Lecture Notes: Small-Step Operational Semantics

17-363/17-663: Programming Language Pragmatics (Fall 2021)
Jonathan Aldrich
jonathan.aldrich@cs.cmu.edu

1 Operational Semantics

As we saw in the previous lecture, big-step semantics has some advantages: it intuitively captures the structure of a simple interpreter for the language. However, it has some drawbacks as well. Since big-step semantics produces a value from a program in a single "big step," it says little about the intermediate states the program might go through, and it cannot even model a program that does not terminate. For programs like servers that run indefinitely, what happens while the program is running is the main thing of interest.

To reason about intermediate program states, we can instead define a small-step operational semantics, which models how a program executes one step at a time. For example, imagine the reduction of expressions such as ((1+2)+3)+(4+5). In a big-step semantics, this expression would evaluate to the value 15 in one step; the derivation tree would include 4 different addition operations. In constrast, a small-step semantics would break each addition operation into a separate step. The reduction would go in steps that iteratively transform the expression, as follows:

$$\begin{aligned}((1+2)+3)+(4+5)\\ \rightarrow (3+3)+(4+5)\\ \rightarrow 6+(4+5)\\ \rightarrow 6+9\\ \rightarrow 15\end{aligned}$$

In this example, each step evalutes one addition operation somewhere in the abstract syntax tree representing the expression; there are 4 additions, so there are four steps in the reduction.

In a small-step semantics, a possibly infinite sequence of such steps constitutes the whole run of the program. In between steps, we model the state of the program with a program configuration. For today, we'll model the program configuration as an AST representation of the executing program; later, we may add other elements, such as a store that models the contents of memory or an environment that models the contents of variables on the stack. We'll model execution by changing the program, so that the program in the program configuration always expresses what is left to be done. Our steps are thus rewritings from one program to another, starting with the source program and ending with the value produced by the program–if execution ends at all, that is.

We'll use the following syntax for a simple language with constructs similar to those found in ML (but without static types) or JavaScript:

$$e \quad ::= \quad x \mid n \mid e + e \mid \text{let } x = e \text{ in } e \mid x \Rightarrow e \mid e(e)$$

This is similar to the language we used in the previous class, but a little bit simplified. n represents a concrete number. We've modeled JavaScript's const declaration with a let statement which is a bit more convenient since it is an expression, and so we don't have to model statements separately. We also include JavaScript-style arrow functions instead of using the more heaviweight version that requires statements in the body.

Let's define the semantics of the language in a small-step style, starting with arithmetic expressions. We define the judgment $e_1 \to e_2$ to mean "Expression e_1 takes a single step, resulting in expression e_2." We can define a rule for addition as follows:

$$\frac{n_1 + n_2 = n_3}{n_1 + n_2 \to n_3}\ \textit{step-plus}$$

This rule applies to expressions that add two concrete numbers, n_1 and n_2, reducing the expression with a number n_3 that is the sum of the two original numbers. The premise expresses that we use the semantics of the plus operator from mathematics (e.g. as formalized two lectures ago) to determine n_3.

Of course, an addition expression may be adding expressions that are not numbers, but have interesting structure of their own. In that case we must first reduce those subexpressions to a value, then do the addition. Unlike with big-step semantics, however, we don't want to reduce a big expression down to a value all at once. Instead, if the left subexpression isn't a value, we'll take a single step to work on reducing it to one:

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2}\ \textit{congruence-plus-left}$$

This is a congruence rule, meaning that it doesn't change the plus expression itself, but it represents a step that changes the left argument of plus. Here's an example of using that rule (twice) in a derivation of a single step for the expression ((1+2)+3)+(4+5):

$$\frac{\dfrac{\dfrac{1 + 2 = 3}{1+2 \to 3}\ \textit{step-plus}}{(1 + 2) + 3 \to 3 + 3}\ \textit{congruence-plus-left}}{((1 + 2) + 3) + (4 + 5) \to (3 + 3) + (4 + 5)}\ \textit{congruence-plus-left}$$

Once the left hand side has reduced to a value, we can apply another congruence rule:

$$\frac{e_1 \text{ value} \quad e_2 \to e_2'}{e_1 + e_2 \to e_1 + e_2'}\ \textit{congruence-plus-right}$$

This second congruence rule will be applied over and over again until the right hand side of the plus is a number, at which point we can actually perform the addition. Here we appeal to a separate judgment to determine that e_1 is a value. There are two rules in that judgment, stating that numbers and functions are values:

$$\frac{}{n \text{ value}}\ \textit{value-number} \qquad \frac{}{x \Rightarrow e \text{ value}}\ \textit{value-function}$$

We could add more such rules if we had other kinds of values, such as booleans, strings, or object references. Here is a derivation for a step further along in the execution of our expression. Let's assume the expression has been reduced in two steps to $6 + (4 + 5)$. The next step is:

$$\dfrac{\dfrac{}{6 \text{ value}}\ \textit{value-number} \quad \dfrac{4{+}5 = 9}{4{+}5 \to 9}\ \textit{step-plus}}{6 + (4 + 5) \to 6 + 9}\ \textit{congruence-plus-right}$$

Now we can formalize a complete execution of a program with a transitive judgment $e \to^* e'$, read "e reduces to e' in zero or more steps." We can define this judgment formally with two rules:

$$\dfrac{}{e \to^* e}\ \textit{multi-reflexive} \qquad \dfrac{e \to e' \quad e' \to^* e''}{e \to^* e''}\ \textit{multi-inductive}$$

With these definitions we could create a derivation for a complete execution. The whole tree is large, so I'll just show the parts defininging transitive reduction with $\to^*$:

$$\dfrac{\dfrac{\dfrac{\dots}{((1 + 2) + 3) + (4 + 5) \to (3 + 3) + (4 + 5)} \quad \dfrac{\dfrac{\dots}{(3 + 3) + (4 + 5) \to 6 + (4 + 5)} \quad \dfrac{\dfrac{\dots}{6 + (4 + 5) \to 6 + 9} \quad \dfrac{\dfrac{\dots}{6 + 9 \to 15} \quad \dfrac{}{15 \to^* 15}}{6 + 9 \to^* 15}}{6 + (4 + 5) \to^* 15}}{(3 + 3) + (4 + 5) \to^* 15}}{((1 + 2) + 3) + (4 + 5) \to^* 15}\ \textit{multi-inductive}$$

Let's now define small-step execution rules for the other constructs in the language:

$$\dfrac{e_1 \to e_1'}{\text{let } x = e_1 \text{ in } e_2 \to \text{let } x = e_1' \text{ in } e_2}\ \textit{congruence-let}$$

$$\dfrac{e_1 \text{ value}}{\text{let } x = e_1 \text{ in } e_2 \to [e_1/x]e_2}\ \textit{step-let}$$

$$\dfrac{e_1 \to e_1'}{e_1(e_2) \to e_1'(e_2)}\ \textit{congruence-call-fn}$$

$$\dfrac{e_1 \text{ value} \quad e_2 \to e_2'}{e_1(e_2) \to e_1(e_2')}\ \textit{congruence-call-arg}$$

$$\dfrac{e_2 \text{ value}}{(x \Rightarrow e)(e_2) \to [e_2/x]e}\ \textit{step-call}$$

The let statement evaluates e_1 to a value one step at a time using multiple applications of the congruence rule. When it is a value, the *step-let* rule applies, and we substitute the value for the variable in the body of the let statement. Function calls have two congruence rules, because we must first evaluate the function down to a value, then the argument. When both are values, *step-call* replaces x with the argument value in the function body. The next small step will begin to evaluate the function body.

Exercise 1. Show the derivation for a single step of execution of the following program:
$1 + (x \Rightarrow x + 2)(3)$

Exercise 2. To get some additional practice about how substitution works, show the derivations for the first three single steps of execution for the following program:

let $x = 1$ in (let $y =$ (let $x = 3$ in x) in $x + y$)

Based on our semantics, we can prove theorems about execution. For example, we'd like to show that our rules are deterministic.

Theorem [Evaluation is Deterministic]: If $e \to e'$ and $e \to e''$, then $e' = e''$.

Proof: by induction on the derivation of $e' \to e''$. We case analyze on the rule used. We'll show the two cases for the let rules:

Case $\dfrac{e_1 \text{ value}}{\text{let } x = e_1 \text{ in } e_2 \to [e_1/x]e_2}$ *step-let*:

We now case analyze on the rule used in $e \to e''$. Since e is of the form let $x = e_1$ in e_2, there are only two possibilities.

If the rule *step-let* is used, e'' will be $[e_1/x]e_2$ which is the same as e', completing the subcase.

The other possibility is *congruence-let*. But if we case analyze on the derivation of e_1 value, we find that e_1 is either a number or a function, and there is no evaluation rule for either that would make up the premise of *congruence-let*. Thus this subcase is impossible.

Case $\dfrac{e_1 \to e_1'}{\text{let } x = e_1 \text{ in } e_2 \to \text{let } x = e_1' \text{ in } e_2}$ *congruence-let*:

Again, we now case analyze on the rule used in $e \to e''$. Since e is of the form let $x = e_1$ in e_2, there are only two possibilities.

If the rule *congruence-let* is used, then by the induction hypothesis we know that $e_1 \to e_1'$ in the subderivation of $e \to e''$. Thus e'' will be let $x = e_1'$ in e_2 which is the same as e', completing the subcase.

The other possibility is *step-let*. But if we case analyze on the derivation of $e_1 \to e_1'$, we discover that e_1 does not have either a number or a function form in either of them, and it must have one of those two forms to construct the required premise e_1 value. Thus this subcase is impossible.