Small-step Operational Semantics

Lecture 2 Thursday, January 25, 2024

1 Intro to semantics

What is the meaning of a program? When we write a program, we use a sequence of characters to represent the program. But this *syntax* is just how we represent the program: it is not what the program means.

Maybe we could define the meaning of a program to be whatever happens when we execute the program (perhaps using an interpreter, or by compiling it first). But we can have bugs in interpreters and compilers! That is, an interpreter or compiler may not accurately reflect the meaning of a program. So we must look elsewhere for a definition of what a program means.

One place to look for the meaning of a program is in the language specification manual. Such manuals typically give an informal description of the language constructs.

Another option is to give a formal, mathematical definition of the language semantics. A formal mathematical definition can have the following advantages over an informal description.

- **Less ambiguous.** The behavior of the language is clearer, which is useful for anyone who needs to write programs in the language, implement a compiler or interpreter for the language, add a new feature to the language, etc.
- **More concise.** Mathematical concepts and notation can clearly and concise describe a language, and state restrictions on legal programs in the language. For example, the Java Language Specification (2rd edition) devotes a chapter (26 pages) to describing the concept of *definite assignment*, most of which is describing, in English, a dataflow analysis that can be expressed more succinctly using mathematics.
- **Formal arguments.** Most importantly, a formal semantics allows us to state, and prove, program properties that we're interested in. For example: we can state and prove that *all* programs in a language are guaranteed to be free of certain run-time errors, or free of certain security violations; we can state the specification of a program and prove that the program meets the specification (i.e., that the program is guaranteed to produce the correct output for all possible inputs).

However, the drawback of formal semantics is that they can lead to fairly complex mathematical models, especially if one attempts to describe all details in a full-featured modern language. Few real programming languages have a formal semantics, since modeling all the details of a real-world language is hard: real languages are complex, with many features. In order to describe these features, both the mathematics and notation for modeling can get very dense, making the formal semantics difficult to understand. Indeed, sometimes novel mathematics and notation needs to be developed to accurately model language features. So while there can be many benefits to having a formal semantics for a programming language, they do not yet outweigh the costs of developing and using a formal semantics for real programming languages.

There are three main approaches to formally specify the semantics of programming languages:

- operational semantics: describes how a program would execute on an abstract machine;
- denotational semantics: models programs as mathematical functions;
- axiomatic semantics: defines program behavior in terms of the logical formulae that are satisfied before and after a program;

Each of these approaches has different advantages and disadvantages in terms of how mathematically sophisticated they are, how easy it is to use them in proofs, or how easy it is to implement an interpreter or compiler based on them.

2 A simple language of arithmetic expressions

To understand some of the key concepts of semantics, we will start with a very simple language of integer arithmetic expressions, with assignment. A program in this language is an expression; executing a program means evaluating the expression to an integer.

To describe the structure of this language we will use the following domains:

$$x, y, z \in \textbf{Var}$$
$$n, m \in \textbf{Int}$$
$$e \in \textbf{Exp}$$

Var is the set of program variables (e.g., foo, bar, baz, i, etc.). **Int** is the set of constant integers (e.g., 42, -40, 7). **Exp** is the domain of expressions, which we specify using a BNF (Backus-Naur Form) grammar:

$$e ::= x \mid n \mid e_1 + e_2 \mid e_1 \times e_2 \mid x := e_1; e_2$$

Informally, the expression $x := e_1; e_2$ means that x is assigned the value of e_1 before evaluating e_2. The result of the entire expression is that of e_2.

This grammar specifies the syntax for the language. An immediate problem here is that the grammar is ambiguous. Consider the expression $1 + 2 \times 3$. One can build two abstract syntax trees:

There are several ways to deal with this problem. One is to rewrite the grammar for the same language to make it unambiguous. But that makes the grammar more complex, and harder to understand. Another possibility is to extend the syntax to require parentheses around all expressions:

$$x \mid n \mid (e_1 + e_2) \mid (e_1 \times e_2) \mid x := e_1; e_2$$

However, this also leads to unnecessary clutter and complexity.

Instead, we separate the "concrete syntax" of the language (which specifies how to unambiguously parse a string into program phrases) from the "abstract syntax" of the language (which describes, possibly ambiguously, the structure of program phrases). In this course we will use the abstract syntax and assume that the abstract syntax tree is known. When writing expressions, we will occasionally use parenthesis to indicate the structure of the abstract syntax tree, but the parentheses are not part of the language itself. (For details on parsing, grammars, and ambiguity elimination, see or take the compiler course Computer Science 153.)

3 Small-step operational semantics

At this point we have defined the syntax of our simple arithmetic language. We have some informal, intuitive notion of what programs in this language mean. For example, the program $7 + (4 \times 2)$ should equal 15, and the program $\mathsf{foo} := 6 + 1; 2 \times 3 \times \mathsf{foo}$ should equal 42.

We would like now to define formal semantics for this language.

Operational semantics describe how a program would execute on an abstract machine. A *small-step operational semantics* describe how such an execution in terms of successive reductions of an expression, until we reach a number, which represents the result of the computation.

The state of the abstract machine is usually referred to as a configuration, and for our language it must include two pieces of information:

- a store (also called an *environment* or *state*), which assigns integer values to variables. During program execution, we will refer to the store to determine the values associated with variables, and also update the store to reflect assignment of new values to variables.
- the expression left to evaluate.

Thus, the domain of stores is functions from **Var** to **Int** (written **Var** $\to$ **Int**), and the domain of configurations is pairs of expressions and stores.

$$\textbf{Config} = \textbf{Exp} \times \textbf{Store}$$
$$\textbf{Store} = \textbf{Var} \to \textbf{Int}$$

We will denote configurations using angle brackets. For instance, $\langle(\mathsf{foo} + 2) \times (\mathsf{bar} + 1), \sigma\rangle$, where σ is a store and $(\mathsf{foo} + 2) \times (\mathsf{bar} + 1)$ is an expression that uses two variables, foo and bar.

The small-step operational semantics for our language is a relation $\longrightarrow \subseteq \textbf{Config} \times \textbf{Config}$ that describes how one configuration transitions to a new configuration.[1] That is, the relation $\longrightarrow$ shows us how to evaluate programs, one step at a time. We use infix notation for the relation $\longrightarrow$. That is, given any two configurations $\langle e_1, \sigma_1\rangle$ and $\langle e_2, \sigma_2\rangle$, if $(\langle e_1, \sigma_1\rangle, \langle e_2, \sigma_2\rangle)$ is in the relation $\longrightarrow$, then we write $\langle e_1, \sigma_1\rangle \longrightarrow \langle e_2, \sigma_2\rangle$.

For example, we have $\langle(4 + 2) \times \mathsf{y}, \sigma\rangle \longrightarrow \langle 6 \times \mathsf{y}, \sigma\rangle$. That is, we can evaluate the configuration $\langle(4 + 2) \times \mathsf{y}, \sigma\rangle$ by one step, to get the configuration $\langle 6 \times \mathsf{y}, \sigma\rangle$.

Now defining the semantics of the language boils down to defining the relation $\longrightarrow$ that describes the transitions between machine configurations.

One issue here is that the domain of integers is infinite, and so is the domain of expressions. Therefore, there is an infinite number of possible machine configurations, and an infinite number of possible one-step transitions. We need to use a finite description for the infinite set of transitions.

We can compactly describe the transition function $\longrightarrow$ using inference rules:

$$\textsc{Var}\ \frac{}{\langle x, \sigma\rangle \longrightarrow \langle n, \sigma\rangle}\ \text{where } n = \sigma(x)$$

$$\textsc{LAdd}\ \frac{\langle e_1, \sigma\rangle \longrightarrow \langle e_1', \sigma'\rangle}{\langle e_1 + e_2, \sigma\rangle \longrightarrow \langle e_1' + e_2, \sigma'\rangle} \qquad \textsc{RAdd}\ \frac{\langle e_2, \sigma\rangle \longrightarrow \langle e_2', \sigma'\rangle}{\langle n + e_2, \sigma\rangle \longrightarrow \langle n + e_2', \sigma'\rangle}$$

$$\textsc{Add}\ \frac{}{\langle n + m, \sigma\rangle \longrightarrow \langle p, \sigma\rangle}\ \text{where } p \text{ is the sum of } n \text{ and } m$$

$$\textsc{LMul}\ \frac{\langle e_1, \sigma\rangle \longrightarrow \langle e_1', \sigma'\rangle}{\langle e_1 \times e_2, \sigma\rangle \longrightarrow \langle e_1' \times e_2, \sigma'\rangle} \qquad \textsc{RMul}\ \frac{\langle e_2, \sigma\rangle \longrightarrow \langle e_2', \sigma'\rangle}{\langle n \times e_2, \sigma\rangle \longrightarrow \langle n \times e_2', \sigma'\rangle}$$

$$\textsc{Mul}\ \frac{}{\langle n \times m, \sigma\rangle \longrightarrow \langle p, \sigma\rangle}\ \text{where } p \text{ is the product of } n \text{ and } m$$

$$\textsc{Asg1}\ \frac{\langle e_1, \sigma\rangle \longrightarrow \langle e_1', \sigma'\rangle}{\langle x := e_1; e_2, \sigma\rangle \longrightarrow \langle x := e_1'; e_2, \sigma'\rangle} \qquad \textsc{Asg}\ \frac{}{\langle x := n; e_2, \sigma\rangle \longrightarrow \langle e_2, \sigma[x \mapsto n]\rangle}$$

[1] If you are not familiar and comfortable with the concept of *relations*, please go to the course website and find the readings for background material.

The meaning of an inference rule is that if the fact above the line holds and the side conditions also hold, then the fact below the line holds. The fact(s) above the line are called premises; the fact below the line is called the conclusion. The rules without premises are axioms; and the rules with premises are inductive rules.

Also, we use the notation $\sigma[x \mapsto n]$ for a store that maps the variable x to integer n, and maps every other variable to whatever σ maps it to. More explicitly, if f is the function $\sigma[x \mapsto n]$, then we have

$$f(y) = \begin{cases} n & \text{if } y = x \\ \sigma(y) & \text{otherwise} \end{cases}$$

4 Using the Semantic Rules

Let's see how we can use these rules. Suppose we want to evaluate expression $(\mathsf{foo} + 2) \times (\mathsf{bar} + 1)$ in a store σ where $\sigma(\mathsf{foo}) = 4$ and $\sigma(\mathsf{bar}) = 3$. That is, we want to find the transition for configuration $\langle(\mathsf{foo} + 2) \times (\mathsf{bar} + 1), \sigma\rangle$. For this, we look for a rule with this form of a configuration in the conclusion. By inspecting the rules, we find that the only matching rule is LMUL, where $e_1 = \mathsf{foo} + 2$, $e_2 = \mathsf{bar} + 1$, but e_1' is not yet known. We can *instantiate* the rule LMUL, replacing the metavariables e_1 and e_2 with appropriate expressions.

$$\textsc{LMul}\ \frac{\langle \mathsf{foo} + 2, \sigma\rangle \longrightarrow \langle e_1', \sigma'\rangle}{\langle(\mathsf{foo} + 2) \times (\mathsf{bar} + 1), \sigma\rangle \longrightarrow \langle e_1' \times (\mathsf{bar} + 1), \sigma'\rangle}$$

Now we need to show that the premise actually holds and find out what e_1' and σ' are. We look for a rule whose conclusion matches $\langle \mathsf{foo} + 2, \sigma\rangle \longrightarrow \langle e_1', \sigma'\rangle$. We find that LADD is the only matching rule:

$$\textsc{LAdd}\ \frac{\langle \mathsf{foo}, \sigma\rangle \longrightarrow \langle e_1'', \sigma'\rangle}{\langle \mathsf{foo} + 2, \sigma\rangle \longrightarrow \langle e_1''+2, \sigma'\rangle}$$

where $e_1' = e_1''+2$. We repeat this reasoning for $\langle \mathsf{foo}, \sigma\rangle \longrightarrow \langle e_1'', \sigma'\rangle$, and we find that the only applicable rule is the axiom VAR:

$$\textsc{Var}\ \frac{}{\langle \mathsf{foo}, \sigma\rangle \longrightarrow \langle 4, \sigma\rangle}$$

because we have $\sigma(\mathsf{foo}) = 4$. Since this is an axiom and has no premises, there is nothing left to prove. Hence, $e'' = 4$ and $e_1' = 4 + 2$ and $\sigma' = \sigma$. We can put together the above pieces and build the following proof:

$$\textsc{LMul}\ \frac{\textsc{LAdd}\ \dfrac{\textsc{Var}\ \dfrac{}{\langle \mathsf{foo}, \sigma\rangle \longrightarrow \langle 4, \sigma\rangle}}{\langle \mathsf{foo} + 2, \sigma\rangle \longrightarrow \langle 4 + 2, \sigma\rangle}}{\langle(\mathsf{foo} + 2) \times (\mathsf{bar} + 1), \sigma\rangle \longrightarrow \langle(4 + 2) \times (\mathsf{bar} + 1), \sigma\rangle}$$

This proves that, given our inference rules, the one-step transition $\langle(\mathsf{foo} + 2) \times (\mathsf{bar} + 1), \sigma\rangle \longrightarrow \langle(4 + 2) \times (\mathsf{bar} + 1), \sigma\rangle$ is possible. The above proof structure is called a "proof tree" or "derivation". It is important to keep in mind that proof trees must be finite for the conclusion to be valid.

We can use a similar reasoning to find out the next evaluation step:

$$\textsc{LMul}\ \frac{\textsc{Add}\ \dfrac{}{\langle 4 + 2, \sigma\rangle \longrightarrow \langle 6, \sigma\rangle}}{\langle(4 + 2) \times (\mathsf{bar} + 1), \sigma\rangle \longrightarrow \langle 6 \times (\mathsf{bar} + 1), \sigma\rangle}$$

And we can continue this process. At the end, we can put together all of these transitions, to get a view of the entire computation:

$$\begin{aligned}\langle(\mathsf{foo}+2)\times(\mathsf{bar}+1),\sigma\rangle &\longrightarrow\langle(4+2)\times(\mathsf{bar}+1),\sigma\rangle\\ &\longrightarrow\langle 6\times(\mathsf{bar}+1),\sigma\rangle\\ &\longrightarrow\langle 6\times(3+1),\sigma\rangle\\ &\longrightarrow\langle 6\times 4,\sigma\rangle\\ &\longrightarrow\langle 24,\sigma\rangle\end{aligned}$$

The result of the computation is a number, 24. The machine configuration that contains the final result is the point where the evaluation stops; they are called *final configurations*. For our language of expressions, the final configurations are of the form $\langle n, \sigma\rangle$ where n is a number and σ is a store.

We write $\longrightarrow^*$ for the reflexive transitive closure of the relation $\longrightarrow$. That is, if $\langle e, \sigma\rangle \longrightarrow^* \langle e', \sigma'\rangle$, then using zero or more steps, we can evaluate the configuration $\langle e, \sigma\rangle$ to the configuration $\langle e', \sigma'\rangle$. Thus, we can write

$$\langle(\mathsf{foo}+2)\times(\mathsf{bar}+1),\sigma\rangle \longrightarrow^* \langle 24,\sigma\rangle.$$