Structural Operational Semantics

CPSC 509: Programming Language Principles

Ronald Garcia[*]

4 February 2013

So far we have been defining the semantics of our programming languages using big-step semantics. This approach has some very nice properties:

1. Compared to using recursive functions, we have more flexibility for modeling languages with programs that produce no observable results.
2. The inversion lemmas on the inductive rules set up the skeleton of an interpreter for the language. If you are already comfortable with writing interpreters, you can just about read the big-step rules (bottom-up) as though they were the interpreter written in a compact and stylized notation.
3. Compared to an actual implementation, we can use derivations of big-steps to reason about particular programs (e.g. what does this program evaluate to), and also about classes of programs (e.g. does a certain program transformation always produce equivalent programs?).

However, big-step semantics have some shortcomings. In particular, if you consider the big-step relation $\Downarrow$, it's really just a set of program/result pairs, and tells us nothing about the *process* of computation. This is useful at a first pass, but if we want to reason about programs that terminate, but crash, or programs that never terminate with a final answer, but do useful things while they run, then we need to model the progression of a computation. In this lecture we introduce a particular kind of *small-step* semantics called *structural operational semantics* as our way of capturing steps of computation [Plotkin, 2004b].

1 The big picture

Let's consider again the Boolean and Arithmetic Language.

$$\begin{array}{rcl}
\multicolumn{3}{c}{t \in \textsc{Term}, \quad v \in \textsc{Value}, \quad nv \in \textsc{Num}} \\
t & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \\
& \mid & \mathsf{z} \mid \mathsf{succ}(t) \mid \mathsf{pred}(t) \mid \mathsf{zero?}(t) \\
v & ::= & \mathsf{true} \mid \mathsf{false} \mid nv \\
nv & ::= & \mathsf{z} \mid \mathsf{succ}(nv)
\end{array}$$

To define its semantics, we introduced a big-step relation $\Downarrow \subseteq \textsc{Term} \times \textsc{Value}$, and then defined our evaluator as a partial function in terms of this.

$$\begin{array}{l}
eval_{bs} : \textsc{Term} \rightharpoonup \textsc{Value} \\
eval_{bs}(t) = v \text{ iff } t \Downarrow v.
\end{array}$$

Our goal here is to define the *same* evaluator *eval*, but do it in a way that lets us reason about steps of computation. We do this as follows:

[*]© Ronald Garcia.

1. Define a relation $\longrightarrow \subseteq \textsc{Term} \times \textsc{Term}$ that represents a *single step* of computation. It's up to us to determine what counts as a single step of computation, and our choice may vary depending on our goals.

2. Use this relation to define a *multi-step* relation $\longrightarrow^* \subseteq \textsc{Term} \times \textsc{Term}$, which represents taking 0 or more steps of computation.

3. Observe that $\textsc{Value} \subseteq \textsc{Term}$ in this semantics, so we can consider programs evaluating to completion as instances of $t \longrightarrow^* v$, that is, TERMs that multi-step all the way to a VALUE. This gives us a new definition of our evaluator.

$$eval_{ss} : \textsc{Term} \rightharpoonup \textsc{Value}$$
$$eval_{ss}(t) = v \text{ iff } t \longrightarrow^* v.$$

Now one of our criteria for success here is to be sure that we have indeed defined *the same* semantics (i.e., evaluator) for our language, which we are obligated to establish.

Proposition 1. $eval_{bs} = eval_{ss}$.

To understand the meaning of the above statement, remember that a partial function is just a kind of binary relation, and a binary relation in turn is just a set of pairs. So we have to prove that both sets have *exactly* the same pairs in them, i.e.

$$\forall \langle t, v\rangle \in \textsc{Term} \times \textsc{Value}.\ \langle t, v\rangle \in eval_{bs} \text{ iff } \langle t, v\rangle \in eval_{ss}.$$

2 Small-step Semantics of BA

So to develop our small-step semantics, we need to establish a notion of what counts as a "small-step". We can look to our big-step semantics for some guidance.

First, consider the following big-step derivation:

$$\frac{}{\mathsf{z} \Downarrow \mathsf{z}}$$

Shall we count that as a step of computation? Maybe not: it doesn't seem like this big-step did anything interesting. Nothing happened. On the other hand, in the derivation:

$$\frac{\frac{}{\mathsf{z} \Downarrow \mathsf{z}}}{\mathsf{zero?}(\mathsf{z}) \Downarrow \mathsf{true}}$$

Something interesting happens: the zero? operator examines it's argument and decides that it is indeed zero, and yields true in response. Again, the evaluation of z is not very interesting.

Taking these ideas together, we will claim that the above computation counts as a single step, i.e.

$$\text{szero?-z}\frac{}{\mathsf{zero?}(\mathsf{z}) \longrightarrow \mathsf{true}}.$$

Now consider the slightly more complicated evaluation

$$\frac{\frac{\frac{}{\mathsf{z} \Downarrow \mathsf{z}}}{\mathsf{zero?}(\mathsf{z}) \Downarrow \mathsf{true}}}{\mathsf{if\ zero?}(\mathsf{z})\ \mathsf{then\ false\ else\ true} \Downarrow \mathsf{false}}$$

This computation includes the previous computation, but does more, so we would expect this to count as multiple steps of computation. If we consider zero?(z) to be computed in one step, then in considering the whole program, the if expression is considered to have taken a step if its predicate position has taken a step. Stating this formally:

$$\text{(sif)}\frac{t_1 \longrightarrow t_1'}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \longrightarrow \mathsf{if}\ t_1'\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3}$$

After this step, the if itself can be resolved if its predicate is true. Stated formally:

$$(\text{sif-t})\ \frac{}{\textsf{if true then } t_2 \textsf{ else } t_3 \longrightarrow t_2}$$

We can apply this perspective to all of the big-step rules that we gave for the Boolean and Arithmetic language. What we end up with is the following definition of the small-step (or single-step) relation:

$$(\text{sif})\ \frac{t_1 \longrightarrow t_1'}{\textsf{if } t_1 \textsf{ then } t_2 \textsf{ else } t_3 \longrightarrow \textsf{if } t_1' \textsf{ then } t_2 \textsf{ else } t_3}$$

$$(\text{sif-t})\ \frac{}{\textsf{if true then } t_2 \textsf{ else } t_3 \longrightarrow t_2} \qquad (\text{sif-f})\ \frac{}{\textsf{if false then } t_2 \textsf{ else } t_3 \longrightarrow t_3}$$

$$(\text{ssucc})\ \frac{t \longrightarrow t'}{\textsf{succ}(t) \longrightarrow \textsf{succ}(t')}$$

$$(\text{spred})\ \frac{t \longrightarrow t'}{\textsf{pred}(t) \longrightarrow \textsf{pred}(t')} \qquad (\text{spred-s})\ \frac{}{\textsf{pred}(\textsf{succ}(nv)) \longrightarrow nv} \qquad (\text{szero?})\ \frac{t \longrightarrow t'}{\textsf{zero?}(t) \longrightarrow \textsf{zero?}(t')}$$

$$(\text{szero?-z})\ \frac{}{\textsf{zero?}(\textsf{z}) \longrightarrow \textsf{true}} \qquad (\text{szero?-s})\ \frac{}{\textsf{zero?}(\textsf{succ}(nv)) \longrightarrow \textsf{false}}$$

Furthermore, we can describe the evaluation of our program with respect to the steps of computation that happen:

$$\begin{aligned} & \textsf{if zero?(z) then false else true} \\ \longrightarrow\ & \textsf{if true then false else true} \\ \longrightarrow\ & \textsf{false} \end{aligned}$$

That is to say: our program evaluates in two steps according to our model.

Now that we have our notion of "steps of computation", we need to tie them together into "zero or more steps". We define the $\longrightarrow^*$ multi-step relation also using inductive rules:

$$(\text{incl})\ \frac{t_1 \longrightarrow t_2}{t_1 \longrightarrow^* t_2} \qquad (\text{refl})\ \frac{}{t \longrightarrow^* t} \qquad (\text{trans})\ \frac{t_1 \longrightarrow^* t_2 \quad t_2 \longrightarrow^* t_3}{t_1 \longrightarrow^* t_3}$$

The (incl) rule just says that a single step of computation counts as a multi-step. Note that the premise is really a side-condition on the rule, so we could rewrite it as follows:

$$(\text{incl})\ \frac{}{t_1 \longrightarrow^* t_2} \quad \text{where } t_1 \longrightarrow t_2.$$

This distinction is significant when it's time to explicitly spell out the principle of induction that you get on these derivations.

Finally, the (trans) case simply says that you can paste together two multi-steps that meet in the middle.

These three rules formally suggest that $\longrightarrow^*$ is the *reflexive-transitive closure* of $\longrightarrow$.

Typically an S.O.S. semantics does not bother to present the definition of $\longrightarrow^*$, because it is always essentially the same: the reflexive-transitive closure of whatever single-step relation $\longrightarrow$ is defined.

3 Frames: a simplifying abstraction

As we noticed, a number of the rules in the small-step semantics don't really capture interesting computations, they just facilitate computation inside of other terms. Consider the single step evaluation of the following program:

$$\begin{aligned} & \textsf{if zero?(z) then false else true} \\ \longrightarrow\ & \textsf{if true then false else true} \\ \longrightarrow\ & \textsf{false} \end{aligned}$$

These steps are computed in terms of two different derivation trees:

$$\frac{\dfrac{}{\textsf{zero?}(\textsf{z}) \longrightarrow \textsf{true}}}{\textsf{if zero?}(\textsf{z}) \textsf{ then false else true} \longrightarrow \textsf{if true then false else true}}$$

and

$$\frac{}{\textsf{if true then false else true} \longrightarrow \textsf{false}}$$

The first of these trees uses two rules, (sif) and (szero?-z) to justify the step, but these two rules are fundamentally different.

The (szero?-z) rule encompasses a substantive computational step:

$$\text{szero?-z}\,\frac{}{\textsf{zero?}(\textsf{z}) \longrightarrow \textsf{true}}\,.$$

However, (sif) in general serves just to find the spot in the program where an interesting computation will happen:

$$\text{(sif)}\,\frac{t_1 \longrightarrow t_1'}{\textsf{if } t_1 \textsf{ then } t_2 \textsf{ else } t_3 \longrightarrow \textsf{if } t_1' \textsf{ then } t_2 \textsf{ else } t_3}$$

It basically says that if we can do interesting work in the predicate position of the if expression, then do so.

To make the definition of languages a bit more concise, we distinguish between the interesting and uninteresting rules. In particular, we keep the rules that perform interesting computations (like szero?-z), but we replace the rules that simply point to places in the program with a representation of "position in a program". We call this representation a *frame*, which is a reference to the idea of a "stack frame" that shows up in compilers literature.

Consider the (sif) rule again. For all practical purposes, it says "look at the predicate position of the if." We can capture that more explicitly with a bit of notation:

$$\textsf{if } \square \textsf{ then } t_1 \textsf{ else } t_2$$

We've marked the predicate position with a *hole*, and what we have is not a program anymore, but a simple expression with a hole in it, which we'll call a *context frame*, because what it is doing is describing the context around a place where an interesting computation might happen.

However, we want to be able to build up these contexts for every possible position in a program that might be the next place where a step of computation happens. We can describe all of these places simply by looking at the uninteresting rules in the structural operational semantics, which we'll call the *structural rules*, and figure out what the corresponding context frame is:

rule	**context frame**
(sif)	$\textsf{if } \square \textsf{ then } t_1 \textsf{ else } t_2$
(ssucc)	$\textsf{succ}(\square)$
(spred)	$\textsf{pred}(\square)$
(szero?)	$\textsf{zero?}(\square)$

This leads us to define the set of frames:

$$f \in \textsc{Frame}$$
$$f ::= \textsf{if } \square \textsf{ then } t_1 \textsf{ else } t_2 \mid \textsf{succ}(\square) \mid \textsf{pred}(\square) \mid \textsf{zero?}(\square)$$

Plugging Now that we have a representation of an expression with a hole in it, we need a way to plug that hole. Intuitively, if I have a frame f, and I plug its hole with a term t, then the result should be a full-fledged term. We formalize this by introducing a function for plugging, whose fancy notation is $f[t]$, which stands for plugging the term t into f's hole.

$$\begin{aligned}
\cdot[\cdot] : \textsc{Frame} \times \textsc{Term} &\to \textsc{Term} \\
(\mathsf{if}\ \square\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)[t] &= \mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \\
(\mathsf{succ}(\square))[t] &= \mathsf{succ}(t) \\
(\mathsf{pred}(\square))[t] &= \mathsf{pred}(t) \\
(\mathsf{zero?}(\square))[t] &= \mathsf{zero?}(t)
\end{aligned}$$

Armed with this function, we can replace all of the structural rules from our semantics with a single rule:

$$\frac{t \longrightarrow t'}{f[t] \longrightarrow f[t']}\ (\text{sf})$$

Note that this rule makes explicit use of the plug function as part of its definition. We often use functions in the definition of rules, and they should be viewed as side conditions on their results. For example, a desugared version of this rule is as follows:

$$(\text{sf})\frac{t_1' \longrightarrow t_2'}{t_1 \longrightarrow t_2} \quad \exists f \in \textsc{Frame}.\, t_1 = f[t_1'] \wedge t_2 = f[t_2']$$

So you see, the rule is constrained by a side condition that ensures that t_1' is a subpart of t_1, t_2' is a subpart of t_2 and t_1 and t_2 are related by a common frame.

Typically a frame-style S.O.S. semantics does not bother to explicitly define the plug function since it is trivial and always essentially the same. For this reason, this presentation can be quite concise.

References

G. D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004a. doi: 10.1016/j.jlap.2004.03.009. URL `http://dx.doi.org/10.1016/j.jlap.2004.03.009`.

G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004b.