02—ePL: An Overture

CS4215: Programming Language Implementation

Martin Henz

January 20, 2012

Generated on Thursday 19 January, 2012, 22:56
1. The Syntax of ePL
2. Dynamic Semantics of ePL Programs
3. Static Semantics for ePL
4. A Virtual Machine for ePL
1. The Syntax of ePL
2. Dynamic Semantics of ePL Programs
3. Static Semantics for ePL
4. A Virtual Machine for ePL
Syntax of ePL

\[
\begin{align*}
&n & \text{true} & \text{false} \\
&E & E_1 & E_2 \\
&p_1[E] & p_2[E_1, E_2] \\
\end{align*}
\]

\[
P_1 = \{\}\text{,} \\
P_2 = \{|, &, +, -, *, /, =, >, <\}.
\]
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 $>[[+1,*[2,3]],-[10,4]]$
Examples

42

\[-15 \times (7 + 2)\]

\[\text{false} \land \text{true} \lor \text{false}\]

\[17 < 20 - 4 \land 10 = 4 + 11\]
Implementation of Syntax

Lexical analysis
Language syntax is typically implemented in a two-step process: (1) lexical analysis, (2) parsing

Parsing
Construct a tree-like data structure that corresponds to the structure of the program
public interface Expression {}
```java
public class BoolConstant implements Expression {
    public String value;
    public BoolConstant(String v) {
        value = v;
    }
}
```
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;
  }
}
1 The Syntax of ePL

2 Dynamic Semantics of ePL Programs

3 Static Semantics for ePL

4 A Virtual Machine for ePL
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, we denote values by $v$. 
Contraction

\[
\begin{align*}
\text{[OpVals}_1] & : \quad p_1[v_1] >_{ePL} v \\
\text{[OpVals}_2] & : \quad p_2[v_1, v_2] >_{ePL} v
\end{align*}
\]

One instance of the second rule is:

\[
+[1, 1] >_{ePL} 2
\]
One-Step Evaluation

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

[Contraction]

\[ E \xhookleftarrow{\text{ePL}} E' \]

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

[OpArg₁]

\[ p₁[E] \xrightarrow{\text{ePL}} p₁[E'] \]

\[ E₁ \xhookleftarrow{\text{ePL}} E₁' \]

[OpArg₂]

\[ p₂[E₁, E₂] \xhookleftarrow{\text{ePL}} p₂[E₁', E₂] \]

\[ E₂ \xrightarrow{\text{ePL}} E₂' \]

[OpArg₃]

\[ p₂[E₁, E₂] \xrightarrow{\text{ePL}} p₂[E₁, E₂'] \]
One-step evaluation does not prescribe the order in which a given ePL expression is evaluated. Both of the following statements hold:

\[
3 \times 2 + 4 \times 5 \xrightarrow{\text{ePL}} 3 \times 2 + 20 \\
3 \times 2 + 4 \times 5 \xrightarrow{\text{ePL}} 6 + 4 \times 5
\]
Evaluation is the reflexive transitive closure of one-step evaluation.

\[
E_1 \xrightarrow{\text{ePL}} E_2 \quad \Rightarrow \quad [\xrightarrow{\text{ePL}^*}] \quad [\xrightarrow{\text{ePL}^*_B}]
\]

\[
E_1 \xrightarrow{\text{ePL}^*} E_2 \quad \Rightarrow \quad [\xrightarrow{\text{ePL}^*_R}]
\]

\[
E_1 \xrightarrow{\text{ePL}^*} E_2 \quad E_2 \xrightarrow{\text{ePL}^*} E_3 \quad \Rightarrow \quad [\xrightarrow{\text{ePL}^*_T}]
\]

\[
E_1 \xrightarrow{\text{ePL}^*} E_3
\]
Implementation

Idea

Keep reducing expression until it is not reducible any longer

Code snippet

```java
static public Expression evaluate(Expression exp) {
    return reducible(exp) ? evaluate(oneStep(exp)) : exp;
}
```
1. The Syntax of ePL
2. Dynamic Semantics of ePL Programs
3. Static Semantics for ePL
4. A Virtual Machine 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.
Typing Relation

The set of well-typed expressions is defined by the binary typing relation

```
"::": ePL → {int, bool}
```

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

Examples

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

but:
- true + 1 1 : t
- $3 + 1 * 5 : \text{bool}$

do not hold for any $t$
Typing Relation

\[ n : \text{int} \]

\[ \text{true} : \text{bool} \]

\[ \text{false} : \text{bool} \]

\[ E : \text{bool} \]

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

\[ E_1 : t_1 \quad E_2 : t_2 \]

\[ p[E_1, E_2] : t \]
## Types of Primitive Operations

<table>
<thead>
<tr>
<th>$p$</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$+$</code></td>
<td><code>int</code></td>
<td><code>int</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>$-$</code></td>
<td><code>int</code></td>
<td><code>int</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>$*$</code></td>
<td><code>int</code></td>
<td><code>int</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>$/</code></td>
<td><code>int</code></td>
<td><code>int</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>$\&amp;$</code></td>
<td><code>bool</code></td>
<td><code>bool</code></td>
<td><code>bool</code></td>
</tr>
<tr>
<td>`$</td>
<td>$`</td>
<td><code>bool</code></td>
<td><code>bool</code></td>
</tr>
<tr>
<td><code>$\\$</code></td>
<td><code>bool</code></td>
<td></td>
<td><code>bool</code></td>
</tr>
<tr>
<td><code>$=$$</code></td>
<td><code>int</code></td>
<td><code>int</code></td>
<td><code>bool</code></td>
</tr>
<tr>
<td><code>$&lt;$$</code></td>
<td><code>int</code></td>
<td><code>int</code></td>
<td><code>bool</code></td>
</tr>
<tr>
<td><code>$&gt;$</code></td>
<td><code>int</code></td>
<td><code>int</code></td>
<td><code>bool</code></td>
</tr>
</tbody>
</table>
An expression $E$ is well-typed, if there is a type $t$ such that $E : t$. 
A Proof

\[
\begin{array}{ll}
\mathbf{2} : \text{int} & \quad \mathbf{3} : \text{int} \\
\hline
\mathbf{2} \times \mathbf{3} : \text{int} & \quad \mathbf{7} : \text{int} \\
\hline
\mathbf{2} \times \mathbf{3} > \mathbf{7} : \text{bool}
\end{array}
\]
Implementation

**Idea**
Compute the type of an expression *bottom-up*, starting from the constants at the leaves of the syntax tree.

**Checking**
At each node, check that the components have the right type wrt the operator.
Type safety is a property of a given language with a given static and dynamic semantics. In a type-safe language, certain problems are guaranteed not to occur at runtime for well-typed programs.

Problems:

- No progress,
- No preservation
The Syntax of ePL

Dynamic Semantics of ePL Programs

Static Semantics for ePL

A Virtual Machine for ePL
Motivation

- How do we remember intermediate results?
- How do we know what to do next?

Idea: Translate ePL to a machine language. Execute machine code using an emulator.
Definition of eVML

DONE

LDCI \ i \ . \ s

LDCB \ b \ . \ s
Definition of eVML (cont’d)

\[
\begin{align*}
S \quad & \quad S \quad & \quad S \quad & \quad S \\
PLUS.s \quad & \quad MINUS.s \quad & \quad TIMES.s \quad & \quad DIV.s \\
AND.s \quad & \quad OR.s \quad & \quad NOT.s \\
LT.s \quad & \quad GT.s \quad & \quad EQ.s
\end{align*}
\]
The instruction sequence

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

represents a valid eVML program.
Compiling ePL to eVML

: ePL → eVML

\[ E \rightarrow s \]

\[ Es.\text{DONE} \]
Compiling ePL to eVML (cont’d)

\[ n \rightarrow \text{LDCI } n \]

\[ \text{false} \rightarrow \text{LDCB } \text{false} \]

\[ \text{true} \rightarrow \text{LDCB } \text{true} \]
Compiling **ePL** to eVML (cont’d)

\[
\begin{align*}
E_1 & \rightarrow s_1 \quad \quad E_2 \rightarrow s_2 \\
\hline
E_1 + E_2 & \rightarrow s_1.s_2.PLUS \\
E_1 & \rightarrow s_1 \quad \quad E_2 \rightarrow s_2 \\
\hline
E_1 / E_2 & \rightarrow s_1.s_2.DIV \\
\ldots
\end{align*}
\]
Examples

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

\[1 + (2 \times 3)\]
\[\text{[LDCI 1, LDCI 2, LDCI 3, TIMES, PLUS, DONE]}\].
Executing eVML Code

Registers:

- \textit{pc}: program counter,
- \textit{os}: operand stack
Example

\[ pc = 2 \]

\[ s = [\text{LDCI 1, LDCI 2, PLUS, LDCI 3, TIMES, DONE}] \]

\[ s(pc) = \text{PLUS} \]
Transition Function

\[
s(pc) = \text{LDCI } i \\
\hline
(os, pc) \xrightarrow{s} (i.os, pc + 1)
\]

\[
s(pc) = \text{LDCB } b \\
\hline
(os, pc) \xrightarrow{s} (b.os, pc + 1)
\]
Transition Function (cont’d)

\[ s(pc) = \text{PLUS} \]

\[ (i_2 \cdot i_1.o, pc) \Rightarrow_s (i_1 + i_2.o, pc + 1) \]
End Configuration

\[ s(pc) = \text{DONE} \]

The result of the computation can be found on top of the operand stack of the end configuration.

\[ R(M_s) = v, \text{ where } (\langle \rangle, 0) \xRightarrow{s}^* (\langle v.os \rangle, pc), \text{ and } s(pc) = \text{DONE} \]
Example

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

\[ (\langle \rangle, 0) \Rightarrow (\langle 10 \rangle, 1) \Rightarrow (\langle 20, 10 \rangle, 2) \Rightarrow \]

\[ (\langle 30 \rangle, 3) \Rightarrow (\langle 6, 30 \rangle, 4) \Rightarrow (\langle 180 \rangle, 5) \]
Implementation

Registers
Keep registers in local variables

Execution
Use a switch statement to interpret each instruction
public static Value run(INSTRUCTION[] instructionArray) {
    int pc = 0;
    Stack<Value> os = new Stack<Value>();
    loop:
    while (true) {
        INSTRUCTION i = instructionArray[pc];
        switch (i.OPCODE) {
            case OPCODES.LDCI: os.push(
                new IntValue(Integer.parseInt(i.VALUE)));
                pc++;
                break;
            ...}
        }
    }
}
Virtual Machine as Interpreter

```
        ePL
      Java → JVM
       x86

        ePL
      Java → JVM
       x86

        bill
       ePL
       JVM
       x86
       x86
```
Compiling the ePL Compiler

```
<table>
<thead>
<tr>
<th>ePL</th>
<th>eVML</th>
<th>ePL</th>
<th>eVML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>Java</td>
<td>JVM</td>
<td>JVM</td>
</tr>
<tr>
<td>x86</td>
<td></td>
<td>x86</td>
<td></td>
</tr>
</tbody>
</table>
```
Compiling the eVML Emulator
Compilation and Execution of Example Program
Next Week

- Introduction of simPL