Lecture Notes: Program Semantics

17-355/17-665/17-819O: Program Analysis (Spring 2018)
Claire Le Goues and Jonathan Aldrich
clegoues@cs.cmu.edu, aldrich@cs.cmu.edu

1 Operational Semantics

To reason about analysis correctness, we need a clear definition of what a program *means*. One way to do this is using natural language (e.g., the Java Language Specification). However, although natural language specifications are accessible, they are also often imprecise. This can lead to many problems, including incorrect compiler implementations or program analyses.

A better alternative is a formal definition of program semantics. We begin with *operational semantics*, which mimics, at a high level, the operation of a computer executing the program. Such a semantics also reflects the way that techniques such as dataflow analysis or Hoare Logic reason about the program, so it is convenient for our purposes.

There are two broad classes of operational semantics: *big-step operational semantics*, which specifies the entire operation of a given expression or statement; and *small-step operational semantics*, which specifies the operation of the program one step at a time.

1.1 WHILE: Big-step operational semantics

We'll start by restricting our attention to arithmetic expressions, for simplicity. What is the meaning of a WHILE expression? Some expressions, like a natural number, have a very clear meaning: The "meaning" of 5 is just, well, 5. But what about $x + 5$? The meaning of this expression clearly depends on the value of the variable x. We must *abstract* the value of variables as a function from variable names to integer values:

$$E \in \mathit{Var} \to \mathbb{Z}$$

Here E denotes a particular program *state*. The meaning of an expression with a variable like $x+5$ involves "looking up" the x's value in the associated E, and substituting it in. Given a state, we can write a *judgement* as follows:

$$\langle e, E \rangle \Downarrow n$$

This means that given program state E, the expression e evaluates to n. This formulation is called *big-step* operational semantics; the $\Downarrow$ judgement relates an expression and its "meaning."[1] We then build up the meaning of more complex expressions using *rules of inference* (also called *derivation* or *evaluation* rules). An inference rule is made up of a set of judgments above the line, known as *premises*, and a judgment below the line, known as the *conclusion*. The meaning of an inference rule is that the conclusion holds if all of the premises hold:

[1] Note that I have chosen $\Downarrow$ because it is a common notational convention; it's not otherwise special. This is true for many notational choices in formal specification.

$$\frac{\mathit{premise}_1 \quad \mathit{premise}_2 \quad \ldots \quad \mathit{premise}_n}{\mathit{conclusion}}$$

An inference rule with no premises is an *axiom*, which is always true. For example, integers always evaluate to themselves, and the meaning of a variable is its stored value in the state:

$$\frac{}{\langle n, E \rangle \Downarrow n}\ \textit{big-int} \qquad \frac{}{\langle x, E \rangle \Downarrow E(x)}\ \textit{big-var}$$

Addition expressions illustrate a rule with premises:

$$\frac{\langle e_1, E \rangle \Downarrow n_1 \quad \langle e_2, E \rangle \Downarrow n_2}{\langle e_1 + e_2, E \rangle \Downarrow n_1 + n_2}\ \textit{big-add}$$

But, how does the value of x come to be "stored" in E? For that, we must consider WHILE *Statements*. Unlike expressions, statements have no direct result. However, they can have *side effects*. That is to say: the "result" or *meaning* of a Statement is a *new state*. The judgement $\Downarrow$ as applied to statements and states therefore looks like:

$$\langle S, E \rangle \Downarrow E'$$

This allows us to write inference rules for statements, bearing in mind that their *meaning* is not an integer, but a new state. The meaning of `skip`, for example, is an unchanged state:

$$\frac{}{\langle \texttt{skip}, E \rangle \Downarrow E}\ \textit{big-skip}$$

Statement sequencing, on the other hand, does involve premises:

$$\frac{\langle s_1, E \rangle \Downarrow E' \quad \langle s_2, E' \rangle \Downarrow E''}{\langle s_1; s_2, E \rangle \Downarrow E''}\ \textit{big-seq}$$

The `if` statement involves two rules, one for if the boolean predicate evaluates to `true` (rules for boolean expressions not shown), and one for if it evaluates to `false`. I'll show you just the first one for demonstration:

$$\frac{\langle b, E \rangle \Downarrow \texttt{true} \quad \langle s_1, E \rangle \Downarrow E'}{\langle \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2, E \rangle \Downarrow E'}\ \textit{big-iftrue}$$

What should the second rule for `if` *look like?*

This brings us to assignments, which produce a new state in which the variable being assigned to is mapped to the value from the right-hand side. We write this with the notation $E[x \mapsto n]$, which can be read "a new state that is the same as E except that x is mapped to n."

$$\frac{\langle e, E \rangle \Downarrow n}{\langle x := e, E \rangle \Downarrow E[x \mapsto n]}\ \textit{big-assign}$$

Note that the update to the state is modeled *functionally*; the variable E still refers to the old state, while $E[x \mapsto n]$ is the new state represented as a mathematical map.

Fully specifying the semantics of a language requires a judgement rule like this for every language construct. These notes only include a subset for WHILE, for brevity.

Exercise 1. What are the rule(s) for `while`?

1.2 WHILE: Small-step operational semantics

Big-step operational semantics has its uses. Among other nice features, it directly suggests a simple interpreter implementation for a given language. However, it is difficult to talk about a statement or program whose evaluation does not terminate. Nor does it give us any way to talk about intermediate states (so modeling multiple threads of control is out).

Sometimes it is instead useful to define a *small-step operational semantics*, which specifies program execution one step at a time. We refer to the pair of a statement and a state ($\langle S, E\rangle$) as a *configuration*. Whereas big step semantics specifies program meaning as a function between a configuration and a new state, small step models it as a step from one configuration to another.

You can think of small-step semantics as a set of rules that we repeatedly apply to configurations until we reach a *final configuration* for the language ($\langle \texttt{skip}, E\rangle$, in this case) if ever.[2] We write this new judgement using a slightly different arrow: $\to$. $\langle S, E\rangle \to \langle S', E'\rangle$ indicates one step of execution; $\langle S, E\rangle \to^* \langle S', E'\rangle$ indicates zero or more steps of execution. We formally define multiple execution steps as follows:

$$\frac{}{\langle S, E\rangle \to^* \langle S, E\rangle}\ \textit{multi-reflexive} \qquad \frac{\langle S, E\rangle \to \langle S', E'\rangle \quad \langle S', E'\rangle \to^* \langle S'', E''\rangle}{\langle S, E\rangle \to^* \langle S'', E''\rangle}\ \textit{multi-inductive}$$

To be complete, we should also define auxiliary small-step operators $\to_a$ and $\to_b$ for arithmetic and boolean expressions, respectively; only the operator for statements results in an updated state (as in big step). The types of these judgements are thus:

$$\begin{array}{rcl} \to & : & (\texttt{Stmt} \times E) \to (\texttt{Stmt} \times E) \\ \to_a & : & (\texttt{Aexp} \times E) \to \texttt{Aexp} \\ \to_b & : & (\texttt{Bexp} \times E) \to \texttt{Bexp} \end{array}$$

We can now again write the semantics of a WHILE program as new rules of inference. Some rules look very similar to the big-step rules, just with a different arrow. For example, consider variables:

$$\frac{}{\langle x, E\rangle \to_a E(x)}\ \textit{small-var}$$

Things get more interesting when we return to statements. Remember, small-step semantics express a single execution step. So, consider an `if` statement:

$$\frac{\langle b, E\rangle \to_b\ b'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, E\rangle \to \langle \texttt{if}\ b'\ \texttt{then}\ S_1\ \texttt{else}\ S_2, E\rangle}\ \textit{small-if-congruence}$$

$$\frac{}{\langle \texttt{if true then}\ s_1\ \texttt{else}\ s_2, E\rangle \to \langle s_1, E'\rangle}\ \textit{small-iftrue}$$

Exercise 2. We have again omitted the *small-iffalse* case, as well as rule(s) for `while`, as exercises to the reader.

Note also the change for statement sequencing:

$$\frac{\langle s_1, E\rangle \to \langle s_1', E'\rangle}{\langle s_1; s_2, E\rangle \to \langle s_1'; s_2 E'\rangle}\ \textit{small-seq-congruence}$$

[2] Not all statements reach a final configuration, like `while true do skip`.

$$\frac{}{\langle \texttt{skip}; s_2, E\rangle \to \langle s_2, E\rangle}\ \textit{small-seq}$$

1.3 WHILE3ADDR: Small-step semantics

The ideas behind big- and small-step operational semantics are consistent across languages, but the way they are written can vary based on what is notationally convenient for a particular language or analysis. WHILE3ADDR is slightly different from WHILE, so beyond requiring different rules for its different constructs, it makes sense to modify our small-step notation a bit for defining the meaning of a WHILE3ADDR program.

First, let's revisit the *configuration* to account for the slightly different *meaning* of a WHILE3ADDR program. As before, the configuration must include the state, which we still call E, mapping variables to values. However, a well-formed, terminating WHILE program was effectively a single statement that can be iteratively reduced to `skip`; a WHILE3ADDR program, on the other hand, is a mapping from natural numbers to program instructions. So, instead of a statement that is being reduced in steps, the WHILE3ADDR c must includes a program counter n, representing the next instruction to be executed.

Thus, a configuration c of the abstract machine for WHILE3ADDR must include the stored program P (which we will generally treat implicitly), the state environment E, and the current program counter n representing the next instruction to be executed ($c \in E \times \mathbb{N}$). The abstract machine executes one step at a time, executing the instruction that the program counter points to, and updating the program counter and environment according to the semantics of that instruction.

This adds a tiny bit of complexity to the inference rules, because they must explicitly consider the mapping between line number/labels and program instructions. We represent execution of the abstract machine via a judgment of the form $P \vdash \langle E, n\rangle \rightsquigarrow \langle E', n'\rangle$ The judgment reads: "When executing the program P, executing instruction n in the state E steps to a new state E' and program counter n'."[3] To see this in action, consider a simple inference rule defining the semantics of the constant assignment instruction:

$$\frac{P[n] = x := m}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E[x \mapsto m], n+1\rangle}\ \textit{step-const}$$

This states that in the case where the nth instruction of the program P (looked up using $P[n]$) is a constant assignment $x := m$, the abstract machine takes a step to a state in which the state E is updated to map x to the constant m, written as $E[x \mapsto m]$, and the program counter now points to the instruction at the following address $n + 1$. We similarly define the remaining rules:

$$\frac{P[n] = x := y}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E[x \mapsto E[y]], n+1\rangle}\ \textit{step-copy}$$

$$\frac{P[n] = x := y\ op\ z \quad E[y]\ \mathbf{op}\ E[z] = m}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E[x \mapsto m], n+1\rangle}\ \textit{step-arith}$$

[3] I could have used the same → I did above instead of ⇝, but I don't want you to mix them up.

$$\frac{P[n] = \text{goto } m}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E, m\rangle}\ \textit{step-goto}$$

$$\frac{P[n] = \text{if } x\ op_r\ 0 \text{ goto } m \quad E[x]\ \mathbf{op_r}\ 0 = true}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E, m\rangle}\ \textit{step-iftrue}$$

$$\frac{P[n] = \text{if } x\ op_r\ 0 \text{ goto } m \quad E[x]\ \mathbf{op_r}\ 0 = false}{P \vdash \langle E, n\rangle \rightsquigarrow \langle E, n+1\rangle}\ \textit{step-iffalse}$$

1.4 Derivations and provability

Among other things, we can use operational semantics to prove that concrete program expressions will evaluate to particular values. We do this by chaining together rules of inference (which simply list the hypotheses necessary to arrive at a conclusion) into *derivations*, which interlock instances of rules of inference to reach particular conclusions. For example:

$$\frac{\dfrac{\langle 4, E_1\rangle \Downarrow 4 \quad \langle 2, E_1\rangle \Downarrow 2}{\langle 4 * 2, E_1\rangle \Downarrow 8} \qquad \langle 6, E_1\rangle \Downarrow 6}{\langle (4 * 2) - 6, E_1\rangle \Downarrow 2}$$

We say that $\langle e, E\rangle \Downarrow n$ is *provable* (expressed mathematically as $\vdash \langle e, E\rangle \Downarrow n$) if there exists a well-formed derivation with $\langle e, E\rangle \Downarrow n$ as its conclusion. "Well formed" simply means that every step in the derivation is a valid instance of one of the rules of inference for this system.

A proof system like our operational semantics is *complete* if every true statement is provable. It is *sound* (or *consistent*) if every provable judgement is true. Typically, a system of semantics is always complete, unless you forget a rule; soundness can be easier to mess up!

2 Proof techniques using operational semantics

A precise language specification lets us precisely prove properties of our language or programs written in it (and analyses of those programs!). Note that this exposition primarily uses big-step semantics to illustrate, but the concepts generalize.

Well-founded induction. A key family of proof techniques in programming languages is based on *induction*. You may already be familiar with *mathematical induction*. As a reminder: if $P(n)$ is a property of the natural numbers that we want to show holds for all n, mathematical induction says that it suffices to show that $P(0)$ is true (the base case), and then that if $P(m)$ is true, then so is $P(m + 1)$ for any natural number m (the inductive step). This works because there are no infinite descending chains of natural numbers. So, for any n, $P(n)$ can be obtained by simply starting from the base case and applying n instances of the inductive step.

Mathematical induction is a special case of *well-founded induction*, a general, powerful proof principle that works as follows: a relation $\preceq \subseteq A \times A$ is well-founded if there are no infinite descending chains in A. If so, to prove $\forall x \in A.P(x)$ it is enough to prove $\forall x \in A.[\forall y \preceq x \Rightarrow P(y)] \Rightarrow P(x)$.[4]

[4] Mathematical induction as a special case arises when $\preceq$ is simply the predecessor relation $((x, x + 1) | x \in \mathbb{N})$.

Structural induction. *Structural induction* is another special case of well-founded induction where the $\preceq$ relation is defined on the structure of a program or a derivation. For example, consider the syntax of arithmetic expressions in WHILE, `Aexp`. Induction on a recursive definition like this proves a property about a a mathematical structure by demonstrating that the property holds for all possible forms of that structure. We define the relation $a \preceq b$ to hold if a is a substructure of b. For `Aexp` expressions, the relation $\preceq \subseteq \texttt{Aexp} \times \texttt{Aexp}$ is:

$$\begin{gathered} e_1 \preceq e_1 + e_2 \\ e_1 \preceq e_1 * e_2 \\ e_2 \preceq e_1 + e_2 \\ e_2 \preceq e_1 * e_2 \\ \textit{...etc., for all arithmetic operators } op_a \end{gathered}$$

To prove that a property P holds for all arithmetic expressions in WHILE (or, $\forall e \in \texttt{Aexp}.P(e)$), we must show P holds for both the base cases and the inductive cases. e is a base case if there is no e' such that $e' \preceq e$; e is an inductive case if $\exists e'.e' \preceq e$. There is thus one proof case per form of the expression. For `Aexp`, the base cases are:

$$\begin{gathered} \vdash \forall n \in Z.P(n) \\ \vdash \forall x \in \text{Vars}.P(x) \end{gathered}$$

And the inductive cases:

$$\begin{gathered} \vdash \forall e_1, e_2 \in \texttt{Aexp}.P(e_1)P(e_2) \Rightarrow P(e_1 + e_2) \\ \vdash \forall e_1, e_2 \in \texttt{Aexp}.P(e_1)P(e_2) \Rightarrow P(e_1 * e_2) \\ \textit{...and so on for the other arithmetic operators...} \end{gathered}$$

Example. Let $L(e)$ be the number of literals and variable occurrences in some expression e and $O(e)$ be the number of operators in e. Prove by induction on the structure of e that $\forall e \in Aexp.L(e) = O(e) + 1$:

Base cases:

- Case $e = n.L(e) = 1$ and $O(e) = 0$
- Case $e = x.L(e) = 1$ and $O(e) = 0$

Inductive case 1: Case $e = e_1 + e_2$

- By definition, $L(e) = L(e_1) + L(e_2)$ and $O(e) = O(e_1) + O(e_2) + 1$.
- By the induction hypothesis, $L(e_1) = O(e_1) + 1$ and $L(e_2) = O(e_2) + 1$.
- Thus, $L(e) = O(e_1) + O(e_2) + 2 = O(e) + 1$.

The other arithmetic operators follow the same logic.

Other proofs for the expression sublanguages of WHILE can be similarly conducted. For example, we could prove that the small-step and big-step semantics will obtain equivalent results on expressions:

$$\forall e \in \texttt{AExp}.\forall n \in \mathbb{N}.e \rightarrow_a^* n \Leftrightarrow e \Downarrow n$$

The actual proof is left as an exercise, but note that this works because the semantics rules for expressions are strictly syntax-directed: the meaning of an expression is determined entirely by the meaning of its subexpressions, the structure of which guides the induction.

Induction on the structure of derivations. Unfortunately, that last statement is *not* true for *statements* in the WHILE language. For example, imagine we'd like to prove that WHILE is *deterministic* (that is, if a statement terminates, it always evaluates to the same value). More formally, we want to prove that:

$$\forall e \in \texttt{Aexp}.\ \ \forall E \in \mathcal{E}.\forall n, n' \in \mathbb{N}.\ \ \langle e, E\rangle \Downarrow n \wedge \langle e, E\rangle \Downarrow n' \Rightarrow n = n' \tag{1}$$

$$\forall b \in \texttt{Bexp}.\ \ \forall E \in \mathcal{E}.\forall t, t' \in \mathcal{B}.\ \ \langle b, E\rangle \Downarrow t \wedge \langle b, E\rangle \Downarrow b' \Rightarrow b = b' \tag{2}$$

$$\forall s \in S.\ \ \forall E, E', E'' \in \mathcal{E}.\ \ \langle s, E\rangle \Downarrow E' \wedge \langle s, E\rangle \Downarrow E'' \Rightarrow E' = E'' \tag{3}$$

We can't prove the third statement with structural induction on the language syntax because the evaluation of statements (like `while`) does *not* depend only on the evaluation of its subexpressions.

Fortunately, there is another way. Recall that the operational semantics assign meaning to programs by providing rules of inference that allow us to prove judgements by making derivations. Derivation trees (like the expression trees we discussed above) are also defined inductively, and are built of sub-derivations. Because they have structure, we can again use structural induction, but here, on the structure of derivations.

Instead of assuming (and reasoning about) some statement $s \in S$, we instead assume a derivation $D :: \langle s, E\rangle \Downarrow E'$ and induct on the structure of that derivation (we define $D ::$ Judgement to mean "D is the derivation that proves judgement." e.g., $D :: \langle x + 1, E\rangle \Downarrow 2$). That is, to prove that property P holds for a statement, we will prove that P holds for all possible derivations of that statement. Such a proof consists of the following steps:

Base cases: show that P holds for each atomic derivation rule with no premises (of the form $\overline{S}$).

Inductive cases: For each derivation rule of the form

$$\frac{H_1 ... H_n}{S}$$

By the induction hypothesis, P holds for H_i, where $i = 1 \ldots n$. We then have to prove that the property is preserved by the derivation using the given rule of inference.

A key technique for induction on derivations is *inversion.* Because the number of forms of rules of inference is finite, we can tell which inference rules might have been used last in the derivation. For example, given $D :: \langle x := 55, E_i\rangle \Downarrow E$, we know (by inversion) that the assignment rule of inference must be the last rule used in D (because no other rules of inference involve an assignment statement in their concluding judgment). Similarly, if $D :: \langle \text{while } b \text{ do } c, E_i\rangle \Downarrow E$, then (by inversion) the last rule used in D was either the `while-true` rule or the `while-false` rule.

Given those preliminaries, to prove that the evaluation of statements is deterministic (equation (3) above), pick arbitrary s, E, E', and $D :: \langle s, E\rangle \Downarrow E'$

Proof: by induction of the structure of the derivation D, which we define $D :: \langle s, E\rangle \Downarrow E'$.

Base case: the one rule with no premises, `skip`:

$$D :: \overline{\langle \texttt{skip}, E\rangle \Downarrow E}$$

By inversion, the last rule used in D' (which, again, produced E'') must also have been the rule for `skip`. By the structure of the `skip` rule, we know $E'' = E$.

Inductive cases: We need to show that the property holds when the last rule used in D was each of the possible non-skip WHILE commands. I will show you one representative case; the rest are left as an exercise. If the last rule used was the `while-true` statement:

$$D :: \frac{D_1 :: \langle b, E \rangle \Downarrow \texttt{true} \quad D_2 :: \langle s, E \rangle \Downarrow E_1 \quad D_3 :: \langle \texttt{while}\ b\ \texttt{do}\ S, E \rangle \Downarrow E'}{\langle \texttt{while}\ b\ \texttt{do}\ s, E \rangle \Downarrow E'}$$

Pick arbitrary E'' such that $D'' :: \langle \texttt{while}\ b\ \texttt{do}\ S, E \rangle \Downarrow E''$

By inversion, and determinism of boolean expressions, D'' must also use the same `while-true` rule. So D'' must also have subderivations $D_2'' :: \langle s, E \rangle \Downarrow E_1''$ and $D_3'' :: \langle \texttt{while}\ b\ \texttt{do}\ s, E_1'' \rangle \Downarrow E''$. By the induction hypothesis on D_2 with D_2'', we know $E_1 = E_1''$. Using this result and the induction hypothesis on D_3 with D_3'', we have $E'' = E'$.