Programming Language Semantics and Compiler Design

(Sémantique des Langages de Programmation et Compilation)

Natural Operational Semantics of Language **While**

Yliès Falcone

ylies.falcone@univ-grenoble-alpes.fr — www.ylies.fr

Univ. Grenoble Alpes, and LIG-Inria team CORSE

Master of Sciences in Informatics at Grenoble (MoSIG)
Master 1 info

Univ. Grenoble Alpes - UFR IM^{2}AG
www.univ-grenoble-alpes.fr — im2ag.univ-grenoble-alpes.fr

Academic Year 2020 - 2021

About Operational Semantics

Semantics is

- concerned with the *meaning* of grammatically correct programs;
- defined on abstract syntax trees, obtained after type analysis.

With Operational Semantics the meaning of a construct tells how to execute it.

Semantics is described in terms of "*sequences of configurations*", which give the state-history of the machine.

Outline

Outline - Natural Operational Semantics of Language **While**

Meta-Variables

Meta-variables:

- x: variable
- S: statement
- a: arithmetic expression
- b: Boolean expression

Meta-variables can be primed or sub-scripted

Example 1 (Meta-Variables)

- variables: $x, x', x_1, x_2, \ldots$
- statements: $S, S_1, S', \ldots$
- arithmetic expressions: $a_1, a_2, \ldots$
- Boolean expressions: $b_1, b', b_2, \ldots$

Abstract Grammar of language **While**

Definition 1 (Abstract Grammar of language **While**)

$$
\begin{array}{rcl}
S & ::= & x := a \mid \text{skip} \\
 & & \mid S; S \\
 & & \mid \text{if } b \text{ then } S \text{ else } S \text{ fi} \\
 & & \mid \text{while } b \text{ do } S \text{ od}
\end{array}
$$

Remark This is an *inductive* definition:

- $x := a$ and skip are basis elements;
- $S; S$, if b then S else S fi, while b do S od are composition rules to define composite elements.

Syntactic Categories

- Numbers

$$n \in \textbf{Num} = \{0, \ldots, 9\}^+$$

- Variables

$$x \in \textbf{Var}$$

- Arithmetic expressions

$$\begin{array}{lll} a & \in & \textbf{Aexp} \\ a & ::= & n \mid x \mid a + a \mid a * a \mid a - a \end{array}$$

Num, **Var**, and **Aexp** are *syntactic categories*.

Remark Other operators for arithmetic expressions can be defined from the proposed ones. □

Syntactic categories (ctd)

- Boolean expressions

$$\begin{array}{lll} b & \in & \textbf{Bexp} \\ b & ::= & \text{true} \mid \text{false} \mid a = a \mid a \leq a \mid \neg b \mid b \wedge b \end{array}$$

- Statements

$$\begin{array}{lll} S & \in & \textbf{Stm} \\ S & ::= & x := a \mid \text{skip} \mid S; S \\ & & \mid \text{if } b \text{ then } S \text{ else } S \text{ fi} \\ & & \mid \text{while } b \text{ do } S \text{ od} \end{array}$$

Bexp and **Stm** are *syntactic categories*.

Concrete vs. abstract syntax

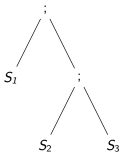
We focus on *abstract syntax* and abstract away concrete syntax.

- Term $S_1; S_2$ represents the tree, s.t.
  - the root is ;
  - left child is S_1 tree
  - right child is S_2 tree

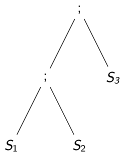
- Parenthesis shall be used to avoid ambiguities.

Example 2 (Abstract Syntax Tree)

- $S_1; (S_2; S_3)$



- $(S_1; S_2); S_3$



We will only use the linear notation.

Outline - Natural Operational Semantics of Language **While**

Semantic domains

- Integers: $\mathbb{Z}$
- Booleans: $\mathbb{B} = \{\mathbf{tt}, \mathbf{ff}\}$
- States:

$$\mathbf{State} = \mathbf{Var} \to \mathbb{Z}$$

Intuition: a state is a "memory".
In the following σ denotes a state in **State**.

Definition 2 (Substitution for a state)

Let $v \in \mathbb{Z}$. Then, $\sigma[y \mapsto v]$ denotes the state σ' such that:

$$\text{for all } x \in \mathbf{Var}, \sigma'(x) = \begin{cases} \sigma(x) & \text{if } x \neq y, \\ v & \text{otherwise.} \end{cases}$$

Example 3 (Substitution for a state)

For $\sigma = [x \mapsto 0, y \mapsto 1]$:

- $\sigma[x \mapsto 2] = [x \mapsto 2, y \mapsto 1]$,
- $\sigma(z) = \mathit{undef} = \sigma[x \mapsto 2](z)$.

Remark Substitution can be generalized to several pairs of variables/values. One should choose to do them in sequence or in parallel. □

Semantic functions

- Numerals: integers

$$\begin{array}{rcl} \mathcal{N} & : & \mathbf{Num} \rightarrow \mathbb{N} \\ \mathcal{N}(n_1 \cdots n_k) & = & \Sigma_{i=1}^{k} n_i \times 10^{k-i} \end{array}$$

- Arithmetic expressions: for each state, a value in $\mathbb{Z}$
  $\mathcal{A} : \mathbf{Aexp} \rightarrow (\mathbf{State} \rightarrow \mathbb{Z})$

$$\begin{array}{rcl} \mathcal{A}[n]\sigma & = & \mathcal{N}(n) \\ \mathcal{A}[x]\sigma & = & \sigma(x) \\ \mathcal{A}[a_1 + a_2]\sigma & = & \mathcal{A}[a_1]\sigma +_I \mathcal{A}[a_2]\sigma \\ \mathcal{A}[a_1 * a_2]\sigma & = & \mathcal{A}[a_1]\sigma *_I \mathcal{A}[a_2]\sigma \\ \mathcal{A}[a_1 - a_2]\sigma & = & \mathcal{A}[a_1]\sigma -_I \mathcal{A}[a_2]\sigma \end{array}$$

- *inductive /compositional* semantics: defined over the structure
- Caution: distinguish $*$ and $*_I$, $+$ and $+_I$, $-$ and $-_I$

- Boolean expressions: for each state, a value in $\mathbb{B}$

$$\mathcal{B} : \mathbf{Bexp} \rightarrow (\mathbf{State} \rightarrow \mathbb{B})$$

Remark The semantics of expressions can also be defined in an operational way (cf. tutorial). □

Semantic functions (ctd): some examples/exercises

Example 4 (Semantics of arithmetic expressions)

Let σ be $[x \mapsto 4, y \mapsto 3]$

$$\begin{aligned}
\mathcal{A}[2 * x + y]\sigma &= \mathcal{A}[2 * x]\sigma +_I \mathcal{A}[y]\sigma \\
&= (\mathcal{A}[2]\sigma *_I \mathcal{A}[x]\sigma) +_I \mathcal{A}[y]\sigma \\
&= (\mathcal{N}(2) *_I \sigma(x)) +_I \sigma(y) \\
&= (2 *_I 4) +_I 3 = 11
\end{aligned}$$

Definition of $\mathcal{A}$

$$\begin{aligned}
\mathcal{A}[n]\sigma &= \mathcal{N}(n) \\
\mathcal{A}[x]\sigma &= \sigma(x) \\
\mathcal{A}[a_1 @ a_2]\sigma &= \mathcal{A}[a_1]\sigma @_I \mathcal{A}[a_2]\sigma
\end{aligned}$$

$@ \in \{+_I, -_I, *_I\}$

Exercise 1 (Semantic function for digits in base 2)

- Define the syntactic category $\mathbf{Num}_2$ of numerals in base 2.
- Give them a compositional semantics

- Non-inductive definition with a regular expression: $\{0, 1\}^+$.
  Inductive definition with a grammar: $n ::= 0 \mid 1 \mid n0 \mid n1$.
- Compositional semantics: $\mathcal{N}_2 : \mathbf{Num}_2 \to \mathbb{N}$

$$\begin{array}{rcl|rcl}
\mathcal{N}_2(0) & = & 0 & \mathcal{N}_2(n0) & = & 2 *_I \mathcal{N}_2(n) \\
\mathcal{N}_2(1) & = & 1 & \mathcal{N}_2(n1) & = & 2 *_I \mathcal{N}_2(n) +_I 1
\end{array}$$

Semantic functions (ctd): some examples/exercises

Definition of Bexp

$$
\begin{array}{lll}
b & \in & \textbf{Bexp} \\
b & ::= & \text{true} \mid \text{false} \mid a = a \mid a \leq a \\
 & & \mid \neg b \mid b \wedge b
\end{array}
$$

Exercise 2 (Semantic function for Boolean expressions)

Define a declarative (and inductive) semantics for Boolean expressions.

$$\mathcal{B} : \textbf{Bexp} \rightarrow (\textbf{State} \rightarrow \mathbb{B})$$

$$
\begin{aligned}
\mathcal{B}[\text{true}]\sigma &= \textbf{tt} \\
\mathcal{B}[\text{false}]\sigma &= \textbf{ff} \\
\mathcal{B}[\neg b]\sigma &= \neg_{\mathbb{B}} \mathcal{B}[b]\sigma \\
\mathcal{B}[a_1 = a_2]\sigma &= \mathcal{A}[a_1]\sigma =_I \mathcal{A}[a_2]\sigma \\
\mathcal{B}[a_1 \leq a_2]\sigma &= \mathcal{A}[a_1]\sigma \leq_I \mathcal{A}[a_2]\sigma \\
\mathcal{B}[b_1 \wedge b_2]\sigma &= \mathcal{B}[b_1]\sigma \wedge_{\mathbb{B}} \mathcal{B}[b_2]\sigma
\end{aligned}
$$

Semantic functions (ctd): some examples/exercises

Exercise 3 (Negative integers)

We add $-a$ as a construct for arithmetical expressions.

- Extend the semantic function of arithmetical expressions
  Semantics of arithmetical expressions should remain compositional.

A compositional definition is a definition where a composite elements is defined in terms of the definitions of its components.

We have two possible solutions:

- $\mathcal{A}[-a]\sigma = 0 -_I \mathcal{A}[a]\sigma$ (preserves compositionality),
- $\mathcal{A}[-a]\sigma = \mathcal{A}[0 - a]\sigma$ (does not preserve compositionality).

Outline

Semantic functions

- Statements:

$$\mathcal{S} : \textbf{Stm} \rightarrow (\textbf{State} \xrightarrow{\textit{part.}} \textbf{State})$$

Function $\mathcal{S}$ gives the *meaning* of a statement S as a partial function from **State** to **State**.

Question: why is it a **partial** function?

Various semantic styles

- Axiomatic semantics allows to prove program properties (later in the course).
- Denotational semantics describes the effect of program execution (from a given state), *without telling how* the program is executed (later in the course).
- **Operational semantics** tells **how a program is executed**
  $\hookrightarrow$ It helps to write interpreters or code generators

Another important feature is *compositionality*: the semantics of a compound program is a function of the semantics of its components.

Operational Semantics

An operational semantics defines a transition system

Definition 3 (Transition System)

A transition system is given by $(\Gamma, T, \rightarrow)$, where:

- Γ is the set of *configurations*
- $T \subseteq \Gamma$ is the set of *final configurations*
- $\rightarrow \subseteq \Gamma \times \Gamma$ is the *transition relation*

Example 5 (Transition System)

The semantics of a DFA (over Σ) is a transition system:

- $\Gamma = Q \times \Sigma^*$
  - Q is the set of states of the DFA
  - Σ^* is the set of finite words over Σ

  DFA in (q, w) means: it is in state q and w is the remaining word to read
- $T = \{(q, \epsilon) \mid q \in Q\}$
- $\rightarrow (q, a \cdot w) = (q', w)$ s.t. q' is the state reached by firing a in state q

An execution of the DFA is a sequence of configurations.

Natural Operational Semantics (NOS)

- Defines the relationship between initial and final steps of an execution.
- This relationship is specified for each statement, w.r.t. a current **State**.

Transition system for Natural Operational Semantics

- Configurations: $(\mathbf{Stm} \times \mathbf{State}) \cup \mathbf{State}$.
- Final configurations (a subset of the set of configurations): **State**.
  (Configurations in $\mathbf{Stm} \times \mathbf{State}$ are called non-final.)
- Transition relation: $\rightarrow \subseteq (\mathbf{Stm} \times \mathbf{State}) \times \mathbf{State}$
  We note $(S, \sigma) \rightarrow \sigma'$, when the program moves from configuration (S, σ) to the terminal configuration σ'.
  - "The execution of S from σ *terminates* in state σ'"
  - Goal: to describe how the result of a program execution is obtained.

Example 6 (Elements from a transition system of NOS)

- Configuration: $(x := 10; y := x + 42, [x \mapsto 0, y \mapsto 1])$: "the program has $x := 10; y := x + 42$ to execute and its memory is s.t. x (resp. y) has value 0 (resp. 1).
- Final configuration: $[x \mapsto 10, y \mapsto 52]$: "the execution of the program has terminated and its memory is s.t. x (resp. y) has value 10 (resp. 52).
- Transition: $(x := 10; y := x + 42, [x \mapsto 0, y \mapsto 1]) \rightarrow [x \mapsto 10, y \mapsto 52]$.

Natural semantics: about rules

Semantics is defined by an inference system: axioms and rules.

Rules of the form:

$$\frac{(S_1, \sigma_1) \to \sigma_1' \quad (S_2, \sigma_2) \to \sigma_2' \quad \ldots \quad (S_n, \sigma_n) \to \sigma_n'}{(S, \sigma) \to \sigma'} \text{ if} \cdots$$

- $S_1, S_2, \ldots, S_n$ are immediate constituents of S, i.e., S is "built on" $S_1, \ldots, S_n$ or statements built from immediate constituents,
- $(S_1, \sigma_1) \to \sigma_1', (S_2, \sigma_2) \to \sigma_2', \ldots, (S_n, \sigma_n) \to \sigma_n'$ are called premises of the rule ; if $n = 0$, the rule is called axiom (schema) and the solid line is omitted,
- $(S, \sigma) \to \sigma'$ is the conclusion of the rule,
- a rule may also have a condition (if $\cdots$).

Remark The evolution between configurations is described in terms of "big steps" as there is always a terminal configuration on the "right-hand side" of $\to$. This is a distinguishing feature of **natural** operational semantics. □

Natural semantics: axioms and rules

Axioms

$$\frac{}{(x := a, \sigma) \to \sigma[x \mapsto \mathcal{A}[a]\sigma]}$$

$$\frac{}{(\mathsf{skip}, \sigma) \to \sigma}$$

Rule for Sequential Statements

$$\frac{(S_1, \sigma) \to \sigma' \quad (S_2, \sigma') \to \sigma''}{(S_1; S_2, \sigma) \to \sigma''}$$

Natural semantics: axioms and rules (ctd)

Rules for Conditional Statements

$$\frac{(S_1, \sigma) \rightarrow \sigma'}{(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}, \sigma) \rightarrow \sigma'} \ \textit{if}\, \mathcal{B}[b]\sigma = \mathbf{tt}$$

$$\frac{(S_2, \sigma) \rightarrow \sigma'}{(\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi}, \sigma) \rightarrow \sigma'} \ \textit{if}\, \mathcal{B}[b]\sigma = \mathbf{ff}$$

Rules for Iterative Statements (unbounded iteration)

$$\frac{(S, \sigma) \rightarrow \sigma' \quad (\text{while } b \text{ do } S \text{ od}, \sigma') \rightarrow \sigma''}{(\text{while } b \text{ do } S \text{ od}, \sigma) \rightarrow \sigma''} \ \textit{if}\, \mathcal{B}[b]\sigma = \mathbf{tt}$$

$$\frac{}{(\text{while } b \text{ do } S \text{ od}, \sigma) \rightarrow \sigma} \ \textit{if}\, \mathcal{B}[b]\sigma = \mathbf{ff}$$

Derivation tree

Describes an execution *from* a statement S and a state σ *to* a state σ'.

- Leaves correspond to (instantiation of) axioms.
- Internal nodes corresponds to (instantiation of) inference rules.
- the root is $(S, \sigma) \to \sigma'$ (it is common to have the root at the bottom rather than at the top when drawing a derivation tree).

Example 7 (Derivation Tree)

Consider $\sigma \in \textbf{State}$, the execution of $x := 1; y := 5$ on σ is described by the following derivation tree:

$$\frac{\overline{(x := 1, \sigma) \to \sigma[x \mapsto 1]} \quad \overline{(y := 5, \sigma[x \mapsto 1]) \to \sigma[x \mapsto 1][y \mapsto 5]}}{(x := 1; y := 5, \sigma) \to \sigma[x \mapsto 1, y \mapsto 5]}$$

Construction of derivation tree

Given,

- a statement (abstract tree) S,
- a state σ,

we want to find σ' (if it exists) such that $(S, \sigma) \rightarrow \sigma'$.

The method tries to construct a tree with root $(S, \sigma) \rightarrow \sigma'$ upwards. We start with an axiom or a rule such that the conclusion where the left-hand side "matches" the configuration (S, σ).

There are two cases :

- *if it is an axiom* and the condition of the axiom holds, then we can compute the final state and the construction of the derivation tree is completed on that branch,
- *if it is a rule*, then the next step is to repeat this step to try to construct a derivation tree for all the premises of the rule.

Construction of derivation tree: a first example

Sequence of assignments

Let

- $S = (z := x; x := y); y := z$
- $\sigma_0 = [x \mapsto 2, y \mapsto 4, z \mapsto 0]$

Applying axioms and rules we obtain:

$$\dfrac{\dfrac{\dfrac{}{(z := x, \sigma_0) \to \sigma_1} \quad \dfrac{}{(x := y, \sigma_1) \to \sigma_2}}{(z := x; x := y, \sigma_0) \to \sigma_2} \qquad \dfrac{}{(y := z, \sigma_2) \to \sigma_3}}{((z := x; x := y); y := z, \sigma_0) \to \sigma_3}$$

with,

- $\sigma_1 = [x \mapsto 2, y \mapsto 4, z \mapsto 2]$,
- $\sigma_2 = [x \mapsto 4, y \mapsto 4, z \mapsto 2]$,
- $\sigma_3 = [x \mapsto 4, y \mapsto 2, z \mapsto 2]$.

Construction of derivation tree: another example

Iterative statement

Consider

- S_0 : while $x > 1$ do $y := y * x; x := x - 1$ od
- S_1 : $y := y * x; x := x - 1$
- $\sigma_{31} = [x \mapsto 3, y \mapsto 1]$.
  We use the notation σ_{uv} to denote the state $[x \mapsto u, y \mapsto v]$.

We try to find $\sigma?$ such that $(S_0, \sigma_{31}) \to \sigma?$.

$$\frac{T_1 \quad T_2}{(S_0, \sigma_{31}) \to \sigma?}$$

Construction of T_1:

$$\frac{\overline{(y := y * x, \sigma_{31}) \to \sigma_{33}} \quad \overline{(x := x - 1, \sigma_{33}) \to \sigma_{23}}}{(S_1, \sigma_{31}) \to \sigma_{23}}$$

Construction of T_2:

$$\frac{T_3 \quad T_4}{(S_0, \sigma_{23}) \to \sigma?}$$

Construction of T_3:

$$\frac{\overline{(y := y * x, \sigma_{23}) \to \sigma_{26}} \quad \overline{(x := x - 1, \sigma_{26}) \to \sigma_{16}}}{(S_1, \sigma_{23}) \to \sigma_{16}}$$

Construction of T_4: $\overline{(\text{while } x > 1 \text{ do } y := y * x; x := x - 1 \text{ od}, \sigma_{16}) \to \sigma_{16}}$

Example cont.

The construction of derivation tree stops when we find σ_{16} because in this state, $\sigma_{16}(x) = 1$ and $\mathcal{B}[x > 1]_{\sigma_{16}} = \textbf{ff}$.

Finally, we find $\sigma? = \sigma_{16}$ and the derivation tree is:

$$\dfrac{T_1 \quad \dfrac{T_3 \quad (S_0, \sigma_{16}) \rightarrow \sigma_{16}}{(S_0, \sigma_{23}) \rightarrow \sigma_{16}}}{(S_0, \sigma_{31}) \rightarrow \sigma_{16}}$$

Exercise 4 (Derivation trees)

Find out the semantics of the following program (from an empty state) by computing the corresponding derivation trees:

1. $x := 2; \text{if } x > 0 \text{ then } x := x + 1 \text{ else } x := x - 1 \text{ fi};$
2. $x := 2; \text{while } x > 0 \text{ do } x := x - 1 \text{ od};$
3. $x := 2; \text{while } x > 0 \text{ do } x := x + 1 \text{ od}.$

Terminology

Consider a statement S and a state σ.

Definition 4 (Termination/Looping)

The execution of S on σ

- terminates iff there is a state σ' s.t. $(S, \sigma) \to \sigma'$;
- loops iff there is no state σ' s.t. $(S, \sigma) \to \sigma'$.

Statement S

- always terminates iff the execution of S terminates on any state σ;
- always loops iff the execution of S loops on any state σ.

Another iterative construct

Exercise 5 (Adding constructs to **While**)

We want to add two forms of iterations to language **While**:

- iteration of fixed length, without iteration variable, bounded iterations, with an iteration variable.

For this, we add the the two following construct to the syntax:

$$\begin{array}{rcl} S & ::= & \text{iterate } n \text{ times } S \\ & & \mid \text{for } x{:=}a \text{ to } a \text{ loop } S \end{array}$$

Give their corresponding semantic rules. The semantic rules should not refer to the other constructs of **While**.

Natural operational semantics is deterministic

Theorem 1

For all statements $S \in \mathbf{Stm}$, *for all states* σ, σ' *and* σ''*:*

1. *If* $(S, \sigma) \rightarrow \sigma'$ *and* $(S, \sigma) \rightarrow \sigma''$, *then* $\sigma' = \sigma''$.
2. *If* $(S, \sigma) \rightarrow \sigma'$, *then there does not exist any infinite derivation tree.*

Proof.

By induction on the structure of the derivation tree.
We will do it during the tutorial sessions. $\square$

Semantic function $\mathcal{S}_{\rm ns}$

Definition 5 (The semantic function $\mathcal{S}_{\rm ns}$)

$$\mathcal{S}_{\rm ns}[S]\sigma = \begin{cases} \sigma' & \text{if } (S, \sigma) \to \sigma', \\ \text{undef} & \text{otherwise,} \end{cases}$$

(because of looping executions, it is a partial function).

Remark Since natural operational semantics is deterministic, $\mathcal{S}_{\rm ns}[S]$ is indeed a function. □

Example 8 (Applying the semantic function)

- $\mathcal{S}_{\rm ns}[x := 2][x \mapsto 0] = [x \mapsto 2]$ because $\overline{(x := 2, [x \mapsto 0]) \to [x \mapsto 2]}$.
- $\mathcal{S}_{\rm ns}[\text{while true do skip od}]\sigma = \text{undef}$, for any $\sigma \in \textbf{State}$.

Outline - Natural Operational Semantics of Language **While**

Summary

Natural Operational Semantics of language **While**

Definition of the While programming language:

- Syntax (inductive definitions of the syntactic categories).
- (Declarative) Semantics of arithmetical and Boolean expressions.
- Semantics of statements (operational semantics defined with a transition system):
  - operational: defines the "how";
  - natural: big step semantics (configurations on the right-hand side are always terminal configurations).

  The transition system associated with a program is defined by:
  - configurations
  - final configurations
  - (set of) transitions (defined by rules)
- Termination and looping of programs.
- Determinism of the semantics.