1 Course Details

The course introduces formal semantics for programming languages. Its first part describes basic techniques for defining formal semantics, and draws mainly on [Winskel]. Its second part describes some applications of formal semantics, such as verification, analysis and type checking of programs, and draws mainly on [Pierce]. The course is essentially theoretical, employing such tools as structural induction and set theory, with little stress on algorithms.

2 Motivation for Formalizing Semantics

Every modern programming language has a well-defined syntax, which is specified formally, usually using context free grammar with contextual constraints. This enables to analyze the syntactic structure of a given string and to determine which strings are considered legal programs. However, description of the meaning that underlies the syntax is often left to informal, natural language. The semantics of a programming language specifies the meaning of programs, enabling to answer questions such as what is the program’s output for a given input, when does a runtime error occur, etc.

Defining meaning through formal semantics has several benefits:

- Program correctness is better assured when semantics is formally defined. Rigorously defined semantics ensures the programming language is well understood, thus saving design mistakes. Conversely, the looser the semantics or the syntax of a programming language are, the harder it is to use and maintain it.

- A formal semantics is essential in order to correctly design a compiler for a programming language. In particular, it is needed to ensure the correctness of program optimizations and to design static analyses.

- By providing an abstract (implementation-independent) notion of a program’s meaning, a formal semantics enhances programming language understanding (e.g., what does it mean for a program to be correct).

- A formal semantics can be used to rigorously check various aspects of program correctness, e.g., its correct use of types.

- A formal semantics provides a notion of functional equivalence between programs, which is of particular importance in compiler design.

- Automation: Given a formal specification of the meaning of a programming language, an interpreter for the programming language can be automatically generated.
• Putting semantics into a formal context eases and encourages advancements in software engineering
techniques. In particular, it helps in understanding advanced concepts such as inheritance, sub-typing,
and modularity.

We will focus on imperative programming languages, i.e. languages that describe a computation as a se-
quence of statements, which alter the state of a program. Examples of such languages are C, Pascal and PL/I.
Instances of other programming paradigms, with examples of corresponding languages in parentheses, are:
object-oriented (C++, Java and C#), functional (Scheme, ML, OCaml, F# and Haskell) and logic (Prolog).

3 Programming Language (PL) Semantics in General

3.1 Desired Features of PL Semantics

We would like PL semantics to hold several desirable properties:

• Tractable: following Occam’s razor principle, the semantics should be as simple as possible, without
losing the ability to accurately describe meaning and behavior.
• Abstract: the semantics should be as abstract as possible in the sense that it does not contain any
information (e.g., implementation details) irrelevant to the understanding of the computation.
• Computational: the semantics should correspond well to runtime behavior, and accurately describe
all information relevant to the understanding of the computation.
• Compositional: the meaning of a compound language construct is derived from the meaning of its
sub-constructs. (This supports modular reasoning and facilitates formal proofs).

However, fully attaining or combining all these properties may be infeasible. For example, the more abstract
a semantics is, the harder it is to keep it compositional.

3.2 Types of PL Semantics

The following types of PL semantics represent different approaches to defining meaning, and as such, they
also complement each other.

1. Operational Semantics [Plotkin, Kahn]
The simplest and most basic of PL semantics, they describe the meaning of the program as if by an
interpreter. They readily support a wide range of programming languages, including object-oriented
ones, but are short of abstractive power. We’ll survey three types of operational semantics: Trace-
based Semantics, Structural Operational Semantics (SOS) and Natural Semantics (NS).

2. Denotational Semantics [Strachey, Scott]
Describe the meaning of the program in terms of input/output relations, without referring to any of
the intermediate computational transitions that underly them. Denotational semantics exhibit a high
level of abstraction as well as compositionality. However, the high level of abstraction also leads to
their difficulty in formulating parallelism.
3. Aximoatic Semantics [Floyd, Hoare]

In essence, these are proof systems, which can be viewed as defining the meaning of a program in terms of its observed properties. Proof rules are applied to prove program correctness. Such a proof system can be seen as an implicit specification of the semantics of a programming language, and it must be kept sound with respect to the explicit (operational or denotational) specification.

4 IMP, a Simple Imperative Language

For instructional purposes, we now present a simple programming language named IMP. We begin by defining its syntax, then describe the meaning associated with the syntax using three different semantics.

Definition 1 (Abstract Syntax for IMP). The following defines an abstract syntax for IMP. The definition also introduces some notations of metavariables, i.e., symbols used to range over the syntactic categories. These notations apply for the rest of the document, where every metavariable \( s \) may later appear indexed \( (s^2) \) or primed \( (s') \) as well.

- Numbers \( N \)
  - Positive and negative numbers
  - Metavariables: \( n, m \in N \)

- Truth Values \( T \)
  - \( T = \{ \text{true}, \text{false} \} \)
  - Metavariabłe: \( t \in T \)

- Locations \( \text{Loc} \)
  - a.k.a program variables
  - Metavariables: \( X, Y \in \text{Loc} \)

- Arithmetic Expressions \( Aexp \)
  - Metavariabłe: \( a \in Aexp \)
  - \( a ::= (n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1) \)

- Boolean Expressions \( Bexp \)
  - Metavariabłe: \( b \in Bexp \)
  - \( b ::= (\text{true} \mid \text{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \land b_1 \mid b_0 \lor b_1) \)

- Commands \( \text{Com} \)
  - Metavariabłe: \( c \in \text{Com} \)
  - \( c ::= (\text{skip} \mid X := a \mid c_0; c_1 \mid \text{if } b \text{ then } c_0 \text{ else } c_1 \mid \text{while } b \text{ do } c) \)
This syntax is abstract in the sense that it is independent of the exact way in which the program is written. We do not define an order of precedence of arithmetic operations, but when writing down expressions and commands we shall include parentheses to impose order of association and guarantee a unique parse tree. For example, to disambiguate the parsing of 2 + 3 × 4 − 5, we may write (2 + (3 × 4)) − 5 or ((2 + 3) × 4) − 5, and the two parenthesized versions have different parse trees (i.e., they are not syntactically equivalent). Similarly, 3 + 5 is syntactically equivalent to (3 + 5), but not to 5 + 3.

An IMP program is any (syntactically valid) IMP command. For example, the command

\[
(Y := 1; \text{ while } \neg(X = 1) \text{ do } (Y := Y \times X; X := X - 1))
\]

is syntactically equivalent to following program, which computes \(X\!\!.

\[
Y := 1;
\text{ while } \neg(X = 1) \text{ do }
Y := Y \times X;
X := X - 1
\]

5 Trace-based Semantics

**Definition 2** (Program Meaning under Trace-based Semantics). Let \(P\) be a program. Denote: \(\Sigma(P)\) the set of potential states of \(P\), \(\Sigma(P)^*\) the set of finite traces over \(\Sigma(P)\), \(\Sigma(P)^\omega\) the set of infinite traces over \(\Sigma(P)\). Let \(\Pi(P)\) be the set of all traces over \(\Sigma(P)\), that is \(\Pi(P) = \Sigma(P)^* \cup \Sigma(P)^\omega\). The meaning of \(P\) is a subset of maximal traces over \(\Sigma(P)\), denoted \([P] \subseteq \Pi(P)\).

In the above definition, a trace is simply a sequence of program states. Thus, every terminating execution path of a program corresponds to a finite trace, and every non-terminating execution path corresponds to an infinite trace. A program state is a mapping that assigns values to the program’s variables and to the program counter (\(pc\)).

5.1 Example Programs

In the following example programs, every command is labeled with a number inside [ ] right before it.

5.1.1 Example 1

Consider the next program.

\[
[1] \text{ while } [2] (X > 0) \text{ do } \[3] X := X - 1
\]

This program terminates for every possible input, i.e., for every initial value of \(X\). It has an infinite number of (finite) traces, each corresponding to the execution path for a particular input. For example, the following trace corresponds to an execution in which the initial value of variable \(X\) is \(-7\), and hence the loop is not entered (the program counter jumps directly from label 2 to label 4):

\[
[pc \mapsto 1, X \mapsto -7]
[pc \mapsto 2, X \mapsto -7]
[pc \mapsto 4, X \mapsto -7]
\]
As another example, the following trace corresponds to an execution in which the initial value of $X$ is 2, and hence the loop body is executed twice:

\[
\begin{align*}
\text{pc} & \mapsto 1, \ X \mapsto 2 \\
\text{pc} & \mapsto 2, \ X \mapsto 2 \\
\text{pc} & \mapsto 3, \ X \mapsto 2 \\
\text{pc} & \mapsto 2, \ X \mapsto 2 \\
\text{pc} & \mapsto 3, \ X \mapsto 1 \\
\text{pc} & \mapsto 2, \ X \mapsto 0 \\
\text{pc} & \mapsto 4, \ X \mapsto 0
\end{align*}
\]

5.1.2 Example 2

Consider the following program.

\[
\begin{align*}
[1] & \text{ while } [2] (\text{true}) \text{ do} \\
[3] & \text{skip}
\end{align*}
\]

This program loops infinitely for every possible input. It has an infinite number of (infinite) traces, all of which share the same sequence of $pc$ values (starting at label 1 and then alternating indefinitely between labels 2 and 3), but differ in their sequence of values for variable $X$ (depending on $X$’s initial value). For example, the following trace corresponds to an execution in which the initial value of variable $X$ is 2:

\[
\begin{align*}
\text{pc} & \mapsto 1, \ X \mapsto 2 \\
\text{pc} & \mapsto 2, \ X \mapsto 2 \\
\text{pc} & \mapsto 3, \ X \mapsto 2 \\
\text{pc} & \mapsto 2, \ X \mapsto 2 \\
\text{pc} & \mapsto 3, \ X \mapsto 2 \\
\text{pc} & \mapsto 2, \ X \mapsto 2 \\
\text{pc} & \mapsto 3, \ X \mapsto 2
\end{align*}
\]

\[\vdots\]

5.2 Properties of Trace-based Semantics

Trace-based semantics resembles the way computer hardware processes a sequence of machine instructions. This makes it a complete method for describing the meaning of any program in any language (including concurrent programs), which makes it a nice candidate for model checking. However, it suffers some significant disadvantages:

- The program counter is an implementation-dependent detail.
- Trace-based semantics does not reflect our intuitive notion of program equivalence: It is often the case that two programs, that are equivalent in terms of the function they compute, have entirely different sets of traces (e.g., when one program is created as a compiler optimization of the other, and the transformation includes the addition of an extra label to the original program).
• Trace-based semantics is not easily defined by structural induction on the syntax.
• Trace-based semantics does not yield easily to proving interesting program properties.

Hence, in general, trace-based semantics lacks abstractive power. We shall next describe a more abstract semantics, which is nevertheless rather tractable.

6 Natural Operational Semantics

As its name implies, Natural Semantics is defined naturally, on top of the syntactic rules of a language. We’ll use it to define semantics for IMP.

Definition 3 (Program State). From here on, denote by $\sigma \in \Sigma$ a state of a program $P$ ($\sigma$ may also be indexed or primed). $\sigma$ maps locations (variables) to numbers (values), i.e. $\sigma : \text{Loc} \rightarrow \mathbb{N}$.

Note that $\text{Loc}$ is an infinite set of all possible locations. However, when describing the state of any particular program $P$, we shall only specify the values assigned to the finite set of variables that appear in $P$. For example, for a program $P \equiv (X := Y)$, the program state $\sigma = [X \mapsto 5, Y \mapsto 7]$ indicates that the value of $X$ is 5, the value of $Y$ is 7, and the value of any other variable $Z$ is undefined.

Note that, as opposed to trace-based semantics, the value of the program counter is not recorded in the program state.

6.1 Evaluation of Expressions

6.1.1 Evaluation Rules

Notation 1. For $X \in \text{Loc}$, denote by $\sigma(X)$ the value of $X$ in $\sigma$.

Notation 2. For $a \in \text{Aexp}$, $\sigma \in \Sigma$ and $n \in \mathbb{N}$, $\langle a, \sigma \rangle \rightarrow n$ indicates that evaluating the arithmetic expression $a$ in state $\sigma$ yields the numeric value $n$. Similarly, for $b \in \text{Bexp}$, $\sigma \in \Sigma$ and $t \in T$, $\langle b, \sigma \rangle \rightarrow t$ indicates that evaluating the boolean expression $b$ in state $\sigma$ yields the boolean value $t$.

Note 1. The relation $\rightarrow$ is not required to map each pair of expression and state to a single value. This allows us to treat cases where a given expression may evaluate, under a given state, to several different values or to no value at all (e.g., when an error such as division by zero occurs).

Notation 3. In inference rules, $\frac{i_1, i_2, \ldots, i_k}{j}$ indicates that if $(i_1 \land i_2 \land \ldots \land i_k)$ holds, then $j$ holds.

Definition 4 (Rules for Expression Evaluation). The way that $\sigma$ maps expressions to values is defined inductively as follows.

• Axioms
  1. Numbers: $\forall n \in \mathbb{N} : \langle n, \sigma \rangle \rightarrow n$
  2. Truth Values: $\langle \text{true}, \sigma \rangle \rightarrow \text{true}$, $\langle \text{false}, \sigma \rangle \rightarrow \text{false}$
  3. Locations: $\forall X \in \text{Loc} : \langle X, \sigma \rangle \rightarrow \sigma(X)$
• Inference Rules

1. Composite Arithmetic Expressions:
   \[ \forall a_0, a_1 \in \text{Aexp}, \otimes \in \{+, -, \times\} : \langle a_0, \sigma \rangle \rightarrow n_0, \langle a_1, \sigma \rangle \rightarrow n_1, \]
   \[ \langle a_0 \otimes a_1, \sigma \rangle \rightarrow n_0 \bullet n_1, \]
   where \( \bullet \) is the arithmetic operator corresponding to the syntactic token \( \otimes \).

2. Composite Boolean Expressions:
   \( (a) \quad \forall a_0, a_1 \in \text{Aexp}, \otimes \in \{=, \leq\} : \langle a_0, \sigma \rangle \rightarrow n, \langle a_1, \sigma \rangle \rightarrow m, \]
   \[ \langle a_0 \otimes a_1, \sigma \rangle \rightarrow n \bullet m, \]
   where \( \bullet \) is the comparison operator corresponding to the syntactic token \( \otimes \).
   \( (b) \quad \forall b \in \text{Bexp} : \langle b, \sigma \rangle \rightarrow \text{true}, \langle b, \sigma \rangle \rightarrow \text{false} \]
   \[ \langle \neg b, \sigma \rangle \rightarrow \text{false}, \langle \neg b, \sigma \rangle \rightarrow \text{true} \]
   \( (c) \quad \forall b_0, b_1 \in \text{Bexp}, \otimes \in \{\land, \lor\} : \langle b_0, \sigma \rangle \rightarrow t_0, \langle b_1, \sigma \rangle \rightarrow t_1, \]
   \[ \langle b_0 \otimes b_1, \sigma \rangle \rightarrow t_0 \bullet t_1, \]
   where \( \bullet \) is the boolean operator corresponding to the syntactic token \( \otimes \).

The rules are to be read as follows:

• Numbers: \( \langle n, \sigma \rangle \rightarrow n \) expresses the fact that a numeric constant symbol \( n \) evaluates to the numeric value \( n \) under any state \( \sigma \) (i.e., independently of the environment).

- Composite arithmetic expressions: The summation rule expresses the fact that if \( a_0 \) evaluates to \( n_0 \) in \( \sigma \) and \( a_1 \) evaluates to \( n_1 \) in \( \sigma \), then \( a_0 + a_1 \) evaluates to \( n_0 + n_1 \) in \( \sigma \). (Note that in the rule’s premises, both \( a_0 \) and \( a_1 \) are evaluated in the same environment \( \sigma \), since the evaluation of an expression has no side-effects.) Hence the summation rule is compositional, because it defines the value of the composite expression \( a_0 + a_1 \) using the values of its subexpressions \( a_0 \) and \( a_1 \). The rules for subtraction and product are similar.

- Boolean expressions: The axioms state that the boolean constant symbols \( \text{true} \) and \( \text{false} \) evaluate to their respective boolean values under any state \( \sigma \). The inference rules are compositional: The first set of rules (2a) defines the value of boolean expressions of the form \( a_0 = a_1 \) and \( a_0 \leq a_1 \), where the subexpressions \( a_0 \) and \( a_1 \) are arithmetic expressions. Thus, these rules use the evaluation of arithmetic expressions \( a_0 \) and \( a_1 \) as premises. The remaining rules (2b,2c) define the value of combinations of boolean expressions with the boolean connectives \( \neg \), \( \land \) and \( \lor \). Thus, these rules use the evaluation of boolean subexpressions as premises. For example, the conjunction rule states that if both \( b_0 \) and \( b_1 \) evaluate to \( \text{true} \) in \( \sigma \), then \( b_0 \land b_1 \) evaluates to \( \text{true} \) in \( \sigma \), whereas if at least one of \( b_0 \) and \( b_1 \) evaluates to \( \text{false} \) in \( \sigma \) then \( b_0 \land b_1 \) evaluates to \( \text{false} \) in \( \sigma \).

Note 2. What we have actually defined is a set of rule schemes, that represent an infinite collection of rule instances. A rule instance is formed by instantiating the metavariables that appear in the rule scheme with corresponding values (from their respective domains). For example, \( \langle n, \sigma \rangle \rightarrow n \) is an axiom scheme which may be instantiated by any particular number \( n \) and any particular state \( \sigma \).

Notation 4 \((\sigma_k)\). For \( k \in \mathbb{N} \), let \( \sigma_k \) denote a state, which maps every location \( X \) to \( k \), i.e. \( \forall X : \sigma_k(X) = k \). In particular, we’ll sometimes use \( \sigma_0 \) as an initial state, which maps every location to 0.
Example 1. From the axiom scheme \( \langle n, \sigma \rangle \rightarrow n \), we get the axiom instance \( \langle 2, \sigma_0 \rangle \rightarrow 2 \) by instantiating the metavariables \( n \) and \( \sigma \) with the values 2 and \( \sigma_0 \), respectively. Similarly, we get the axiom instance \( \langle 3, \sigma_0 \rangle \rightarrow 3 \). From the product inference rule scheme, we get the rule instance \( \frac{\langle 2, \sigma_0 \rangle \rightarrow 2, \langle 3, \sigma_0 \rangle \rightarrow 3}{\langle 2 \times 3, \sigma_0 \rangle \rightarrow 6} \) by instantiating the metavariables \( a_0 \), \( n_0 \), \( a_1 \), \( n_1 \), and \( \sigma \), with the values 2, 2, 3, 3, and \( \sigma_0 \), respectively. Thus, by combining the two axiom instances with the product rule instance, we can derive \( \langle 2 \times 3, \sigma_0 \rangle \rightarrow 6 \).

Note that \( \frac{\langle 2, \sigma_0 \rangle \rightarrow 2, \langle 3, \sigma_0 \rangle \rightarrow 4}{\langle 2 \times 3, \sigma_0 \rangle \rightarrow 12} \) is also a valid instance of the product rule scheme. However, since the incorrect premises \( \langle 2, \sigma_0 \rangle \rightarrow 3 \) and \( \langle 3, \sigma_0 \rangle \rightarrow 4 \) are not instances of the axiom scheme \( \langle n, \sigma \rangle \rightarrow n \), we will not be able to derive the incorrect conclusion \( \langle 2 \times 3, \sigma_0 \rangle \rightarrow 12 \).

Definition 5 (derivation tree). A derivation of an expression evaluation using the above rule system corresponds to a derivation tree. The leaves of the tree correspond to the utilized axioms, and the internal nodes correspond to the application of inference rules. The root of the tree contains the conclusion of the derivation. (Note that the root is drawn at the bottom).

Example 2. The evaluation \( \langle (\text{Init} + 5) + (7 + 9), \sigma_0 \rangle \rightarrow 21 \) may be derived by the following derivation tree:

\[
\frac{(\langle \text{Init}, \sigma_0 \rangle \rightarrow 0)}{(\langle (\text{Init} + 5), \sigma_0 \rangle \rightarrow 5)} \quad \frac{\langle (\text{Init} + 5), \sigma_0 \rangle \rightarrow 5}{\langle (7 + 9), \sigma_0 \rangle \rightarrow 16} \quad \frac{\langle (7 + 9), \sigma_0 \rangle \rightarrow 16}{\langle (\text{Init} + 5) + (7 + 9), \sigma_0 \rangle \rightarrow 21}
\]

In the left subtree, the axioms at the leaves express the fact that the value of variable \( \text{Init} \) in environment \( \sigma_0 \) is \( \sigma_0(\text{Init}) = 0 \), and the value of 5 in environment \( \sigma_0 \) is 5. Applying the summation inference rule to these two premises yields \( \langle (\text{Init} + 5), \sigma_0 \rangle \rightarrow 5 \) at the root’s left child. The right subtree is similar. By applying the summation rule at the root, we conclude from its left and right children that \( \langle (\text{Init} + 5) + (7 + 9), \sigma_0 \rangle \rightarrow 21 \).

The evaluation of an expression can be performed automatically (by an interpreter) by constructing a derivation in a top-down manner, starting at the root. In the above example, to evaluate the expression \( (\text{Init} + 5) + (7 + 9) \) at \( \sigma_0 \), we would begin with an unknown value at the root (denoted by \( \langle (\text{Init} + 5) + (7 + 9), \sigma_0 \rangle \rightarrow ? \)), and recurse to compute the unknown values of the subexpressions \( (\text{Init} + 5) \) and \( (7 + 9) \). Once we reach an expression that can be evaluated using an axiom (such as \( \text{Init} \) or 5), we compute its value and backtrack. While backtracking, we fill in the values at internal nodes by applying the corresponding inference rules. Note that in the general case: (1) there may be several possible derivations, all of which should be constructed in parallel; and (2) the computation process may not terminate, e.g., when an interpreter evaluates a program with an infinite loop.

6.1.2 Equivalence of Expressions

Definition 6 (Equivalence of IMP Arithmetic Expressions). Let \( a_0 \in \text{Aexp} \) and \( a_1 \in \text{Aexp} \) be two arithmetic expressions. We say that \( a_0 \) and \( a_1 \) are equivalent, denoted by \( a_0 \sim a_1 \), iff the following holds: \( \forall n \in \mathbb{N}, \sigma \in \Sigma : \langle a_0, \sigma \rangle \rightarrow n \iff \langle a_1, \sigma \rangle \rightarrow n \).

Intuitively, the above definition means that two arithmetic expressions are equivalent iff they evaluate to the same value in any environment.

Example 3. For any two expressions \( a_1 \) and \( a_2 \), we can prove that \( a_1 + a_2 \sim a_2 + a_1 \). Suppose that \( \langle a_1 + a_2, \sigma \rangle \rightarrow n \) for some \( n \in \mathbb{N} \) and \( \sigma \in \Sigma \). This can only be deduced from the summation rule, thus
there exist \( n_1 \in \mathbb{N} \) and \( n_2 \in \mathbb{N} \) such that (1) \( \langle a_1, \sigma \rangle \rightarrow n_1 \), (2) \( \langle a_2, \sigma \rangle \rightarrow n_2 \), and (3) \( n_1 + n_2 = n \). But since the summation arithmetic operator is commutative, (3) implies that (4) \( n_2 + n_1 = n \), and by combining (2), (1) and (4), and applying the summation rule, we can deduce \( \langle a_2 + a_1, \sigma \rangle \rightarrow n \). The opposite direction is symmetric.

The equivalence of boolean expressions is defined similarly to the equivalence of arithmetic expressions, as follows.

**Definition 7** (Equivalence of IMP Boolean Expressions). Let \( b_0 \in B_{\text{exp}} \) and \( b_1 \in B_{\text{exp}} \) be two boolean expressions. We say that \( b_0 \) and \( b_1 \) are equivalent, denoted by \( b_0 \sim b_1 \), iff the following holds: \( \forall t \in T, \sigma \in \Sigma : \langle b_0, \sigma \rangle \rightarrow t \iff \langle b_1, \sigma \rangle \rightarrow t \).

### 6.1.3 Extensions

The semantics we have defined for IMP expressions may be extended in various ways.

- We may allow rules for shortcut evaluation of boolean expressions, as in the C programming language. For example, if \( \langle b_0, \sigma \rangle \rightarrow \text{false} \), we may deduce that \( \langle b_0 \land b_1, \sigma \rangle \rightarrow \text{false} \) without the need to evaluate \( b_1 \).
- We may allow "parallel" evaluation of boolean expressions by concurrently-executing threads.
- We may treat additional data types, such as floating point numeric values.

We shall not describe such extensions here, but rather go on to define the semantics of IMP commands.

### 6.2 Execution of Commands

When defining the semantics for execution of commands, we’ll use an arrow notation similar to that used for evaluation of expressions. However, unlike evaluation of expressions, execution of commands may have a side effect on the state of the program.

**Notation 5.** For a command \( c \) and states \( \sigma, \sigma_f \), \( \langle c, \sigma \rangle \rightarrow \sigma_f \) indicates that the execution of \( c \) when the program is in state \( \sigma \) terminates, and yields final state \( \sigma_f \).

Note that describing only terminating executions is another way in which natural semantics abstracts away from the machine computation (in addition to not describing the program counter).

**Notation 6.** For a state \( \sigma \) and number \( n \), let \( \sigma[n/X] \) denote the state that maps location \( X \) to the number \( n \), and maps any other location to its value in state \( \sigma \). That is, for every location \( Y \):

\[
\sigma[n/X](Y) = \begin{cases} n & \text{if } Y = X \\ \sigma(Y) & \text{Otherwise} \end{cases}
\]

The above notation is used to describe the effect of an assignment command, which updates the value of a single location, e.g., \( \langle X := 5, \sigma \rangle \rightarrow \sigma[5/X] \).

**Definition 8** (Rules for Execution of Commands). The rules for execution of commands are defined inductively as follows.
Atomic Commands

1. **skip**: \( \langle \text{skip}, \sigma \rangle \rightarrow \sigma \)
2. **Assignment**: \( \frac{\langle a, \sigma \rangle \rightarrow n}{\langle X := a, \sigma \rangle \rightarrow \sigma[n/X]} \)

Composite Commands

1. **Sequencing**: \( \langle c_0, \sigma \rangle \rightarrow \sigma' \), \( \langle c_1, \sigma' \rangle \rightarrow \sigma'' \) \( \langle c_0; c_1, \sigma \rangle \rightarrow \sigma'' \)
2. **Conditionals**: 
   (a) \( \langle b, \sigma \rangle \rightarrow \text{true}, \langle c_0, \sigma \rangle \rightarrow \sigma' \) 
   (b) \( \langle b, \sigma \rangle \rightarrow \text{false}, \langle c_1, \sigma \rangle \rightarrow \sigma' \) 
3. **While**: 
   (a) \( \langle b, \sigma \rangle \rightarrow \text{false} \) 
   (b) \( \langle b, \sigma \rangle \rightarrow \text{true}, \langle c, \sigma \rangle \rightarrow \sigma', \langle \text{while } b \text{ do } c, \sigma' \rangle \rightarrow \sigma'' \) 

The rules are to be read as follows:

- **Assignment rule**: If expression \( a \) evaluates to \( n \) in state \( \sigma \), then executing the assignment \( X := a \) in \( \sigma \) results in state \( \sigma[n/X] \).

- **Sequencing rule**: If executing \( c_0 \) in state \( \sigma \) results in state \( \sigma' \), and executing \( c_1 \) in state \( \sigma' \) results in state \( \sigma'' \), then executing \( c_0; c_1 \) in state \( \sigma \) results in state \( \sigma'' \). (Note that the intermediate state \( \sigma' \) appears only in the rule’s premises.)

- **Conditional rules**: The execution of a conditional in state \( \sigma \) is equivalent to the execution of the then-clause in case the condition evaluates to \( \text{true} \) in \( \sigma \), and to the execution of the else-clause otherwise.

- **While rules**: The first rule states that a while loop whose condition evaluates to \( \text{false} \) in state \( \sigma \) is equivalent to \( \text{skip} \) and does not alter the state. The second rule states that if the loop condition evaluates to \( \text{true} \) in state \( \sigma \), a single execution of the loop body at state \( \sigma \) results in state \( \sigma' \), and an execution of the loop at state \( \sigma' \) results in state \( \sigma'' \), then the execution of the loop at state \( \sigma \) results in state \( \sigma'' \).

Notice that the last inference rule is exceptional: The rules for sequencing and conditionals, as well as the inference rule for a while loop whose condition evaluates to \( \text{false} \), are **compositional**, i.e. they define the meaning of a compound structure in terms of the meaning of its sub-structures. In contrast, the inference rule for a while loop whose condition evaluates to \( \text{true} \) defines the meaning of \( \text{while } b \text{ do } c \) in terms of itself. This can be seen by \( \text{while } b \text{ do } c \) appearing in the upper part of the rule, instead of just its constituents \( b \) and \( c \).

Indeed, we are left with no choice but to define the meaning of \( \text{while} \) by induction on the **computation** (i.e. on the number of iterations executed) instead of on the syntax – repeated application of this rule amounts to loop-peeling one iteration at a time. Thus, if executing the loop at state \( \sigma \) terminates after \( n \) iterations in
Definition 9 (Equivalence of Commands). Commands $c_1$ and $c_2$ are equivalent, denoted by $c_1 \sim c_2$, iff 
$\forall \sigma, \sigma' \in \Sigma : \langle c_1, \sigma \rangle \rightarrow \sigma' \iff \langle c_2, \sigma \rangle \rightarrow \sigma'$.

Intuitively, two commands are equivalent if they are indistinguishable under the given semantics. I.e., if 
executing one command in state $\sigma$ yields state $\sigma'$, then executing the other command in state $\sigma$ also yields 
state $\sigma'$. 

Example 5. Consider the program $F \equiv (Y := 1; \text{while } \neg(X = 1) \text{ do } (Y := Y \times X; X := X - 1))$, and 
denote $c \equiv (Y := Y \times X; X := X - 1), w \equiv \text{while } \neg(X = 1) \text{ do } c$. We show that $\langle F, \sigma_3 \rangle \rightarrow \sigma_3[1/X, 6/Y]$ 
by the following derivation tree, which is displayed in a top-down manner. For brevity, we omit the evaluations 
of some assignments and expressions.
Proposition 1. \((\text{while } b \text{ do } c) \sim (\text{if } b \text{ then } (c; \text{while } b \text{ do } c) \text{ else skip})\)

The proof can be found in [Winskel] (p. 21-23).

7 Small Step Operational Semantics

Natural semantics, which is also called Large Step Operational Semantics, is too abstract for certain needs, such as describing concurrent computations. An alternative formalism is Structural Operational Semantics (SOS), which is also called Small Step Operational Semantics. SOS is more concrete than natural semantics in the sense that it describes the individual steps along the course of the execution, and may describe non-terminating as well as terminating executions. (Yet, similar to natural semantics, SOS abstracts away the value of the program counter.)

Notation 7. For expressions \(a\) and \(a'\), and states \(\sigma\) and \(\sigma'\), \(\langle a, \sigma \rangle \rightarrow^1 \langle a', \sigma' \rangle\) indicates that one step of evaluating \(a\) in state \(\sigma\) yields \(a'\) in state \(\sigma'\). (Intuitively, \(a'\) is the expression that remains to be evaluated.)

We shall now define Small Step Operational Semantics for IMP.

Definition 10 (SOS Rules for Addition). The rules for addition are defined as follows.

\[
1. \quad \langle a_0, \sigma \rangle \rightarrow^1 \langle a'_0, \sigma \rangle \\
\langle a_0 + a_1, \sigma \rangle \rightarrow^1 \langle a'_0 + a_1, \sigma \rangle \\
2. \quad \langle a_1, \sigma \rangle \rightarrow^1 \langle a'_1, \sigma \rangle \\
\langle n + a_1, \sigma \rangle \rightarrow^1 \langle n + a'_1, \sigma \rangle \\
3. \quad \langle n + m, \sigma \rangle \rightarrow^1 \langle p, \sigma \rangle \text{ where } p = n + m
\]

The above rules demonstrate the way evaluation of arithmetic expressions is defined in SOS. Sums are evaluated from left to right: first the left-hand sub-expression is evaluated, and only when its evaluation is completed and yields some number \(n\), do we go on to evaluate the right-hand sub-expression. (Note that each of these sub-expressions might itself be a compound expression, such as \(X + Y + Z\).) Finally, when both left and right sub-expressions have been evaluated, their corresponding numeric values are added. Admittedly, describing evaluation of arithmetic expressions in this way is a bit artificial. Therefore, it is often customary to combine the expression evaluation rules of large step semantics with the command execution rules of SOS.

Definition 11 (SOS Rules for Execution of Commands). The rules for execution of commands are defined as follows.

\[
1. \quad \text{skip: } \langle \text{skip}, \sigma \rangle \rightarrow^1 \sigma \\
2. \quad \text{Assignment: } \langle a, \sigma \rangle \rightarrow^1 n \quad \langle X := a, \sigma \rangle \rightarrow^1 \sigma[n/X] \\
3. \quad \text{Sequencing:} \\
(a) \quad \langle c_0, \sigma \rangle \rightarrow^1 \langle c'_0, \sigma' \rangle \\
\langle c_0; c_1, \sigma \rangle \rightarrow^1 \langle c'_0; c_1, \sigma' \rangle
\]
4. Conditionals:

(a) \( \langle b, \sigma \rangle \rightarrow \text{true} \)
\( \langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma \rangle \rightarrow \langle c_1, \sigma \rangle \)

(b) \( \langle b, \sigma \rangle \rightarrow \text{false} \)
\( \langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma \rangle \rightarrow \langle c_2, \sigma \rangle \)

5. While: \( \langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \langle \text{if } b \text{ then } (c; \text{while } b \text{ do } c) \text{ else } \text{skip}, \sigma \rangle \)

The assignment rule is similar to that for natural semantics. The sequencing rules state that a sequence of commands is executed from left to right: to execute \( c_0; c_1 \), we first execute \( c_0 \) one step at a time, and once its execution is completed we go on to execute \( c_1 \). The conditional rules state that if the condition is true (resp. false), then in one step we switch to executing the then-clause (resp. else-clause). Finally, the while rule unfolds a single iteration of the loop in one step.

8 Summary

In this lesson, we have focused on operational semantics. Operational semantics is useful since it enables to naturally express program behavior, and it can handle a variety of language features such as non-determinism and concurrency. However, in many respects, operational semantics remains too close to the implementation (e.g., two programs that compute the same function using different temporary variables are not considered equivalent under operational semantics, contrary to our desired notion of equivalence). In following lessons we will survey more abstract types of semantics.

References
