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>*</td>
<td>int</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>/</td>
<td>int</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>&amp;</td>
<td>bool</td>
<td>bool</td>
<td>bool</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bool</td>
<td>bool</td>
</tr>
<tr>
<td>=</td>
<td>int</td>
<td>int</td>
<td>bool</td>
</tr>
<tr>
<td>&lt;</td>
<td>int</td>
<td>int</td>
<td>bool</td>
</tr>
<tr>
<td>&gt;</td>
<td>int</td>
<td>int</td>
<td>bool</td>
</tr>
</tbody>
</table>

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 \( 2 * 3 > 7 : \text{bool} \).

\[ \text{[Num]} \]

\[ 2 : \text{int} \]

\[ \text{[Num]} \]

\[ 3 : \text{int} \]

\[ \text{[Prim]} \]

\[ 2*3 : \text{int} \]

\[ \text{[Num]} \]

\[ 7 : \text{int} \]

\[ \text{[Prim]} \]

\[ 2*3>7 : \text{bool} \]
3.4. DENOTATIONAL SEMANTICS OF EPL

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

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 language 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.

As a result of these difficulties, dynamic semantics are rarely used 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
corcepts such as integers, booleans, functions etc to describe the meaning of
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 [Sch88], 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:

\[
\begin{array}{c}
0  \\
\vdots  \\
9  \\
n  \\
\vdots  \\
n  \\
n  \\
n  \\
\end{array}
\]

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 \( \text{Int} \), taking for
granted the ring properties of \( \text{Int} \) with respect to the operations of addition
and multiplication.

The semantic function

\[
\Rightarrow_N : N \rightarrow \text{Int}
\]
describes the meaning of decimal numerals as their corresponding integer value.
We use the usual notation of rules to describe \( \Rightarrow_N \) as a relation.

\[
\begin{array}{c}
0  \\
\Rightarrow_N  \\
0  \\
\end{array}
\]

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 $\vdash_N$ are:

\[
\begin{align*}
1 & \vdash_N 1 \\
\cdot \cdot \cdot \\
9 & \vdash_N 9 \\
n & \vdash_N i \\
n & \vdash_N i \\
\cdot \cdot \cdot \\
n0 & \vdash_N 10 \cdot i \\
n1 & \vdash_N 10 \cdot i + 1 \\
n9 & \vdash_N 10 \cdot i + 9 \\
n0 & \vdash_N n0' \\
n1 & \vdash_N n1' \\
n8 & \vdash_N n8' \\
n9 & \vdash_N n9'
\end{align*}
\]

Again, note the difference between the left and right hand side of $\vdash_N$. 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 left hand sides of $\vdash_N$ 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 $\vdash_N$ is a total function, since the rules defining $\vdash_N$ cover all rules defining $N$.

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:

\[
\begin{align*}
0' &= 1 \\
\cdot \cdot \cdot \\
9' &= 10 \\
n0' &= n1 \\
n1' &= n1 \\
n8' &= n8 \\
n9' &= m0
\end{align*}
\]

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

**Proof:** The cases for $0, \ldots, 9$, and $n0, \ldots, n8$ are immediate. We prove by induction on the rules of $\vdash_N$ that if $n9 \vdash_N i$, and $n9' \vdash_N j$, then $j = i + 1$.

For numerals of the form $n9$, we have $n9' = n9'0$ according to the definition of $'$.

From the definition of $\vdash_N$, we have $n0' \vdash_N 10 \cdot k$, where $n' \vdash_N k$. From the induction hypothesis, we have: if $n \vdash_N h$, then $k = h + 1$. Therefore, $10 \cdot k = 10 \cdot (h + 1) = 10 \cdot h + 9 + 1$. From the definition of $\vdash_N$ and since $n \vdash_N h$, we have $10 \cdot h + 9 + 1 = i + 1$, and thus $n9' \vdash_N j$, where $j = i + 1$. □

### 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.
CHAPTER 3. THE LANGUAGE EPL

\[
\begin{array}{c|c|c}
  n & \text{true} & \text{false} \\
\end{array}
\]

\[
\begin{array}{l}
  E_1 \quad E_2 \\
  \quad \quad \text{where } p \in \{1, \&\,+, -, \ast, =, >, <\} \\
  p[E_1, E_2] \\
\end{array}
\]

\[
\begin{array}{l}
  E \quad \text{where } p \in \{\\} \\
  p[E] \\
\end{array}
\]

The following semantic domains are suitable for this language.

<table>
<thead>
<tr>
<th>Semantic domain</th>
<th>Definition</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bool</td>
<td>{true, false}</td>
<td>ring of booleans</td>
</tr>
<tr>
<td>Int</td>
<td>{..., -2, -1, 0, 1, 2, ...}</td>
<td>ring of integer</td>
</tr>
<tr>
<td>EV</td>
<td>Bool + Int</td>
<td>expressible values</td>
</tr>
</tbody>
</table>

The ring of integers \textbf{Int} is already introduced in the previous section. The ring of booleans is the ring formed by the set \{true, false\} with the operators disjunction, denoted by \(\lor\), and conjunction, denoted by \(\land\).

The symbol + that we are using in the last line denotes \textit{disjoint union}. Informally, disjoint union is a kind of union that preserves the origin of the values. That means from an element of \textbf{Int} + \textbf{Bool} we can find out whether it came from \textbf{Int} or \textbf{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 \textbf{Bool} and \textbf{Int} to the set \textbf{Int} + \textbf{Bool}. We choose the name \textbf{EV} (expressible values) for this set to indicate that its elements are the possible results of evaluating \textbf{ePL} expressions.

The semantic function

\[
\cdot \mapsto : \textbf{ePL} \to \textbf{EV}
\]

defined by the following rules, expresses the meaning of elements of \textbf{ePL}, by defining the value of each element.

\[
\begin{array}{l}
  n \mapsto \_N \ i \\
  \text{true} \mapsto \text{true} \quad \text{false} \mapsto \text{false} \\
  n \mapsto i \\
\end{array}
\]

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

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

\[
\begin{array}{l}
  E_1 \mapsto v_1 \quad E_2 \mapsto v_2 \\
  E_1 + E_2 \mapsto v_1 + v_2 \\
  E_1 - E_2 \mapsto v_1 - v_2 \\
\end{array}
\]

\[
\begin{array}{l}
  E_1 \mapsto v_1 \quad E_2 \mapsto v_2 \\
  E_1 \ast E_2 \mapsto v_1 \ast v_2 \\
\end{array}
\]
3.4. DENOTATIONAL SEMANTICS OF EPL

\[ E_1 \rightarrow v_1 \quad E_2 \rightarrow v_2 \]

\[ E_1 \cdot E_2 \rightarrow v_1 \cdot v_2 \]

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

\[ E_1 \rightarrow v_1 \quad E_2 \rightarrow v_2 \quad E_1 \rightarrow v_1 \quad E_2 \rightarrow v_2 \quad E \rightarrow v \]

\[ E_1 \& E_2 \rightarrow v_1 \land v_2 \quad E_1 \mid E_2 \rightarrow v_1 \lor v_2 \quad \neg E \rightarrow \neg v \]

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

\[ E_1 \rightarrow v_1 \quad E_2 \rightarrow v_2 \]

\[ E_1 \equiv E_2 \rightarrow v_1 \equiv v_2 \]

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

\[ E_1 \rightarrow v_1 \quad E_2 \rightarrow v_2 \]

\[ E_1 > E_2 \rightarrow v_1 > v_2 \quad E_1 < E_2 \rightarrow v_1 < v_2 \]

Example 3.7 \( 1 + 2 > 3 \rightarrow \text{false} \) holds because \( 1 + 2 \rightarrow 3 \) and \( 3 > 3 \) is false.

3.4.3 Denotational Semantics for ePL

The language ePL adds division to ePL0.

\[ E_1 \quad E_2 \]

\[ E_1 / E_2 \]

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.

<table>
<thead>
<tr>
<th>Semantic domain</th>
<th>Definition</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bool</td>
<td>{true, false}</td>
<td>ring of booleans</td>
</tr>
<tr>
<td>Int</td>
<td>{\ldots, -2, -1, 0, 1, 2, \ldots}</td>
<td>ring of integers</td>
</tr>
<tr>
<td>EV</td>
<td>Bool + Int + {\bot}</td>
<td>expressible values</td>
</tr>
<tr>
<td>DV</td>
<td>Bool + Int</td>
<td>denotable values</td>
</tr>
</tbody>
</table>
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 \( \cdot \models \cdot \) is modified to take the occurrence of the error value \( \perp \) into account.

\[
\begin{align*}
true & \rightarrow true & false & \rightarrow false & n & \rightarrow \mathbb{N} i \\
\end{align*}
\]

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 \( \perp \).

\[
\begin{align*}
E_1 & \rightarrow \perp & E_2 & \rightarrow \perp & E_1 & \rightarrow v_1 & E_2 & \rightarrow v_2 & \text{if } v_1, v_2 \neq \perp \\
E_1 + E_2 & \rightarrow \perp & E_1 + E_2 & \rightarrow v_1 + v_2 \\
E_1 - E_2 & \rightarrow \perp & E_2 & \rightarrow \perp & E_1 - E_2 & \rightarrow v_1 - v_2 & \text{if } v_1, v_2 \neq \perp \\
E_1 * E_2 & \rightarrow \perp & E_1 * E_2 & \rightarrow v_1 * v_2 & \text{if } v_1, v_2 \neq \perp \\
\end{align*}
\]

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

\[
\begin{align*}
E_1 & \rightarrow \perp & E_2 & \rightarrow \perp \\
E_1 / E_2 & \rightarrow \perp & E_1 / E_2 & \rightarrow \perp \\
E_1 & \rightarrow v_1 & E_2 & \rightarrow v_2 & \text{if } v_1, v_2 \neq \perp \text{ and } v_2 \neq 0 \\
E_1 / E_2 & \rightarrow v_1 / v_2 \\
\end{align*}
\]
3.4. **DENOTATIONAL SEMANTICS OF EPL**

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 \( \bot \).

\[
E_2 \rightarrow 0
\]

\[
E_1/E_2 \rightarrow \bot
\]

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

\[
E_1 \rightarrow \bot
\]

\[
E_2 \rightarrow \bot
\]

\[
E_1 \& E_2 \rightarrow \bot
\]

\[
E_1 \& E_2 \rightarrow \bot
\]

\[
E_1 \rightarrow v_1 \quad E_2 \rightarrow v_2
\]

if \( v_1, v_2 \neq \bot \)

\[
E_1 \& E_2 \rightarrow v_1 \land v_2
\]

\[
E_1 \rightarrow \bot
\]

\[
E_2 \rightarrow \bot
\]

\[
E_1 \mid E_2 \rightarrow \bot
\]

\[
E_1 \mid E_2 \rightarrow \bot
\]

\[
E_1 \rightarrow v_1 \quad E_2 \rightarrow v_2
\]

if \( v_1, v_2 \neq \bot \)

\[
E_1 \mid E_2 \rightarrow v_1 \lor v_2
\]

\[
E \rightarrow \bot
\]

\[
\neg E \rightarrow \bot
\]

\[
E \rightarrow v
\]

if \( v \neq \bot \)

\[
\neg E \rightarrow \neg v
\]

\[
E_1 \rightarrow \bot
\]

\[
E_2 \rightarrow \bot
\]

\[
E_1 \ast E_2 \rightarrow \bot
\]

\[
E_1 \ast E_2 \rightarrow \bot
\]

\[
E_1 \rightarrow v_1 \quad E_2 \rightarrow v_2
\]

if \( v_1, v_2 \neq \bot \)

\[
E_1 \ast E_2 \rightarrow v_1 \equiv v_2
\]

\[
E_1 \rightarrow \bot
\]

\[
E_2 \rightarrow \bot
\]

\[
E_1 > E_2 \rightarrow \bot
\]

\[
E_1 > E_2 \rightarrow \bot
\]
\[ E_1 \rightarrow v_1 \quad E_2 \rightarrow v_2 \]

\[ E_1 > E_2 \rightarrow v_1 > v_2 \]

if \( v_1, v_2 \neq \bot \)

\[ E_1 \rightarrow \bot \quad E_2 \rightarrow \bot \]

\[ E_1 < E_2 \rightarrow \bot \]

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

**Example 3.8** \( 5+(3/0) \rightarrow \bot \), since \( 3/0 \rightarrow \bot \).

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. We used classes with member functions, recursion etc. Such an approach is questionable; we explained the language ePL by using either a complex mathematical construction or another high-level programming language, Java. How are we going to explain Java? By reduction to another high-level programming language?

**Lack of realism** The substitution operation that we employed in dynamic semantics is far away from what happens in real programming systems. We can therefore not hope to properly account for the space and time complexity of programs using dynamic semantics. The aim in denotational semantics is to describe the meaning of programs as mathematical values, and not as a process, and therefore denotational semantics would have to be significantly modified to account for the resources that executing programs consume.
In this chapter, we are aiming for a simpler, lower-level description of the meaning of ePL programs, which will allow us to realistically capture the run-time 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.

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

\[
E_1 \mapsto v_1 \quad E_2 \mapsto v_2 \\
+ [E_1, E_2] \mapsto v_1 + v_2
\]

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 3.5.2 presents the compilation process from ePL to eVML, Section 3.5.3 shows how the machine runs, and Section 3.5.4 asks whether we could prove that such an implementation is correct. Section 3.5.5 shows how eVML can be implemented in Java, and Section 3.5.6 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.

\[
\begin{align*}
\text{DONE} & \quad s \\
\text{LDCI} \ i \ . \ s & \quad \text{LDCB} \ b \ . \ s
\end{align*}
\]

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”.

\[
E_1 \mapsto v_1 \quad E_2 \mapsto v_2 \\
+ [E_1, E_2] \mapsto v_1 + v_2
\]
The remaining ten rules introduce machine instructions corresponding to each of the operators in ePL.

\[
\begin{array}{cccc}
\text{PLUS}.s & \text{MINUS}.s & \text{TIMES}.s & \text{DIV}.s \\
\text{AND}.s & \text{OR}.s & \text{NOT}.s & \\
\text{LT}.s & \text{GT}.s & \text{EQ}.s & 
\end{array}
\]

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

\[
[\text{LDCI} \ 1, \text{LDCI} \ 2, \text{PLUS}, \text{DONE}]
\]

represents a valid eVML program.

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

```java
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.

```java
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`.

```java
public class OPCODES {
    public static final byte LDCI = 1,
    LDCB = 2,
    ...
}
```
3.5. A VIRTUAL MACHINE FOR EPL

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();

3.5.2 Compiling ePL to eVML

The translation from ePL to eVML is accomplished by a function

\[ \text{ePL} \rightarrow \text{eVML} \]

which appends the instruction DONE to the result of the auxiliary translation function \( \rightarrow \).

\[ E \rightarrow s \]

\[ E \rightarrow s \text{.DONE} \]

The auxiliary translation function \( \rightarrow \) is defined by the following rules.

\[ n \rightarrow \text{LDCI } n \quad \text{true} \rightarrow \text{LDCB true} \quad \text{false} \rightarrow \text{LDCB false} \]

\[ E_1 \rightarrow s_1 \quad E_2 \rightarrow s_2 \]

\[ E_1 + E_2 \rightarrow s_1.s_2 \text{.PLUS} \]

\[ E_1 \rightarrow s_1 \quad E_2 \rightarrow s_2 \]

\[ E_1 / E_2 \rightarrow s_1.s_2 \text{.DIV} \]

\[ E_1 \rightarrow s_1 \quad E_2 \rightarrow s_2 \]

\[ E \rightarrow s \]

\[ E \rightarrow s \text{.NOT} \]

\[ E_1 \rightarrow s_1 \quad E_2 \rightarrow s_2 \]

\[ E_1 \rightarrow s_1 \quad E_2 \rightarrow s_2 \]

\[ E_1 \rightarrow s_1 \quad E_2 \rightarrow s_2 \]

\[ E_1 \rightarrow s_1 \quad E_2 \rightarrow s_2 \]

\[ E_1 \rightarrow s_1 \quad E_2 \rightarrow s_2 \]

\[ E_1 \rightarrow s_1 \quad E_2 \rightarrow s_2 \]

\[ E_1 \rightarrow s_1 \quad E_2 \rightarrow s_2 \]

\[ E_1 \rightarrow s_1 \quad E_2 \rightarrow s_2 \]

Example 3.11 Using the usual derivation trees, we can show

\( (1 + 2) \times 3 \rightarrow [\text{LDCI 1, LDCI 2, PLUS, LDCI 3, TIMES, DONE}] \), and

\( 1 + (2 \times 3) \rightarrow [\text{LDCI 1, LDCI 2, LDCI 3, TIMES, PLUS, DONE}] \).
Observe that the machine code places the operator of an arithmetic expres-
sion after its arguments. This way of writing expressions is called Reverse Polish
Notation, in honor of its inventor, the Polish logician Jan Łukasiewicz, and is
designed for efficient execution on automata.

Our compiler for ePL translates a given ePL expression—as usual repre-
sented by its syntax tree—to an INSTRUCTION array.

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

3.5.3 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, $os = ⟨10, 20, true⟩$ 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
$\Rightarrow_s$, which transforms machine states to machine states, and which is defined
by the following twelve rules.

\[
\begin{align*}
\text{s}(pc) &= \text{LDCI } i \\
(os, pc) \Rightarrow_s (i.os, pc + 1)
\end{align*}
\]  

\[
\begin{align*}
\text{s}(pc) &= \text{LDCB } b \\
(os, pc) \Rightarrow_s (b.os, pc + 1)
\end{align*}
\]

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.

\[
\begin{align*}
\text{s}(pc) &= \text{PLUS} \\
(i_2.i_1.os, pc) \Rightarrow_s (i_1 + i_2.os, pc + 1)
\end{align*}
\]

\[
\begin{align*}
\text{s}(pc) &= \text{MINUS} \\
(i_2.i_1.os, pc) \Rightarrow_s (i_1 - i_2.os, pc + 1)
\end{align*}
\]
3.5. A VIRTUAL MACHINE FOR EPL

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.

\[ s(pc) = \text{TIMES} \]
\[
(i_2 \cdot i_1, os, pc) \xrightarrow{\text{s}} (i_1 \cdot i_2, os, pc + 1)
\]

\[ s(pc) = \text{DIV} \]
\[
(i_2 \div i_1, os, pc) \xrightarrow{\text{s}} (i_1 / i_2, os, pc + 1)
\]

\[ s(pc) = \text{AND} \]
\[
(b_2 \land b_1, os, pc) \xrightarrow{\text{s}} (b_1 \land b_2, os, pc + 1)
\]

\[ s(pc) = \text{OR} \]
\[
(b_2 \lor b_1, os, pc) \xrightarrow{\text{s}} (b_1 \lor b_2, os, pc + 1)
\]

\[ s(pc) = \text{NOT} \]
\[
(b, os, pc) \xrightarrow{\text{s}} (\neg b, os, pc + 1)
\]

\[ s(pc) = \text{LT} \]
\[
(i_2 \cdot i_1, os, pc) \xrightarrow{\text{s}} (i_1 < i_2, os, pc + 1)
\]

\[ s(pc) = \text{GT} \]
\[
(i_2 \cdot i_1, os, pc) \xrightarrow{\text{s}} (i_1 > i_2, os, pc + 1)
\]

\[ s(pc) = \text{EQ} \]
\[
(i_2 \cdot i_1, os, pc) \xrightarrow{\text{s}} (i_1 \equiv i_2, os, pc + 1)
\]

Note that the behavior of the transition function is entirely determined by the instruction, to which \(pc\) points. Like the dynamic semantics \(\Rightarrow\) of \text{ePL}, the evaluation gets stuck if none of the rules apply, which is the case for division by
CHAPTER 3. THE LANGUAGE EPL

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 \(s(pc) = \text{DONE}\). 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

\[ R(M_s) = v, \text{ where } ((\langle \rangle, 0) \Rightarrow_s ((v, os), pc), \text{ and } s(pc) = \text{DONE} \]

Example 3.12 The execution of the eVML program

\([\text{LDCI 10, LDCI 20, PLUS, LDCI 6, TIMES, DONE}]\]

is represented by the following sequence of states:

\[ ((\langle \rangle, 0) \Rightarrow ((10), 1) \Rightarrow ((20, 10), 2) \Rightarrow ((30), 3) \Rightarrow ((6, 30), 4) \Rightarrow ((180), 5) \]

At this point, the machine has reached an end configuration, because \(s(5) = \text{DONE}\). The result of the computation is therefore 180.

3.5.4 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 \(E \Rightarrow v\). Does an instruction sequence \(s\) exist such that \(E \Rightarrow s\), 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.

3.5.5 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.

```java
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:
                // Code for LDCI
            case OPCODES.PLUS:
                // Code for PLUS
            case OPCODES.TIMES:
                // Code for TIMES
            default:
                // Code for other opcodes
            }
        }
    }
}
```
3.5. A VIRTUAL MACHINE FOR EPL

```java
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.

### 3.5.6 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.
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. Since the compiler is written in Java, we first need to translate it to JVM code.

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.ep1, as depicted in the following T-diagrams.

Example 3.13 Using the compiler eplc and the emulator epl, both written in Java, we can execute a given ePL program bill.ep1 as follows:

> javac eplc.java
> javac epl.java
> java eplc bill.ep1
> ls bill.*
bill.epl bill.evml
> java epl bill
249
Bibliography