

Operational Semantics

Class notes for a lecture given by Mooly Sagiv

Tel Aviv University 24/5/2007

By Roy Ganor and Uri Juhasz

Reference

Semantics with Applications, H. Nielson and F. Nielson, Chapter 2.

http://www.daimi.au.dk/~bra8130/Wiley_book/wiley.html

Introduction

Semantics is the study of meaning of languages.

Formal semantics for programming languages is the study of formalization of the practice of computer programming.

A computer language consists of a (formal) syntax – describing the actual structure of programs and the semantics – which describes the meaning of programs.

Many tools exist for programming languages (compilers, interpreters, verification tools, ...), the basis on which these tools can cooperate is the formal semantics of the language.

Formal Semantics

When designing a programming language, formal semantics gives an (hopefully) unambiguous definition of what a program written in the language should do.

This definition has several uses:

- People learning the language can understand the subtleties of its use
- The model over which the semantics is defined (the semantic domain) can indicate what the requirements are for implementing the language (as a compiler/interpreter/...)
- Global properties of any program written in the language, and any state occurring in such a program, can be understood from the formal semantics
- Implementers of tools for the language (parsers, compilers, interpreters, debuggers etc) have a formal reference for their tool and a formal definition of its correctness/completeness
- Programs written in the language can be verified formally against a formal specification (or at least a definition for their correctness exists)
- 2 different programs in the language can be proved formally as equivalent/non-equivalent
- From a computer readable version of the semantics, an interpreter can be automatically generated – full compiler generation is not (yet) feasible

Semantic Domain

Formal semantics gives rules for translation from one domain (usually the program's abstract syntax) to another formally defined domain, the domain includes the state space and the space of functions/relations over the state:

- Operational semantics describes the effect of each statement on the state – usually giving the post-state as a function of the pre-state.

- Axiomatic Semantics usually defines a translation into logic formulae in some well defined logic language – the formulae describe, for each statement, the relation between the pre-state and the post-state of the executing the statement. This is usually given as a predicate transformer: a formula is true at the post-state if some syntactic transformation of it is true at the pre-state – or the other way around. Logic models are sometimes given over which these formulae are interpreted – the effect of a program in these models is the interpretation of the formulae – usually a relation.
- Denotational semantics defines a translation into some (partial) function space usually defined in set/category theory. The meaning of a program is a (partial) function/relation in that space.

Usually semantics are given in a mathematical, machine independent way.

Most formal semantics are tailored for language in which the abstract syntax of a program is a tree – hence a well-formed program can be seen as a tree of statements, with simple statements (e.g. assignment) in leaves and compound statements (e.g. while loops) in nodes.

There are several flavours for formal semantics:

- Operational Semantics:
 - Gives the effect of each statement as an operation or set of operations on some abstract machine
 - The effect is given for simple statements (leaves). For compound statements (nodes) a rule is given to combine the effects of its underlying statements
 - Major types are Large step semantics (Natural Semantics) and small step semantics (Structural Semantics)
 - Most closely related to interpreters and abstract interpretation
 - Example:
 - Assignment: $\langle x := c, s \rangle \rightarrow [s[x \mapsto s(c)]]$
 Meaning – after the assignment $x:=c$ the value of the variable x would be c .
 - While: $\frac{\langle S, s \rangle \rightarrow S'' \quad \langle S'', \mathbf{while\ e\ do\ s} \rangle \rightarrow S'}{\langle S, \mathbf{while\ e\ do\ s} \rangle \rightarrow S'}$
- Denotational Semantics:
 - Gives the effect of each statement as an equation describing a relation between the input state and the output state – hence gives a translation of the program into some relation in a well-formalized mathematical domain (e.g. set theory)
 - The effect of a whole program is a solution of the set of semantic equations – which are usually complicated (e.g. for loops usually fix points are used)
 - Example:
 - Assignment: $S_{DS} \llbracket x := c \rrbracket (s) \triangleq s[x \mapsto \llbracket c \rrbracket_s]$

Meaning that the assignment $x:=c$ is translated into the function that takes a state s and returns s with the value of x replaced by the value of the expression c in the s .

- While:

$$S_{DS} \llbracket \mathbf{while} \ e \ \mathbf{do} \ S \rrbracket (s) \triangleq \text{lfp} \left(F(x) = \text{ite}(\llbracket a \rrbracket, x \circ S_{DS} \llbracket S \rrbracket, id) \right)$$

Where lfp is the least fix point of an operator (Here F is an operator over semantic functions – a function over relations over states) and ite is the if-then-else operator

- Axiomatic Semantics:

- Gives the effect of each statement as a pair of predicates – for the pre and post-state – defined over a logical language, along with deduction rules for compound statements
- A program satisfies a property at a given point iff this property can be proved from the axioms and deduction rules of the semantics
- Can be used to convert program correctness into a set of verification conditions – which can be proved with any automatic/manual theorem proving tool
- Example:

- Assignment: $\{P[x/c]\} x := c \{P\}$

(P is an arbitrary formula and $[a/b]$ denotes textual substitution of free occurrences of a with b) – the meaning is that we can prove that P holds after the assignment if $P[x/c]$ held before the assignment – for example we can prove:

$$\{y > 5\} x := y \{x > 5\}$$

- While:
$$\frac{\{\llbracket e \rrbracket \wedge I\} S \{I\}}{\{I\} \mathbf{while} \ e \ \mathbf{do} \ S \{I \wedge \neg \llbracket e \rrbracket\}}$$

Here I is the loop invariant.

This roughly means: if we can prove that the loop body(S) preserves the invariant assuming e holds at the pre-state, then we can prove that at the end of the loop the invariant holds and e doesn't.

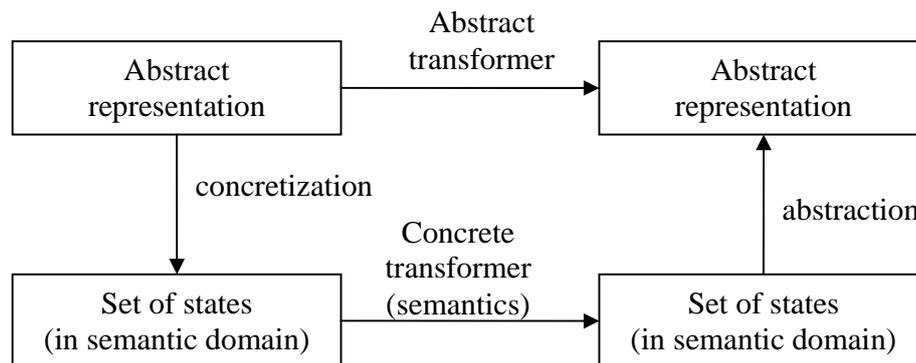
Usually, if several semantics are given for the same language, we would like to prove some form of an equivalence theorem: stating that all semantics describe the same language.

Abstract Interpretation and Semantics

Abstract interpretation is closely related to operational semantics as it is the execution of abstractions of the program – hence the exact semantics must be defined.

The concrete semantics (usually operational semantics) is defined over some concrete domain (the state space – e.g. for a program with variables this would be the set of partial functions from variable Ids to values)

In abstract interpretation we define some abstract domain, define a function mapping the concrete domain to the abstract domain (the abstraction function – which must satisfy certain requirements) and then translate the concrete semantics, using the abstraction function, to the abstract semantics – we get an abstract transformer for each concrete transformer in the concrete semantics.



The accuracy of this translation has direct effect on the accuracy of the analysis. For example, if our concrete domain is partial functions from variable ids to values, and the abstract domain is the signs of variables (plus the unknown sign), then:

The concrete semantics for addition:

$$\langle s, x := y + z \rangle \rightarrow s[x \mapsto s(y) + s(z)]$$

The abstract semantics would be:

$$\langle s^\#, x := y + z \rangle \rightarrow s^\#[x \mapsto s^\#(y) +^\# s^\#(z)]$$

Where:

$$N +^\# N = N$$

$$P +^\# P = P$$

For all other cases

$$x +^\# x = ? \text{ (unknown sign)}$$

Abstract interpretation then executes programs using only the abstract semantics (transformers) the translation ensures that any concrete execution is covered by some abstract execution, so if a property holds for all abstract executions it would hold for all concrete executions (after concretization of the property).

The While Programming Language

We define a simple programming language in order to demonstrate defining formal semantics to a language for which we know the intuitive meaning of statements.

The abstract syntax of the language is:

$$S ::=$$

- skip** |
- $x := a$ |
- $S_1; S_2$ |
- if** b **then** S_1 **else** S_2 |
- while** b **do** S

Where:

- a is an integer expression over variables
- b is a Boolean expression over variables
- x is a variable id

1. Semantic Domain for While:

The syntactic categories of while are:

- Variables: Var
- Aexp: Arithmetic Expression
- Bexp: Boolean Expression
- Statement: Statements (as defined in the abstract syntax)
-

The semantic categories of while are:

- Numbers: \mathbb{N} : are the natural numbers
- Booleans: \mathbb{T} : Boolean truth values $\{tt, ff\}$
- State: $\text{State} : \text{Variable} \rightarrow \text{Value}$
- State lookup : Sv the value of variable v in S (function application)
- State update : $S[v \mapsto c]$ (v is in Var and c in Aexp) gives:
 - $S[v \mapsto c]v=c$
 - $S[v \mapsto c]u=su$ (when $u \neq v$)

Semantics of Expressions

For most kinds of program language semantics, the semantics of side-effect-free expressions is independent of the semantics of statements.

Expressions appear in specific states in the language – for example in **while** we have integer expressions in assignment and Boolean expressions in **if** and **while** statements.

Expressions are interpreted over states and not over pairs of states.

The symbol used for interpretation of arithmetic expressions is $A[[x]]s$ where x is a term in the abstract syntax for arithmetic expressions and s is a state.

Semantics of integer expressions for **While**:

- $A[[n]]s = n$
- $A[[x]]s = sx$
- $A[[-e]]s = -A[[e]]s$
- $A[[e_1 + e_2]]s = A[[e_1]]s + A[[e_2]]s$
- $A[[e_1 * e_2]]s = A[[e_1]]s \times A[[e_2]]s$

Semantics of Boolean expressions for **While**:

- $B[[true]]s = tt$
- $B[[false]]s = ff$
- $B[[x]]s = sx$
- $B[[e_1 = e_2]]s = \text{if } A[[e_1]]s = A[[e_2]]s \text{ then } tt \text{ else } ff$
- $B[[e_1 \wedge e_2]]s = \text{if } B[[e_1]]s = tt \text{ and } B[[e_2]]s = tt \text{ then } tt \text{ else } ff$

As can be seen, the semantics for all expressions is compositional (in the abstract syntax tree for an expression, the meaning of each expression depends solely on the meaning of its direct descendants).

This property allows proving properties inductively on expressions – for example, if we want to show that an arithmetic expression with only even numbers and no variables evaluates to an even number, we would show this for leaves (n, x), and for nodes ($-e, e+e, e*e$), assuming it for direct descendants.

Natural Operational Semantics

Natural operational semantics (large step operational semantics) gives, for each program, the effect of the program – usually the poststate as a function of the prestate and the effects of other commands.

Notations:

S : a state

$\langle S, s \rangle \rightarrow s'$: The program S , when run on input s , terminates in the state s'

The second statement is partial – the program S may also terminate in other states than s' or not terminate at all.

The exact meaning of the program S is the set of pairs of states s.t. $\{(s, s') : \langle S, s \rangle \rightarrow s'\}$.

The semantic rules let us generate the set of true statements $\langle S, s \rangle \rightarrow s'$, where all other such statements are false by definition.

The abstract syntax of the language is given as a context free grammar – hence all programs can be seen as trees (a derivation tree of the abstract syntax) and the semantics can be given in a compositional way.

Compositional semantics are very useful because we can give inductive proofs for many program/language properties.

Each rule is given as (here a rule for “if” when the condition holds):

$$\left[\text{if}_{NS}^{tt} \right] \frac{\langle S_1, s \rangle \rightarrow s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'} B[[b]]s = tt$$

Where:

$\left[\text{if}_{NS}^{tt} \right]$	is the rule name (for later reference)
$\langle S_1, s \rangle \rightarrow s'$	are the premises (may be several)
$\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'$	is the consequence
$B[[b]]s = tt$	is a condition

If there are no premises and no condition we only write the consequence (an axiom scheme)

The meaning of this rule is that if the premises holds and the condition holds (the condition usually refers only the prestate) then the consequence holds as well.

The rules are compositional – hence statements that appear in the premises must be substatements (syntactically) of the statement in the consequence (here S_1 is a substatement of **if** b **then** S_1 **else** S_2).

There would usually be at least one rule per each substatement of the premises (otherwise some subtrees may be redundant)

The operational semantics of **while**:

Simple statements:

- **Skip:** $[\text{skip}_{\text{NS}}] \quad \langle \text{skip}, s \rangle \rightarrow s$
- **Assignment:** $[\text{ass}_{\text{NS}}] \quad \langle x := a, s \rangle \rightarrow s[x \mapsto A[a]s]$

Compound statements:

- **Sequential composition :**

$$[\text{comp}_{\text{NS}}] \quad \frac{\langle S_1, s \rangle \rightarrow s'' \quad \langle S_2, s'' \rangle \rightarrow s'}{\langle S_1; S_2, s \rangle \rightarrow s'}$$

- **Conditional:**

$$[\text{if}_{\text{NS}}^{\text{tt}}] \quad \frac{\langle S_1, s \rangle \rightarrow s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'} B[[b]]_{s=\text{tt}}$$

$$[\text{if}_{\text{NS}}^{\text{ff}}] \quad \frac{\langle S_2, s \rangle \rightarrow s'}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'} B[[b]]_{s=\text{ff}}$$

- **Loops:**

$$[\text{while}_{\text{NS}}^{\text{tt}}] \quad \frac{\langle S, s \rangle \rightarrow s'' \quad \langle \text{while } b \text{ do } S, s'' \rangle \rightarrow s'}{\langle \text{while } b \text{ do } S, s \rangle \rightarrow s'} B[[b]]_{s=\text{tt}}$$

$$[\text{while}_{\text{NS}}^{\text{ff}}] \quad \frac{}{\langle \text{while } b \text{ do } S, s \rangle \rightarrow s} B[[b]]_{s=\text{ff}}$$

The “proof “of a statement $\langle S, s \rangle \rightarrow s'$ is a derivation of $\langle S, s \rangle \rightarrow s'$ from the rules – this forms a tree of rule instances, where at each node there is a rule, at each leaf there is an axiom (a premise free rule) and for each node, there is exactly one descendant per premise of the rule, with its consequence equal to that premise – the condition must hold for all premises. The root of the tree is the statement to be proved. (These trees must be finite)

In order to prove a statement $\langle S, s \rangle \rightarrow s'$, we have to find a derivation tree for it:

Start with the root (the statement)

At each stage, for each leaf n in the tree, for each premise p of the rule in the leaf for which there is no corresponding descendant, find a rule where p is the consequence and add it as a descendant to n .

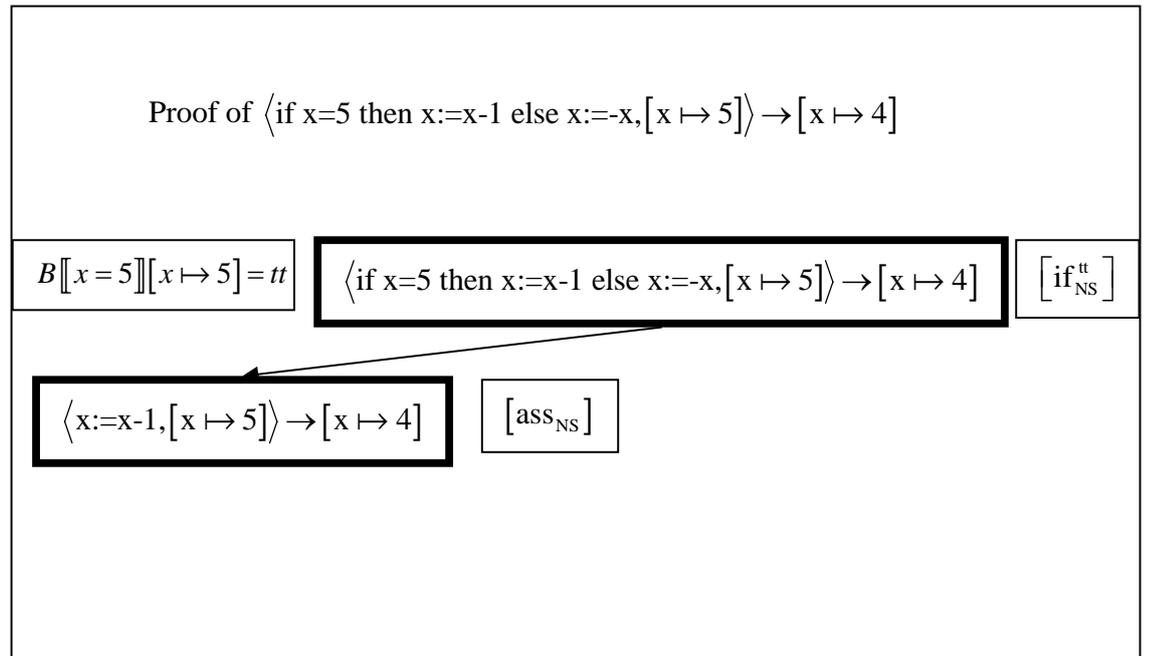
Terminate when there are no more premises to satisfy (all leaves are simple statements).

When choosing a rule to apply, we may have several options (several rules with the same consequence) – in order to prove our statement we have to choose the correct option, in order to prove it wrong (s' cannot be reached from s through S) we have to recursively check each option.

When using a rule, we may have several possible instances of the same rule (for example: in the rule comp_{NS} , s'' does not appear in the consequence therefore we can use any s''). To prove our statement we have to choose the

correct one (for example, for each S_1 and each s there is a unique s'' s.t. $\langle S_1, s \rangle \rightarrow s''$, therefore there is exactly one instance of the rule that can be applied).

In **while** the choice of rule is deterministic (always at most one option), and the choice of premises for true statements is unique (because the semantics is deterministic).



2. Proving program properties - equivalence:

Two programs S_1 and S_2 are equivalent iff for all s, s' : $\langle S_1, s \rangle \rightarrow s'$ iff $\langle S_2, s \rangle \rightarrow s'$.

We will show a proof, using the natural semantics, of the equivalence of the following two program schemes (parametric in S and b):

S_1 : **if** b **then** S ; **while** b **do** S **else skip**

S_2 : **while** b **do** S

Assuming $\langle S_1, s \rangle \rightarrow s'$ we'll show $\langle S_2, s \rangle \rightarrow s'$

We have two cases:

If $B[[b]]_s = ff$ then:

Using $[if_{NS}^{ff}]$ (the only possible derivation) we get $\langle S_1, s \rangle \rightarrow s'$ iff

$\langle \text{skip}, s \rangle \rightarrow s'$ - hence $s=s'$

For S_2 , using $[while_{NS}^{ff}]$ we get $s=s'$

If $B[[b]]_s = tt$ then:

Using $[if_{NS}^t]$ (the only possible derivation) we get $\langle S_1, s \rangle \rightarrow s'$ iff $\langle S; \mathbf{while\ b\ do\ S}, s \rangle \rightarrow s'$.

Now using $[comp_{NS}]$ we get $\langle S_1, s \rangle \rightarrow s'$ iff there exists an s'' s.t. $\langle S, s \rangle \rightarrow s''$ and $\langle \mathbf{while\ b\ do\ S}, s'' \rangle \rightarrow s'$.

Starting from $\langle S_1, s \rangle \rightarrow s'$, using $[while_{NS}^t]$ we get $\langle S_2, s \rangle \rightarrow s'$ iff there exists s'' s.t. $\langle S, s \rangle \rightarrow s''$ and $\langle \mathbf{while\ b\ do\ S}, s'' \rangle \rightarrow s'$.

Hence $\langle S_1, s \rangle \rightarrow s'$ iff $\langle S_2, s \rangle \rightarrow s'$

The other direction is shown similarly

Proving language properties – determinism:

We will show a proof for determinism of the semantics we gave for **while**.

Determinism is a language property – every program written in the language is deterministic.

Formally, determinism means (in our notation):

For a program S , S is deterministic iff for every s, s', s'' : $\langle S, s \rangle \rightarrow s'$ and $\langle S, s \rangle \rightarrow s''$ imply $s' = s''$.

We can prove this property by induction on the derivation tree:

We have to show it for the leaves (axioms) and for nodes assuming their subnodes are deterministic.

The proof for while:

Simple Statements:

- **Skip:**

$\langle \mathbf{skip}, s \rangle \rightarrow s'$ and $\langle \mathbf{skip}, s \rangle \rightarrow s''$ iff $s = s'$ and $s = s''$ iff $s' = s''$

- **Assignment:**

$\langle x := a, s \rangle \rightarrow s'$ iff $s' = s[x \mapsto A[a]s]$

$\langle x := a, s \rangle \rightarrow s''$ iff $s'' = s[x \mapsto A[a]s]$

Hence $\langle x := a, s \rangle \rightarrow s'$ and $\langle x := a, s \rangle \rightarrow s''$ imply $s' = s[x \mapsto A[a]s] = s''$

Compound statements:

- **Sequential composition :**

In the derivation $\frac{\langle S_1, s \rangle \rightarrow s''' \quad \langle S_2, s''' \rangle \rightarrow s'}{\langle S_1; S_2, s \rangle \rightarrow s'}$ s''' is unique (by induction for

S_1), so if $\langle S_1; S_2, s \rangle \rightarrow s'$ and $\langle S_1; S_2, s \rangle \rightarrow s''$ we have to have the exact same s''' in both derivations and because $\langle S_2, s''' \rangle \rightarrow s'$ and $\langle S_2, s''' \rangle \rightarrow s''$ imply $s' = s''$ (induction hypothesis for S_2) we get $\langle S_1; S_2, s \rangle \rightarrow s'$ and $\langle S_1; S_2, s \rangle \rightarrow s''$ imply $s' = s''$

Other compound statements are proved similarly.

Determinism will hold for ANY program in the language – which is a strong theorem.

The Semantic Function:

The semantic function is the translation of a program to a partial function (or relation for non-deterministic languages) over states

$$S_{ns} : \text{Statement} \rightarrow (\text{State} \leftrightarrow \text{State})$$

The semantic function describes mathematically what the program does – hence it can be used to compare different semantics.

Examples:

$$S_{ns}[\text{skip}] = \text{id}$$

$$S_{ns}[y:=1; \text{while } x>0 \text{ do } (y:=y*x; x:=x-1)] = s \mapsto \begin{cases} s[x \mapsto 0, y \mapsto (sx)!] & \text{if } x>0 \\ s[y \mapsto 1] & \text{if } x \leq 0 \end{cases}$$

Example for incorrect rules:

Many programming languages (e.g. PASCAL) have a **repeat** construct in addition to the **while** construct.

If we try to formalize **repeat** we might end up with:

$$\left[\text{repeat}_{NS}^{\text{ff}} \right] \frac{\langle S, s \rangle \rightarrow s'' \quad \langle \text{repeat } S \text{ until } b, s'' \rangle \rightarrow s'}{\langle \text{repeat } S \text{ until } b, s \rangle \rightarrow s'} B[[b]]_{s=\text{ff}}$$

$$\left[\text{repeat}_{NS}^{\text{tt}} \right] \frac{\langle S, s \rangle \rightarrow s'}{\langle \text{repeat } S \text{ until } b, s \rangle \rightarrow s'} B[[b]]_{s=\text{tt}}$$

However these rules do not give the intended meaning as the condition b is evaluated **before** the execution of S rather than after it.

So, for example, the statement:

$$\text{repeat } x:=x+1 \text{ until } x \leq 0$$

When executed with the initial state $x=0$ will give the poststate $x=1$ while the actual program will not terminate.

The correct rules would be:

$$\left[\text{repeat}_{NS}^{\text{ff}} \right] \frac{\langle S, s \rangle \rightarrow s'' \quad \langle \text{repeat } S \text{ until } b, s'' \rangle \rightarrow s'}{\langle \text{repeat } S \text{ until } b, s \rangle \rightarrow s'} B[[b]]_{s''=\text{ff}}$$

$$\left[\text{repeat}_{NS}^{\text{tt}} \right] \frac{\langle S, s \rangle \rightarrow s'}{\langle \text{repeat } S \text{ until } b, s \rangle \rightarrow s'} B[[b]]_{s'=\text{tt}}$$

Structural Operational Semantics

Introduction

As we will see shortly, natural semantics is sometimes not enough for comprehensive analysis, Structural Operational Semantics (SOS) is used for describing a more *detailed* cases like semantics for pointers, multi-threading and more.

The main difference between natural semantics and SOS is that the latter emphasize the individual steps, meaning we have the ability to relate any portion of a statement.

In SOS we write the transition relation as $\langle S, s \rangle \Rightarrow \gamma$, this should be considered as “the *first* step of executing the statement S on state s leads to γ ”.

Therefore we can write two possibilities for γ :

1. $\gamma = \langle S', s' \rangle$: the execution of S from state s is *not* completed, and the remaining computation is expressed by the intermediate configuration $\langle S', s' \rangle$ (which needs to be performed subsequently).
2. $\gamma = s'$: the execution of S from state s has terminated with a final state s'

In case the result of $\langle S, s \rangle$ is not available we say that γ is a *stuck* configuration and there is no subsequent transitions.

Although SOS emphasizes to the intermediate execution, the *meaning* of a program P on an input state s is the set of *final* states (also stuck configuration) that can be executed in arbitrary finite steps.

Example Semantic

Going back to the “While” programming language example, the first two axioms $[ass_{sos}]$ and $[skip_{sos}]$ have not changed at all because the assignment and `skip` statements are fully executed in one step.

$$\begin{array}{l} [ass_{sos}] \quad \langle x := a, s \rangle \Rightarrow s[x \mapsto A[a]] \\ [skip_{sos}] \quad \langle \text{skip}, s \rangle \Rightarrow s \end{array}$$

The two rules $[comp_{sos}^1]$ and $[comp_{sos}^2]$ for a statement $S_1; S_2$ express that the execution starts with the first step of S_1 from s . Then there are two possible outcomes derived from the two possibilities for transition relation:

1. $[comp_{sos}^1]$ - The execution of S_1 has not been completed we have to complete it before embarking on the execution of S_2 . In this case the first step of $\langle S, s \rangle$ is an intermediate configuration $\langle S_1', s' \rangle$ then the next configuration is $\langle S_1'; S_2, s' \rangle$.

$$[comp_{sos}^1] \quad \frac{\langle S_1, s \rangle \Rightarrow \langle S_1', s' \rangle}{\langle S_1; S_2, s \rangle \Rightarrow \langle S_1'; S_2, s' \rangle}$$

2. $[comp_{sos}^2]$ - The execution of S_1 has been completed we can start on the execution of S_2 . In this case the result of execution S_1 from s is a final state s' then the next configuration is $\langle S_2, s' \rangle$

$$[comp_{sos}^2] \quad \frac{\langle S_1, s \rangle \Rightarrow s'}{\langle S_1; S_2, s \rangle \Rightarrow \langle S_2, s' \rangle}$$

The first step of the condition statement starts with testing b , then branching according to the outcome:

$$[if_{sos}^{tt}] \quad \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \Rightarrow \langle S_1, s \rangle \quad \text{if } B[b]s = tt$$

$$[if_{sos}^{ff}] \quad \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \Rightarrow \langle S_2, s \rangle \quad \text{if } B[b]s = ff$$

The first step of the while statement is to “unfold” it one level, that is to rewrite it as a condition. In the next execution step a test is performed and the execution resumes. Note that the $[while_{sos}]$ rule breaks the compositional structure of the semantic (the reason is that the rule defines the semantic in terms of itself), hence structural induction doesn't work here fine.

$$[while_{sos}] \quad \langle \text{while } b \text{ do } S, s \rangle \Rightarrow \langle \text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else } skip, s \rangle$$

Derivation Sequence

A derivation sequence of a statement S starting in state s is either:

1. a finite sequence: $\gamma_0, \gamma_1, \gamma_2, \dots, \gamma_k$ of configurations satisfying:

- $\gamma_0 = \langle S, s \rangle$
- $\gamma_i \Rightarrow \gamma_{i+1}$ for $0 \leq i < k, k \geq 0$
- γ_k is either a terminal configuration or stuck configuration

2. an infinite sequence $\gamma_0, \gamma_1, \gamma_2, \dots$ of configurations satisfying:

- $\gamma_0 = \langle S, s \rangle$
- $\gamma_i \Rightarrow \gamma_{i+1}$ for $0 \leq i$

More notations:

1. For a terminal or a stuck configuration γ_i we write:

$\gamma_0 \Rightarrow^i \gamma_i$ - indicates that there are i steps in the execution from γ_0 to γ_i

$\gamma_0 \Rightarrow^* \gamma_i$ - indicates that there is a finite number of steps from γ_0 to γ_i

2. we construct a derivation tree for each step in the sequence (see example below)

Examples:

1. $(z := x; x := y); y := z$ assuming $s_0 \ x = 5$ and $s_0 \ y = 7$

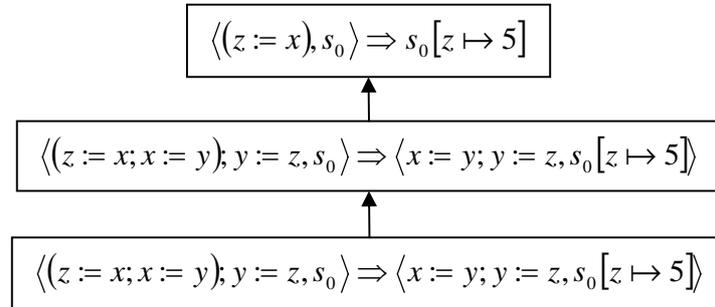
The derivation sequence can be constructed as follows:

$$\begin{aligned} & \langle (z := x; x := y); y := z, s_0 \rangle \\ & \Rightarrow \langle x := y; y := z, s_0 [z \mapsto 5] \rangle \\ & \Rightarrow \langle y := z, (s_0 [z \mapsto 5]) [x \mapsto 7] \rangle \\ & \Rightarrow ((s_0 [z \mapsto 5]) [x \mapsto 7]) [y \mapsto 5] \end{aligned}$$

We can, for example construct a derivation tree for the *first step*, which is

$$\langle\langle z := x; x := y \rangle; y := z, s_0\rangle \Rightarrow \langle x := y; y := z, s_0[z \mapsto 5]\rangle$$

And the derivation tree:

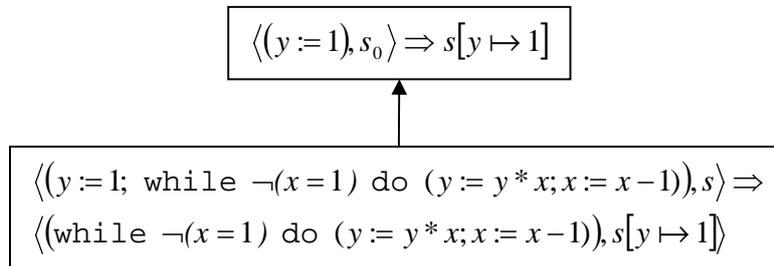


2. The first step of the factorial program, assuming $s \ x = 3$:

The first derivation step is:

$$\begin{aligned} &\langle(y := 1; \text{while } \neg(x = 1) \text{ do } (y := y * x; x := x - 1)), s\rangle \Rightarrow \\ &\langle(\text{while } \neg(x = 1) \text{ do } (y := y * x; x := x - 1)), s[y \mapsto 1]\rangle \end{aligned}$$

We used the axiom $[ass_{sos}]$ and rule $[comp_{sos}^2]$ as presented in the next derivation tree:



* Full derivation sequence along with the derivation trees are presented in the book pages 34-35.

Starting and Terminating a Program:

Given a statement S and a state s :

1. It is always possible to find at least one derivation sequence that starts. The reason is that we apply axioms and rules forever or until a terminal or stuck.
2. The execution terminates or loops:
 - a. terminates \Leftrightarrow there is a finite derivation sequence starting with $\langle S, s \rangle$
 - b. loops \Leftrightarrow there is an infinite derivation sequence starting with $\langle S, s \rangle$
3. The execution terminates successfully – if $\langle S, s \rangle \Rightarrow^* s'$ for some state s' , where s' is a final state and *not* a stuck configuration.
4. The execution always terminates – if it terminates on all states.

5. The execution always loops – if it loops on all states.

Properties of SOS:

1. Two statements S_1 and S_2 are semantically equivalent if for all states s both of the following are true:
 - a. $\langle S_1, s \rangle \Rightarrow^* \gamma \Leftrightarrow \langle S_2, s \rangle \Rightarrow^* \gamma$ (γ is either stuck or terminal, the length of the two sequences may be different)
 - b. There is infinite derivation sequence starting in $\langle S_1, s \rangle \Leftrightarrow$
There is infinite derivation sequence starting in $\langle S_2, s \rangle$
2. Deterministic - if for all choices of S, s, γ and γ' the following is true
 $\langle S, s \rangle \Rightarrow \gamma$ and $\langle S, s \rangle \Rightarrow \gamma'$ imply $\gamma = \gamma'$

Lemma:

$\langle S_1; S_2, s \rangle \Rightarrow^k s'' \rightarrow$ there exists s' and $k_1, k_2 \in \mathbb{N}$ s.t.
 $\langle S_1, s \rangle \Rightarrow^{k_1} s'$ and $\langle S_2, s' \rangle \Rightarrow^{k_2} s''$ where $k = k_1 + k_2$.

Proof:

For SOS it is often useful to conduct a proof by induction on the length of the derivation sequences, the induction has two steps:

1. The basis: Prove that the property holds for all derivation sequences of length 0.
2. The inductive step: Assume that the property holds for all derivation sequences of length at most k and show that it holds for a derivation sequences of length $k+1$, inspecting one of the following:
 - a. the structure of the syntactic element
 - b. the derivation tree validating the first transition of the derivation sequence.

So our induction will be on k - the length of the derivation sequence $\langle S_1; S_2, s \rangle \Rightarrow^k s''$.

1. For basic step $k=0$ the result holds vacuously.
2. For the induction step we assume that the lemma holds for $k \leq k_0$ and we shall prove it for $k_0 + 1$,

$\langle S_1; S_2, s \rangle \Rightarrow^{k_0+1} s''$, can be rewritten as $\langle S_1; S_2, s \rangle \Rightarrow \gamma \Rightarrow^{k_0} s''$ for some configuration γ .

Regarding $\langle S_1; S_2, s \rangle \Rightarrow \gamma$ there are two rules that can be used: $[comp_{sos}^1]$ and $[comp_{sos}^2]$.

- a) if we used $[comp_{sos}^1]$ - $\langle S_1; S_2, s \rangle \Rightarrow \langle S_1'; S_2, s' \rangle$ and taking the second part of the initial sequence we have $\langle S_1'; S_2, s' \rangle \Rightarrow^{k_0} s''$ and the induction hypothesis can be applied here (since this is shorter than the one we started with), hence there exist s' and $k_1, k_2 \in \mathbb{N}$ s.t $\langle S_1', s' \rangle \Rightarrow^{k_1} s_0$ and $\langle S_2, s_0 \rangle \Rightarrow^{k_2} s''$ where $k_1 + k_2 = k_0$. so we conclude that $\langle S_1, s \rangle \Rightarrow^{k_1+1} s_0$ and $\langle S_2, s_0 \rangle \Rightarrow^{k_2} s''$ where $(k_1 + 1) + k_2 = k_0 + 1$.

- b) if we used $[comp_{sos}^2]$ - then we have $\langle S_1, s \rangle \Rightarrow s'$ and (from that γ is $\langle S_2, s' \rangle$) we get $\langle S_2, s' \rangle \Rightarrow^{k_0} s''$ so we can choose $k_1=1$ and $k_2=k_0$ for our proof.

The Semantic Function S_{sos}

The meaning of a statement S is a partial function (not every element of the domain has to be associated with an element of the co-domain) from state to state:

$$S_{sos} : Stm \rightarrow (State \rightarrow State)$$

Such that:

$$S_{sos}[S]_s = \begin{cases} s' & \text{if } \langle S, s \rangle \Rightarrow^* s' \\ undef & \text{otherwise} \end{cases}$$

An Equivalent Result of NS and SOS

For every statement S of **while** we have $S_{ns}[S] = S_{sos}[S]$.

1. This theorem derives that if the execution of S from state terminates in one semantics then it also terminates in the other and the resulting states will be equal (the same can be derived for loop)
2. the full proof of this theorem is presented in book pages 50-54. the summary of the proof is given here:
 - a) Prove by induction on the shape of derivation trees that for each derivation tree in the NS there is a corresponding finite derivation sequence in the SOS.
 - b) Prove by induction on the length of derivation sequences that for each finite derivation sequence in the SOS semantic there is a corresponding derivation tree in the NS.

Extensions to While

We present here five different extensions to **while** programming language, to illustrate the power and weakness of the two approaches to operational semantics.

Every time we add a new component to the basic language we show how to modify the semantic. We could also add them one by one.

The extensions are:

1. abort statement
2. Non determinism
3. Parallelism
4. Local Variables and Procedures

abort Statement

Syntax: $S ::= x := a \mid skip \mid S1 ; S2 \mid \text{if } b \text{ then } S1 \text{ else } S2 \mid \text{while } b \text{ do } S \mid \text{abort}$

Meaning: abort stops the execution of the complete program.

abort a program is modeled by ensuring that the configuration of the form $\langle \text{abort}, s \rangle$ is stuck. Therefore the extended language is still defined in the original NS and SOS

semantic axioms and rules. Not setting a new rule to the new statement results in a stuck configuration when calling it from a program.

Semantically Equivalent and Different to `skip` and “while true do skip”

From the SOS point of view the three statements are different.

1. `skip` is *not semantically equivalent* to `abort` following the first condition of equivalency - since $\langle \text{skip}, s \rangle \Rightarrow s'$ but there is no state s' such that $\langle \text{abort}, s \rangle \Rightarrow s'$
2. “while true do skip” is *not semantically equivalent* to `abort` following the second condition of equivalency - there is infinite derivation sequence starting $\langle \text{while true do skip}, s \rangle$ but there is not starting $\langle \text{abort}, s \rangle$.

From the NS point of view things are bit different:

1. `skip` is (still) *not semantically equivalent* to `abort` following the condition of equivalency - since $\langle \text{skip}, s \rangle \Rightarrow s$ but there is no state s' such that $\langle \text{abort}, s \rangle \Rightarrow s'$
2. `(while true do skip)` is *semantically equivalent* to `abort` since the condition is true (holds vacuously – there is no such state), we are always concerned with executions that terminates properly.

Conclusions:

1. The natural semantics cannot distinguish between *looping* and *abnormal termination* (unless the states are modified)
2. In the structural operational semantics looping reflected by infinite derivations and abnormal termination is reflected by stuck configuration

Non-Determinism

Syntax: $S ::= x := a \mid \text{skip} \mid S_1 ; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S \mid S_1 \text{ or } S_2$

Meaning: we can non-deterministically choose to execute either one statement or another. Another example of non-determinism (in other languages) is allocating new memory or using random data.

Since we have two possible final states (depends who is executed), we will have two rules in our operational semantics (and subsequently – we will have two trees).

From the NS point of view we extend with two rules

$$[or_{NS}^1] \quad \frac{\langle S_1, s \rangle \rightarrow s'}{\langle S_1 \text{ or } S_2 \rangle \rightarrow s'}$$

$$[or_{NS}^2] \quad \frac{\langle S_2, s \rangle \rightarrow s'}{\langle S_1 \text{ or } S_2 \rangle \rightarrow s'}$$

From the SOS point of view we extend with two rules

$$[or_{SOS}^1] \quad \langle S_1 \text{ or } S_2 \rangle \Rightarrow \langle S_1, s \rangle$$

$$[or_{SOS}^2] \quad \langle S_1 \text{ or } S_2 \rangle \Rightarrow \langle S_2, s \rangle$$

Important note: a non-deterministic statement may or may not be equivalent under NS and SOS:

- (while true do skip) or $x := 1$
Under NS we will have only one derivation tree, rather than two under SOS (one infinite – representing the while loop and one finite representing the assignment).
- $x := 1$ or $(x := 2 ; x := x+2)$ – equivalent under the two semantics.

Conclusions:

1. In the natural semantics non-determinism will suppress looping if possible (mnemonic)
2. In the structural operational semantics non-determinism does not suppress termination configuration
3. Comparing the NS and SOS we see that the latter can choose the “wrong” branch of the or statement whereas the first always chooses the right branch.

Parallelism

Syntax: $S ::= x := a \mid \text{skip} \mid S1 ; S2 \mid \text{if } b \text{ then } S1 \text{ else } S2 \mid \text{while } b \text{ do } S \mid S1 \text{ par } S2$

Meaning: for a statement $S1 \text{ par } S2$ – both statements have to be executed but the execution can be interleaved (like multi-processing is) the order is not set by the order of the statement.

Example: $x := 1 \text{ par } (x := 2 ; x := x+2)$, can be executed as one of the following:

1. $x := 1 \rightarrow x := 2 \rightarrow x := x+2$
2. $x := 2 \rightarrow x := x+2 \rightarrow x := 1$
3. $x := 2 \rightarrow x := 1 \rightarrow x := x+2$

From the SOS point of view we write four extra rules:

$$[par_{sos}^1] \quad \frac{\langle S_1, s \rangle \Rightarrow \langle S_1', s' \rangle}{\langle S_1 \text{ par } S_2, s \rangle \Rightarrow \langle S_1' \text{ par } S_2, s' \rangle}$$

$$[par_{sos}^2] \quad \frac{\langle S_1, s \rangle \Rightarrow s'}{\langle S_1 \text{ par } S_2, s \rangle \Rightarrow \langle S_2, s' \rangle}$$

$$[par_{sos}^3] \quad \frac{\langle S_2, s \rangle \Rightarrow \langle S_2', s' \rangle}{\langle S_1 \text{ par } S_2, s \rangle \Rightarrow \langle S_1 \text{ par } S_2', s' \rangle}$$

$$[par_{sos}^4] \quad \frac{\langle S_2, s \rangle \Rightarrow s'}{\langle S_1 \text{ par } S_2, s \rangle \Rightarrow \langle S_1, s' \rangle}$$

The first two rules take account of the case where we begin by executing the first step of statement S_1 . if the execution of S_1 is not fully completed we modify the configuration so as to remember how far we have reached. Otherwise only S_2 has to be executed and we update the configuration accordingly. The last two rules are similar for the case where we begin by executing the first step of S_2 .

From the NS point of view we can not write extra rules in order to express the parallelism since we consider the execution of a statement as an atomic entity that cannot be split into smaller pieces. If we have tried to start defining some rules we could see this will not do because the rules only express that either S_1 is executed before S_2 or vice versa.

Conclusions:

1. In the natural semantics immediate constituent is an atomic entity so we cannot express interleaving of computations.
2. In the structural operational semantics we concentrate on small steps so interleaving of computations can be easily expressed

Local Variables and procedures (p. 60 - 68)

Syntax: $S ::= x := a \mid \text{skip} \mid S1 ; S2 \mid \text{if } b \text{ then } S1 \text{ else } S2 \mid \text{while } b \text{ do } S \mid$
 $\quad \text{begin } D_v D_p \text{ end} \mid \text{call } p$

$D_v := \text{var } x := a ; \mid D_v \ \varepsilon$

$D_p := \text{proc } p \text{ is } S ; \mid D_p \ \varepsilon$

* ε - empty declaration

Meaning: We extend the language with blocks containing declaration of variables and procedures. The idea that the variables declared inside a block statement are local to it.

In order to add these features to our language we need to add two new concepts:

1. Variable and procedures environment – (models the scope) this will allow us to write a transition function for variable and procedure declaration with a notation of \rightarrow_D which specifies the relationship between initial and final states. We generalize the substitution operation on states and write $s'[X \rightarrow s]$ for the state that is as s' except for variables in the set X where it is as specified by s . Formally,

$$s'[X \rightarrow s] = \begin{cases} sx & \text{if } x \in X \\ s'x & \text{if } x \notin X \end{cases}$$

2. Location and stores – (models the stack) in order to handle simple and recursive procedure calls we introduce the location of the environment. So rather than having a single mapping s from variables to values we have split it into two mapping env_v and sto and the idea is that $s = sto \circ env_v$ - to determine the variable's value we observe both the location and the relative value of the variable.

Giving these two new concepts we can write and represent new rules for our extended language, for example we shall see how to represent a block in the natural semantics:

$$[block_{NS}] \quad \frac{\langle D_v, s \rangle \rightarrow_D s', \langle S, s' \rangle \rightarrow s''}{\langle \text{begin } D_v \ S \ \text{end}, s \rangle \rightarrow s'' [DV(D_v) \rightarrow s]}$$

And variable declaration rule will be:

$$[var_{NS}] \quad \frac{\langle D_v, s[x \rightarrow A[a]s] \rangle \rightarrow_D s'}{\langle \text{var } x := a; D_v, s \rangle \rightarrow_D s'}$$

$$[none_{NS}] \quad \langle \varepsilon, s \rangle \rightarrow_D s$$

Example usage:

```
begin var y := 1;
  ( x := 1;
    begin var x := 2; y := x + 1 end;
    x := y + x)
end
```

- Note that there is a different usage with x as a local and global variable.

From the NS point of view we have one transition for each of the syntactic categories. The operation of begin / end will ensure that local variables are restored to their previous values when the block is let.

Conclusions:

1. The natural semantics can “remember” local states
2. Need to introduce stack or heap into state of the structural semantics

Transition Systems

Transition Systems are used to describe a more low-level semantics on programs that has relations over the states.

The advantage is that we focus on the lowest structure of a program.

More properties:

It is a private case of SOS. We include the program counter (pc) in the set of states Σ .

The meaning of a program is a relation $\tau \subseteq \Sigma \times \Sigma$ and the execution is finite number of states.

Summary

1. SOS is powerful enough to describe imperative programs, we can define the set of traces or represent program counter implicitly or even handle gotos.
2. Natural operational semantics is an abstraction
3. Different semantics may be used to justify different behaviors
4. Thinking in concrete semantics is essential for a compiler writer