Introduction to Structural Operational Semantics

Patricia Hill
School of Computing
University of Leeds, United Kingdom

Roberto Bagnara
Department of Mathematics
University of Parma, Italy
Syntax and Semantics

- **Syntax** is concerned with the form of expressions that are allowed in a language.

- **Semantics** describes what should be the result of executing or evaluating the program.

- **Static semantics** describes what kinds of syntactically correct expressions are executable.

- **Dynamic semantics** describes the run-time behaviour including the expected results.
Approaches to Semantics

- **Operational Semantics**: The meaning of a program in terms of how it executes on an abstract machine. This course will follow the approach called “Structural Operational Semantics” (SOS), proposed by G. Plotkin.
  Useful for modelling the execution behaviour of a program.

- **Denotational Semantics**: The mathematical meaning of a program and models the expected result rather than how the result might be obtained. This approach was first proposed by C. Strachey and later formalised by D. Scott.
  Useful for understanding the internal logic of a program.
**APPROACHES TO SEMANTICS, CONT.**

- **Axiomatic Semantics**: Provides correctness assertions for each program construct. The approach was developed by R.W. Floyd and C.A.R. Hoare.

  Useful for verifying that a program’s computed results are correct with respect to the specification.
**IMP: AN IMPERATIVE LANGUAGE**

**Imp** is a simple imperative language for elementary arithmetic with just a conditional command and a while loop command (see Chapter 2 of Winskel). We present here:

- the abstract syntax (i.e., the sentences to which we will associate a **semantics**);
- the operational semantics of expressions, i.e., the rules governing their evaluation;
- the operational semantics of commands, i.e., the rules governing their evaluation (execution).
IMP: THE ABSTRACT SYNTAX

The basic syntactic sets are:

**Integers:** \( m \in \text{Int} \overset{\text{def}}{=} \{\ldots, -2, -1, 0, 1, 2, \ldots\} \);

**Booleans:** \( t \in \text{Bool} \overset{\text{def}}{=} \{\text{tt, ff}\} \);

**Variables:** \( x \in \text{Var} \overset{\text{def}}{=} \{x_1, x_2, \ldots\} \);

**Arithmetic expressions:** \( a \in \text{Aexp} \);

**Boolean expressions:** \( b \in \text{Bexp} \);

**Commands:** \( c \in \text{Com} \).

We shall regard the variables \( x_1, x_2, \ldots \) as locations.

The \( m, t, x, a, b \) and \( c \) are called syntactic metavariables: they denote a generic element of the respective syntactic category.
**IMP: The Language Formation Rules**

Aexp:

\[ a ::= m \mid x \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \ast a_1 \]

Bexp:

\[ b ::= t \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \text{b}_0 \text{ and } b_1 \mid \text{b}_0 \text{ or } b_1 \mid \text{not } b_1 \]

Com:

\[ c ::= \text{skip} \mid x ::= a \mid c_0; c_1 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \]
IMP: Examples of Syntactically Correct Sentences

Aexp:

\[(3 + x) - 5 + (2 \times y)\]

Bexp:

\[0 \geq 1\]

\[(3 + x) - 5 \leq 4\]

\[(\text{not} x = 3) \text{ or } y = 2\]

Com:

\[x := 0; y := 0\]

\[\text{while } x \leq 100 \text{ do } x := x + 1\]

\[\text{if } x \leq y \text{ skip else } y := x\]
EXECUTION STORES

A store $\sigma$ maps each variable (location) to a value.

In the language $\textbf{Imp}$, these values must be numbers. Let

$$\text{Stores} \overset{\text{def}}{=} \{ \sigma : V \rightarrow \text{Int} \mid V \subseteq_f \text{Var} \}.$$  

Example: Let

$$\sigma = \{(x, 3), (y, 2)\},$$

so that

$$\sigma(x) = 3$$

$$\sigma(y) = 2.$$  

The store $\sigma$ can (and will) be more compactly denoted by

$$\{x = 3, y = 2\}.$$
A **non-terminal configuration** is a pair of the form $\langle a, \sigma \rangle$, $\langle b, \sigma \rangle$, or $\langle c, \sigma \rangle$.

$\langle a, \sigma \rangle$, $\langle b, \sigma \rangle$ denote the situation of an arithmetic expression $a$, or a boolean expression $b$ “waiting for evaluation” using the store $\sigma$:

\[
\begin{align*}
\langle x + 2 \times y, \{x = 3, y = 4\} \rangle, \\
\langle x = y + 1, \{x = 3, y = 4\} \rangle.
\end{align*}
\]

$\langle c, \sigma \rangle$ denotes the situation of a command $c$ “waiting for execution” using the store $\sigma$:

\[
\begin{align*}
\langle x := x + 1, \{x = 3, y = 4\} \rangle, \\
\langle \text{skip}, \{x = 3, y = 4\} \rangle.
\end{align*}
\]
Transition Rules

Structural operational semantics provides transition rules for the evaluation of expressions and execution of commands.

A transition for an arithmetic expression $e$ in a store $\sigma$ has one of the forms:

$$
\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle,
$$

$$
\langle e, \sigma \rangle \rightarrow m.
$$

A transition rule is written as

premise

\[
\begin{array}{c}
\text{premise} \\
\hline
\text{conclusion}
\end{array}
\]
EVALUATION OF ARITHMETIC EXPRESSIONS

\[ \langle m, \sigma \rangle \rightarrow m \quad m \in \text{Int}; \]

\[ \sigma(x) = m \quad x \in \text{Var}; \]

\[ \langle x, \sigma \rangle \rightarrow m \]

\[ \langle a_0, \sigma \rangle \rightarrow m_0 \quad \langle a_1, \sigma \rangle \rightarrow m_1 \]

\[ \langle a_0 + a_1, \sigma \rangle \rightarrow m \quad \text{if} \ m = m_0 + m_1; \]

\[ \langle a_0, \sigma \rangle \rightarrow m_0 \quad \langle a_1, \sigma \rangle \rightarrow m_1 \]

\[ \langle a_0 - a_1, \sigma \rangle \rightarrow m \quad \text{if} \ m = m_0 - m_1; \]

\[ \langle a_0, \sigma \rangle \rightarrow m_0 \quad \langle a_1, \sigma \rangle \rightarrow m_1 \]

\[ \langle a_0 \ast a_1, \sigma \rangle \rightarrow m \quad \text{if} \ m = m_0 m_1. \]
Let

\[ \sigma = \{x = 3, y = 4\} \]

Then \( \sigma(x) = 3 \) and \( \sigma(y) = 4 \).

Using the rules we write:

\[
\frac{\sigma(x) = 3}{\langle x, \sigma \rangle \to 3} \quad \frac{\sigma(y) = 4}{\langle y, \sigma \rangle \to 4}
\]

and then

\[
\frac{\langle x, \sigma \rangle \to 3}{\langle x + y, \sigma \rangle \to 7} \quad \frac{\langle y, \sigma \rangle \to 4}{\quad}
\]
The rules for evaluating arithmetic expressions can be combined to form a derivation tree. For example, consider the expression

\[ x + 2 \times y \]

with

\[ \sigma = \{ x = 3, y = 4 \}. \]

We obtain the derivation tree

\[
\begin{align*}
\sigma(x) &= 3 & \sigma(y) &= 4 \\
\langle x, \sigma \rangle &\rightarrow 3 & \langle 2, \sigma \rangle &\rightarrow 2 & \langle y, \sigma \rangle &\rightarrow 4 \\
\langle 2 \times y, \sigma \rangle &\rightarrow 8 & \langle x + 2 \times y, \sigma \rangle &\rightarrow 11
\end{align*}
\]
Evaluation of Boolean Expressions I

Truth values:

\[
\langle \text{tt}, \sigma \rangle \rightarrow \text{tt} \quad \langle \text{ff}, \sigma \rangle \rightarrow \text{ff}
\]

Equality:

\[
\begin{align*}
\langle a_0, \sigma \rangle &\rightarrow m_0 & \langle a_1, \sigma \rangle &\rightarrow m_1 \\
\langle a_0 = a_1, \sigma \rangle &\rightarrow \text{tt} & \text{if } m_0 = m_1; \\
\langle a_0, \sigma \rangle &\rightarrow m_0 & \langle a_1, \sigma \rangle &\rightarrow m_1 \\
\langle a_0 = a_1, \sigma \rangle &\rightarrow \text{ff} & \text{if } m_0 \neq m_1.
\end{align*}
\]
Evaluation of Boolean Expressions II

Inequality:

\[
\begin{align*}
&\langle a_0, \sigma \rangle \rightarrow m_0 \quad \langle a_1, \sigma \rangle \rightarrow m_1 \\
&\frac{\langle a_0 \leq a_1, \sigma \rangle \rightarrow \text{tt}}{} \quad \text{if } m_0 \leq m_1;
\end{align*}
\]

\[
\begin{align*}
&\langle a_0, \sigma \rangle \rightarrow m_0 \quad \langle a_1, \sigma \rangle \rightarrow m_1 \\
&\frac{\langle a_0 \leq a_1, \sigma \rangle \rightarrow \text{ff}}{} \quad \text{if } m_0 > m_1.
\end{align*}
\]
EVALUATION OF BOOLEAN EXPRESSIONS III

Negation:

\[
\begin{align*}
\langle b, \sigma \rangle &\rightarrow \text{tt} \\
\langle \text{not } b, \sigma \rangle &\rightarrow \text{ff} \\
\langle b, \sigma \rangle &\rightarrow \text{ff} \\
\langle \text{not } b, \sigma \rangle &\rightarrow \text{tt}.
\end{align*}
\]

Conjunction:

\[
\begin{align*}
\langle b_0, \sigma \rangle &\rightarrow t_0 \\
\langle b_1, \sigma \rangle &\rightarrow t_1 \\
\langle b_0 \text{ and } b_1, \sigma \rangle &\rightarrow t \\
\text{if } t &\equiv t_0 \land t_1.
\end{align*}
\]

Disjunction:

\[
\begin{align*}
\langle b_0, \sigma \rangle &\rightarrow t_0 \\
\langle b_1, \sigma \rangle &\rightarrow t_1 \\
\langle b_0 \text{ or } b_1, \sigma \rangle &\rightarrow t \\
\text{if } t &\equiv t_0 \lor t_1.
\end{align*}
\]
EVALUATION OF EXPRESSIONS: EXAMPLES

The rules for evaluating arithmetic expressions can be combined. For example, consider the expression

\[ x = 3 \text{ and not } y = 2 \]

with

\( \sigma = \{ x = 3, y = 4 \} \).

\[ \begin{align*}
\sigma(x) &= 3 \\
\langle x, \sigma \rangle &\rightarrow 3 \\
\langle x = 3, \sigma \rangle &\rightarrow \text{tt}
\end{align*} \]

\[ \begin{align*}
\sigma(y) &= 4 \\
\langle y, \sigma \rangle &\rightarrow 4 \\
\langle y = 2, \sigma \rangle &\rightarrow \text{ff}
\end{align*} \]

\[ \begin{align*}
\langle \text{not } y = 2, \sigma \rangle &\rightarrow \text{tt}
\end{align*} \]

\[ \langle x = 3 \text{ and not } y = 2, \sigma \rangle \rightarrow \text{tt} \]
EQUIVALENCE OF EXPRESSIONS

Two expressions are equivalent if they evaluate to the same value in any given store:

\[ a_0 \sim a_1 \]
\[ \iff \left( \forall m \in \text{Int} : \forall \sigma \in \text{Stores} : \left\langle a_0, \sigma \right\rangle \to m \iff \left\langle a_1, \sigma \right\rangle \to m \right) ; \]

\[ b_0 \sim b_1 \]
\[ \iff \left( \forall t \in \text{Bool} : \forall \sigma \in \text{Stores} : \left\langle b_0, \sigma \right\rangle \to t \iff \left\langle b_1, \sigma \right\rangle \to t \right) . \]
Ever heard of short-circuit evaluation?

It is a mechanism, used by C, C++ and many other programming languages — but not by IMP — whereby the evaluation of boolean expressions is stopped as soon as the overall value of the expression is known.

If \( b_0 \) is true, \( b_0 \text{ or } b_1 \) is always true, independently from \( b_1 \).

Short-circuit evaluation means that, in this case, \( b_1 \) is not evaluated. The case for \( b_0 \text{ and } b_1 \) is dual: if \( b_0 \) is false, evaluating \( b_1 \) can be avoided.

**Exercise:** Define a variation of IMP that uses short-circuit evaluation of boolean expressions.

**Exercise:** Can something similar be done for arithmetic expressions? How?
The execution of a command may change the values of the store. Transition relations and transition systems are used to define the operational semantics of a program.

Example:

\[ \langle x := 10, \sigma \rangle \rightarrow \sigma' \]

where \( \sigma' \) is the same as the store \( \sigma \) except for the location \( x \) which has the value 10.
Execution of Commands: Notation

Let $\sigma$ be a store. Then the store $\sigma[m/x]$ is defined as follows, for each $y \in \text{Var}$:

$$
\sigma[m/x](y) \overset{\text{def}}{=} \begin{cases} 
m, & \text{if } y = x; \\
\sigma(y), & \text{if } y \neq x.
\end{cases}
$$

Then we can write

$$
\langle x := 10, \sigma \rangle \rightarrow \sigma' = \sigma[10/x].
$$

It is assumed that in the initial state of the store, all the variables have a value 0.

Suppose $\sigma$ is the initial store, then

$$
\sigma(x) = 0, \quad \sigma(y) = 0,
$$

$$
\sigma'(x) = 10, \quad \sigma'(y) = 0.
$$
EVALUATION OF COMMANDS: RULES I

Skip operation:

\[ \langle \text{skip}, \sigma \rangle \rightarrow \sigma. \]

Assignment:

\[ \langle a, \sigma \rangle \rightarrow m \]
\[ \langle x := a, \sigma \rangle \rightarrow \sigma[m/x]. \]

Command sequence:

\[ \langle c_0, \sigma \rangle \rightarrow \sigma'' \]
\[ \langle c_1, \sigma'' \rangle \rightarrow \sigma' \]
\[ \langle c_0; c_1, \sigma \rangle \rightarrow \sigma'. \]
EVALUATION OF COMMANDS: RULES II

Conditional:

\[
\begin{align*}
&\langle b, \sigma \rangle \to \text{tt} \quad \langle c_0, \sigma \rangle \to \sigma' \\
&\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \to \sigma'; \\
&\langle b, \sigma \rangle \to \text{ff} \quad \langle c_1, \sigma \rangle \to \sigma' \\
&\langle \text{if } b \text{ then } c_0 \text{ else } c_1, \sigma \rangle \to \sigma'.
\end{align*}
\]

While-loop:

\[
\begin{align*}
&\langle b, \sigma \rangle \to \text{ff} \\
&\langle \text{while } b \text{ do } c, \sigma \rangle \to \sigma'; \\
&\langle b, \sigma \rangle \to \text{tt} \quad \langle c, \sigma \rangle \to \sigma'' \quad \langle \text{while } b \text{ do } c, \sigma'' \rangle \to \sigma' \\
&\langle \text{while } b \text{ do } c, \sigma \rangle \to \sigma'.
\end{align*}
\]
For example, with $\sigma_0 = \{x = 0\}$, consider the command

\[
\textbf{while not} \ (x = 1) \ \textbf{do} \ (x := x + 1)
\]

\[
\sigma_0(x) = 0
\]

\[
\langle x, \sigma_0 \rangle \rightarrow 0 \quad \langle 1, \sigma_0 \rangle \rightarrow 1
\]

\[
\langle x = 1, \sigma_0 \rangle \rightarrow \text{ff}
\]

\[
\langle \text{not} (x = 1), \sigma_0 \rangle \rightarrow \text{tt}
\]

\[
\sigma_0(x) = 0
\]

\[
\langle x, \sigma_0 \rangle \rightarrow 0 \quad \langle 1, \sigma_0 \rangle \rightarrow 1
\]

\[
\langle x + 1, \sigma_0 \rangle \rightarrow 1
\]

\[
\langle x := x + 1, \sigma_0 \rangle \rightarrow \sigma_0[1/x] = \{x = 1\}
\]
Let $\sigma_1 = \{ x = 1 \}$. So we can continue the derivation:

$$\sigma_1(x) = 1$$

$$\langle x, \sigma_1 \rangle \rightarrow \sigma_1(x) = 1 \quad \langle 1, \sigma_1 \rangle \rightarrow 1$$

$$\langle x = 1, \sigma_1 \rangle \rightarrow \text{tt}$$

$$\langle \text{not } (x = 1), \sigma_1 \rangle \rightarrow \text{ff}$$

$$\langle \text{while not } (x = 1) \text{ do } (x := x + 1), \sigma_1, \rangle \rightarrow \sigma_1$$

$$\langle \text{not } (x = 1), \sigma_0 \rangle \rightarrow \text{tt} \quad \langle x := x + 1, \sigma_0 \rangle \rightarrow \sigma_1 \quad \langle w, \sigma_1, \rangle \rightarrow \sigma_1$$

$$\langle \text{while not } (x = 1) \text{ do } (x := x + 1), \sigma_0, \rangle \rightarrow \sigma_1$$
EQUIVALENCE OF COMMANDS

Two commands are equivalent if, when executed starting from the same store, they evaluate to the same store.

\[ c_0 \sim c_1 \iff (\forall \sigma, \sigma' \in \text{Stores} : \langle c_0, \sigma \rangle \to \sigma' \iff \langle c_1, \sigma \rangle \to \sigma'). \]

**Exercise:** Let

\[
\begin{align*}
    c_0 & = \textbf{while} \ true \ \textbf{do} \ x := 1, \\
    c_1 & = \textbf{while} \ true \ \textbf{do} \ x := 6.
\end{align*}
\]

Is \( c_0 \sim c_1 \)?