CS422 - Programming Language Design

Operational Semantics

Grigore Roșu
Department of Computer Science
University of Illinois at Urbana-Champaign
By an operational semantics of a programming language, one typically understands a collection of rules specifying how its expressions and statements are evaluated/executed. These rules say how a possible implementation of a programming language should “operate”.

There is no definite agreement on how an operational semantics of a language should be given, because any description of a programming language which is rigorous enough to quickly lead to a correct implementation of the language can be considered to be a valid operational semantics.
Let us next consider a very simple non-procedural imperative language which has arithmetic and boolean expressions, conditionals and while loops.

A program is a sequence of statements followed by an expression. The expression is evaluated in the state obtained after evaluating all the statements and its result is returned as the result of the evaluation of the entire program.

Formally, its syntax is given as a context-free grammar (CFG) as follows:
\[
\begin{align*}
\text{Name} &::= \text{standard identifiers} \\
\text{AExp} &::= \text{Name} \mid 1 \mid 2 \mid 3 \mid \ldots \\
&\quad AExp + AExp \mid AExp - AExp \mid AExp \ast AExp \mid AExp/\text{AExp} \\
\text{BExp} &::= \text{true} \mid \text{false} \mid AExp \leq AExp \mid AExp \geq AExp \mid AExp == AExp \\
&\quad BExp \text{ and } BExp \mid BExp \text{ or } BExp \mid \text{not } BExp \\
\text{Stmt} &::= \text{skip} \mid \text{Name} := \text{AExp} \mid \text{Stmt} ; \text{Stmt} \mid \{\text{Stmt}\} \\
&\quad \text{if } BExp \text{ then } \text{Stmt} \text{ else } \text{Stmt} \mid \text{while } BExp \text{ Stmt} \\
\text{Pgm} &::= \text{Stmt} ; \text{AExp}
\end{align*}
\]
Traditionally, the operational semantics of a language, typically called "structural" (and abbreviated \textit{SOS}) because it is defined inductively over the structure of the syntax, is given by defining a transition relation on configurations. A configuration consists of a term over the syntax of the language and a state.

For this simple imperative language, a \textit{state} is a map from names to integer numbers $\text{Name} \rightarrow \text{Int}$. We let $\sigma$, $\sigma'$, etc., denote states. If $\sigma$ is a state and $x$ a name, then we let $\sigma[x]$ or $\sigma(x)$ denote the integer value to which $\sigma$ maps $x$. Moreover, if $x$ is a name and $i$ an integer, then we let $\sigma[x \leftarrow i]$ denote the function $\text{Name} \rightarrow \text{Int}$ defined as follows:
\[
\sigma[x \leftarrow i](y) = \begin{cases} 
\sigma(x) & \text{if } x \neq y, \\
i & \text{if } x = y.
\end{cases}
\]

We also let \(\emptyset\) denote the initial state. One has two options at this point: one is to consider that \(\emptyset[x]\) is some default initial value for any \(x\), another is to consider states as partial functions and thus let \(\emptyset[x]\) undefined.

**SOS Rules for Arithmetic Expressions**

We introduce a relation \(\langle \_ , \_ \rangle \rightarrow \_\) on triples of arithmetic expressions, states and integers, with the following intuition: \(\langle a, \sigma \rangle \rightarrow i\), also called a transition, states that the arithmetic expression \(a\) evaluates/executes/transits to the integer \(i\) in state \(\sigma\). The SOS for arithmetic expressions can then be given as a
collection of *parametric rules* of the form:

\[
\text{transition}_1, \text{transition}_2, \ldots, \text{transition}_k
\]

\[
\text{transition}
\]

The intuition here is that *transition* is possible whenever \(\text{transition}_1, \text{transition}_2, \ldots, \text{transition}_k\) are possible. We may also say that *transition* is derivable, or can be inferred, from \(\text{transition}_1, \text{transition}_2, \ldots, \text{transition}_k\). This reflects the fact that, like our previous equational semantics, SOS can also be viewed as a logic system, where one can deduce possible behaviors of programs.

If \(k = 0\), then we simply write

\[
\cdot
\]

\[
\text{transition}
\]

In the case of our simple language, the transition relation is going to be deterministic, in the sense that whenever \(\langle a, \sigma \rangle \rightarrow i_1\) and \(\langle a, \sigma \rangle \rightarrow i_2\) can be deduced, then \(i_1 = i_2\), because our language is deterministic. However, in the context of concurrent languages, as
we will see later, $\langle a, \sigma \rangle \rightarrow i$ states that $a$ may possibly evaluate to $i$ in state $\sigma$, but it may also evaluate to other integers.

We need to inductively define the transition relation for each language construct for arithmetic expressions. Since Name and Int are syntactic subcategories of AExp, we start by introducing the following two SOS rules, one for names and the other for integers:

\[
\begin{align*}
\langle x, \sigma \rangle &\rightarrow \sigma[x] \quad (1) \\
\langle i, \sigma \rangle &\rightarrow i \quad (2)
\end{align*}
\]

We next give the SOS rules for the arithmetic operations of addition, subtraction, multiplication and division:
\( \langle a_1, \sigma \rangle \rightarrow i_1, \langle a_2, \sigma \rangle \rightarrow i_2 \) where \( i \) is the sum of \( i_1 \) and \( i_2 \) \hspace{1cm} (3)

\( \langle a_1 + a_2, \sigma \rangle \rightarrow i \)

\( \langle a_1, \sigma \rangle \rightarrow i_1, \langle a_2, \sigma \rangle \rightarrow i_2 \) where \( i \) is \( i_1 \) minus \( i_2 \) \hspace{1cm} (4)

\( \langle a_1 - a_2, \sigma \rangle \rightarrow i \)

\( \langle a_1, \sigma \rangle \rightarrow i_1, \langle a_2, \sigma \rangle \rightarrow i_2 \) where \( i \) is \( i_1 \) times \( i_2 \) \hspace{1cm} (5)

\( \langle a_1 * a_2, \sigma \rangle \rightarrow i \)

\( \langle a_1 / a_2, \sigma \rangle \rightarrow i \) where \( i \) is the quotient of \( i_1 \) by \( i_2 \) \hspace{1cm} (6)

Note that all these rules are *parametric*, that is, they can be viewed as collections of concrete *instance rules*. A possible instance of rule (3) can be the following, which, of course, seems problematic:
The rule above is indeed a correct instance of (3). However, one will never be able to infer \(\langle 2, \sigma \rangle \rightarrow 9\), so this rule cannot be applied in a correct inference.

The following is a correct inference, where \(x\) and \(y\) are any names and \(\sigma\) is any state with \(\sigma[x] = \sigma[y] = 1\):

\[
\begin{align*}
\langle x, \sigma \rangle &\rightarrow 1, \quad \langle x, \sigma \rangle \rightarrow 1, \\
\langle y, \sigma \rangle &\rightarrow 1, \quad \langle y, \sigma \rangle \rightarrow 1, \\
\langle x - (y \times x + 2), \sigma \rangle &\rightarrow -2
\end{align*}
\]

The proof above can be regarded as an upside-down tree, with dots as leaves and instances of SOS rules as nodes. This way, we have a way to mathematically derive facts about programs within their
semantics. One can even prove behavioral equivalence of programs.

### SOS Rules for Boolean Expressions

We can now similarly add transitions of the form $\langle b, \sigma \rangle \rightarrow t$, where $b$ is a boolean expression and $t$ is a truth value in the set $\{\text{true}, \text{false}\}$:

\[
\begin{align*}
\langle \text{true}, \sigma \rangle & \rightarrow \text{true} \tag{7} \\
\langle \text{false}, \sigma \rangle & \rightarrow \text{false} \tag{8}
\end{align*}
\]

\[
\begin{align*}
\langle a_1, \sigma \rangle & \rightarrow i_1, \quad \langle a_2, \sigma \rangle \rightarrow i_2 \quad \text{where } i_1 \text{ less than or equal to } i_2 \tag{9}
\end{align*}
\]

\[
\langle a_1 \leq a_2, \sigma \rangle \rightarrow \text{true}
\]
\[ \langle a_1, \sigma \rangle \rightarrow i_1, \langle a_2, \sigma \rangle \rightarrow i_2 \quad \text{where } i_1 \text{ larger than } i_2 \] (10)

\[ \langle a_1, \sigma \rangle \rightarrow i_1, \langle a_2, \sigma \rangle \rightarrow i_2 \quad \text{for } i_1 \text{ larger than or equal to } i_2 \] (11)

\[ \langle a_1, \sigma \rangle \rightarrow i_1, \langle a_2, \sigma \rangle \rightarrow i_2 \quad \text{where } i_1 \text{ smaller than } i_2 \] (12)

\[ \langle a_1, \sigma \rangle \rightarrow i, \langle a_2, \sigma \rangle \rightarrow i \] (13)

\[ \langle a_1 \text{ equals } a_2, \sigma \rangle \rightarrow \text{true} \]

\[ \langle b_1, \sigma \rangle \rightarrow \text{true}, \langle b_2, \sigma \rangle \rightarrow \text{true} \] (15)

\[ \langle b_1 \text{ and } b_2, \sigma \rangle \rightarrow \text{true} \]
\[
\langle b_1, \sigma \rangle \rightarrow t_1, \quad \langle b_2, \sigma \rangle \rightarrow t_2 \quad \frac{\langle b_1 \text{ and } b_2, \sigma \rangle \rightarrow \text{false}}{\text{where } t_1 \text{ or } t_2 \text{ is false}} \quad (16)
\]

\[
\frac{\langle b_1, \sigma \rangle \rightarrow \text{false} \quad \langle b_2, \sigma \rangle \rightarrow \text{false}}{\langle b_1 \text{ or } b_2, \sigma \rangle \rightarrow \text{false}} \quad (17)
\]

\[
\langle b_1, \sigma \rangle \rightarrow t_1, \quad \langle b_2, \sigma \rangle \rightarrow t_2 \quad \frac{\langle b_1 \text{ or } b_2, \sigma \rangle \rightarrow \text{true}}{\text{where } t_1 \text{ or } t_2 \text{ is } \text{true}} \quad (18)
\]

\[
\langle b, \sigma \rangle \rightarrow \text{false} \quad \frac{\langle \text{not } b, \sigma \rangle \rightarrow \text{true}}{\text{(19)}}
\]

\[
\langle b, \sigma \rangle \rightarrow \text{true} \quad \frac{\langle \text{not } b, \sigma \rangle \rightarrow \text{false}}{\text{(20)}}
\]
SOS Rules for Statements

Statements in our simple imperative language change the state, so we need to introduce a new transition relation of the form \(\langle s, \sigma \rangle \rightarrow \sigma'\), where \(s\) is a statement and \(\sigma, \sigma'\) are states. The SOS rules for statements are then:

\[
\begin{align*}
\langle \text{skip}, \sigma \rangle &\rightarrow \sigma & (21) \\
\langle a, \sigma \rangle &\rightarrow i \\
\langle x = a, \sigma \rangle &\rightarrow \sigma[x \leftarrow i] & (22) \\
\langle s_1, \sigma \rangle &\rightarrow \sigma'', \langle s_2, \sigma'' \rangle \rightarrow \sigma' \\
\langle s_1; s_2, \sigma \rangle &\rightarrow \sigma' & (23)
\end{align*}
\]
\begin{align}
\langle s, \sigma \rangle &\rightarrow \sigma' \\
\langle \{s\}, \sigma \rangle &\rightarrow \sigma' \\
\langle b, \sigma \rangle &\rightarrow \text{true}, \langle s_1, \sigma \rangle \rightarrow \sigma' \\
\langle \text{if } b \text{ then } s_1 \text{ else } s_2, \sigma \rangle &\rightarrow \sigma' \\
\langle b, \sigma \rangle &\rightarrow \text{false}, \langle s_2, \sigma \rangle \rightarrow \sigma' \\
\langle \text{if } b \text{ then } s_1 \text{ else } s_2, \sigma \rangle &\rightarrow \sigma' \\
\langle b, \sigma \rangle &\rightarrow \text{false} \\
\langle \text{while } b \ s, \sigma \rangle &\rightarrow \sigma \\
\langle b, \sigma \rangle &\rightarrow \text{true}, \langle s, \sigma \rangle \rightarrow \sigma'', \langle \text{while } b \ s, \sigma'' \rangle \rightarrow \sigma' \\
\langle \text{while } b \ s, \sigma \rangle &\rightarrow \sigma'
\end{align}
SOS Rules for Programs

Programs are always executed in the initial state, so we can define their SOS rule as follows:

\[
\langle s, \emptyset \rangle \rightarrow \sigma, \quad \langle a, \sigma \rangle \rightarrow 1 \\
\langle s; a \rangle \rightarrow i
\]

(29)