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 modelling languages with programs whose semantics cannot be simply described by induction over the structure of the programs, or which produce nondeterministic 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. Technically the interpreter embodies the *derivation search* process, but many big-step semantics immediately suggest a proof search strategy.

3. Compared to an actual implementation, we can use big-step derivations 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. Although the *derivations* allude to a process, they have no intrinsic notion of the sequence in which steps of computation might happen.

In this lecture we introduce a particular kind of *small-step* semantics called *structural operational semantics* as our way of explicitly modelling steps of computation [Plotkin, 2004b].

1 The big picture

Let's consider again the Boolean Language.

$$\begin{array}{lcl} t \in \textsc{Term}, \quad v \in \textsc{Value} \\ t & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \\ v & ::= & \mathsf{true} \mid \mathsf{false} \end{array}$$

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

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

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:

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, $\textsc{Term}$s that multi-step all the way to a $\textsc{Value}$. This gives us a new definition of our evaluator.

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

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 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:

$$\frac{}{\text{true} \Downarrow \text{true}}$$

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:

$$\frac{\dfrac{}{\text{true} \Downarrow \text{true}} \quad \dfrac{}{\text{false} \Downarrow \text{false}}}{\text{if true then false else true} \Downarrow \text{false}}$$

Something interesting happens: the if operator examines it's predicate and decides that it is indeed true, and yields the consequent false in response. Again, the evaluation of true is not very interesting, and neither is the evaluation of false, because they are both values so *eValuate* to themselves.

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

$$\text{if true then false else true} \longrightarrow \text{false}.$$

Let's generalize this step to account for arbitrary consequent and alternative expressions and codify it as a rule.

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

Now consider this more complicated evaluation

$$\frac{\dfrac{\dfrac{}{\text{true} \Downarrow \text{true}} \quad \dfrac{}{\text{false} \Downarrow \text{false}}}{\text{if true then false else true} \Downarrow \text{false}} \quad \dfrac{}{\text{true} \Downarrow \text{true}}}{\text{if (if true then false else true) then false else true} \Downarrow \text{true}}$$

This computation includes the previous computation within the predicate position of the outer if expression, but does more, so we would expect this evaluation to count as multiple steps of computation. If we consider

if true then false else true to be computed in one step, then in considering the whole program, the outer if expression is considered to have taken a step if its predicate position has taken a step. So we want it to be true that

$$\text{if } (\text{if true then false else true}) \text{ then false else true} \longrightarrow \text{if false then false else true}$$

We can generalize this to a rule about single-stepping a conditional expression where the predicate can take a step:

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

After the first step, the resulting if itself can be resolved if its predicate is false:

$$\text{if false then false else true} \longrightarrow \text{true}$$

We generalize this as well to a new stepping rule:

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

Since the Boolean language is tiny, these three rules suffice for all programs in the language. Here are the rules, collected, for the single-step relation:

$\boxed{\longrightarrow \subseteq \text{TERM} \times \text{TERM}}$

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

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

And here are the two derivation trees corresponding to the two steps of single-step reduction:

$$\dfrac{\dfrac{}{\text{if true then false else true} \longrightarrow \text{false}} \text{ (sif-t)}}{\text{if } (\text{if true then false else true}) \text{ then false else true} \longrightarrow \text{if false then false else true}} \text{ (sif)}$$

$$\frac{}{\text{if false then false else true} \longrightarrow \text{true}} \text{ (sif-f)}$$

When simply describing the single-step evaluation of a program, we may write the terms in sequence without proofs:

$$\begin{aligned} &\text{if } (\text{if true then false else true}) \text{ then false else true} \\ \longrightarrow\ &\text{if false then false else true} \\ \longrightarrow\ &\text{true} \end{aligned}$$

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

Now that we have our notion of "single 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: $\boxed{\longrightarrow^* \subseteq \text{TERM} \times \text{TERM}}$

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

The (incl) rule, short for "include single-steps" 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 to emphasize that:

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

This distinction between side-conditions and premises is significant when it's time to explicitly spell out the principle of induction that you can derive from these rules.

The (refl) rule, short for "reflexive" says that a vacuous computation, which produces the same output as its input, also counts as a multi-step. This case is used to account for programs that are themselves values, and so require no further computation: they *evaluate* to themselves.

Finally, the (trans) rule, short for "transitive", simply says that you can paste together two multi-steps that meet at some common term t_2 to produce one aggregate multi-step.

Generally speaking, a binary relation $R \subseteq A \times A$ is *reflexive* if $a\ R\ a$ for all $a \in A$. The forward reasoning principle induced by the (refl) rule explicitly asserts that multi-step is reflexive. A binary relation is *transitive* if $a_1 R a_2$ and $a_2 R a_3$ implies $a_1 R a_3$. The forward reasoning principle induced by the (trans) rule explicitly asserts that multi-step is transitive.

These three rules together imply that $\longrightarrow^*$ is what is called the *reflexive-transitive closure* of $\longrightarrow$: the "smallest" binary relation on TERMs that contains $\longrightarrow$ and is also reflexive and transitive (this could be stated more formally and proven).

A typically paper presentation of a structural operational semantics does not bother to explicitly present the definition of $\longrightarrow^*$, because it is always essentially the same relation: the reflexive-transitive closure of whatever single-step relation $\longrightarrow$ is presented. However the principles used for reasoning about structural operational semantics can vary.

3 Many multi-step derivations

As we've seen a number of times so far, we can use derivations to prove that certain pairs of terms are in the multi-step relation. But this relation has a somewhat different property from others we have seen. To see this, consider the following derivation that $\mathsf{true} \longrightarrow^* \mathsf{true}$:

$$\frac{}{\mathsf{true} \longrightarrow^* \mathsf{true}}\ (\text{refl})$$

For most relations we have defined before, there was only one derivation of any given judgment, but not so with multi-step. Here's another derivation of $\mathsf{true} \longrightarrow^* \mathsf{true}$.

$$\dfrac{\dfrac{}{\mathsf{true} \longrightarrow^* \mathsf{true}}\ (\text{refl}) \quad \dfrac{}{\mathsf{true} \longrightarrow^* \mathsf{true}}\ (\text{refl})}{\mathsf{true} \longrightarrow^* \mathsf{true}}\ (\text{trans})$$

and another one:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{true} \longrightarrow^* \mathsf{true}}\ (\text{refl}) \quad \dfrac{}{\mathsf{true} \longrightarrow^* \mathsf{true}}\ (\text{refl})}{\mathsf{true} \longrightarrow^* \mathsf{true}}\ (\text{trans}) \quad \dfrac{}{\mathsf{true} \longrightarrow^* \mathsf{true}}\ (\text{refl})}{\mathsf{true} \longrightarrow^* \mathsf{true}}\ (\text{trans})$$

You might see where this is going. There are an *infinite* number of derivations of $\mathsf{true} \longrightarrow^* \mathsf{true}$! As you imagine, by playing a similar game, we can tell that there are an infinite number of derivations of *any* member of the multi-step relation. This may seem disconcerting, especially since most of them are annoying. But it's useful to get used to the idea that in general an inductive definition can provide many different ways of deducing the same fact.

However, inductive definitions like the ones we have seen so far have a special place.

Definition 1 (Deterministic Inductive Definition)**.** *Let $\mathcal{R}$ be some set of rule instances. Then $\mathcal{R}$ describes a* deterministic *inductive definition if each rule instance in $\mathcal{R}$ has a distinct conclusion.*

If some set has a deterministic inductive definition, then there will be exactly one derivation for each element of the defined set. If you go back and look at our previous inductive definitions, for terms, big-stepping, and single-stepping, you will find that up until now, each has been deterministic. Each deterministic inductive definition leads to a Principle of Function Definition by Recursion for that particular set. So far we have only given such principles for abstract syntax like TERM, but that is because they are the only sets for which it was convenient to use this principle to define functions.

Since our definition of multi-step is non-deterministic, we could not as easily describe a principle of recursion for it. However we *can* provide a different deterministic definition for multi-step:

$\boxed{\longrightarrow \subseteq \text{TERM} \times \text{TERM}}$

$$\frac{}{t \longrightarrow^* t}\ \text{(zero)} \qquad \frac{t_2 \longrightarrow^* t_3}{t_1 \longrightarrow^* t_3}\ \text{(plus-one)} \quad t_1 \longrightarrow t_2.$$

The (zero) rule, as in "zero steps" is exactly the same as the (refl) rule from the earlier definition. The (plus-one) rule, as in "plus one step", is a hybrid between (incl) and (trans).

This definition is equivalent to the earlier one. Proving that it is reflexive (i.e. satisfies the (refl) forward reasoning principle) amounts to just proving the (zero) forward reasoning principle. Proving that it is transitive, on the other hand, requires a substantive proof by induction, whereas it was a simple forward reasoning principle for (trans) in the other definition. This means that our earlier definition makes it easy to splice together two compatible multi-steps. In the other direction, it takes only two forward reasoning steps to prove that the original definition satisfies the forward reasoning principle for (plus-one). Ultimately we could use these *admissibility* propositions to prove that these two definitions in fact define the same set.

On the other hand, proving the proposition $\forall v \in \text{VALUE}.\ \mathsf{true} \longrightarrow^* v \Rightarrow v = \mathsf{true}$ involves only backward reasoning for $\longrightarrow^*$ and $\longrightarrow$, especially that true does not single-step: $\forall t \in \text{TERM}.\ \mathsf{true} \not\longrightarrow t$. To prove the same for the first definition, we must resort to proof by induction, because there are infinitely many derivations of $\mathsf{true} \longrightarrow^* v$ for that definition, and we must consider all infinity of them.

The original definition of multi-step easily demonstrates that the relation is both reflexive and transitive. However the cost is that the definition is non-deterministic, which can affect other reasoning principles, and makes it more involved to define functions over multi-steps. On the other hand, our new definition has exactly one derivation of $\mathsf{true} \longrightarrow^* \mathsf{true}$, and since $\longrightarrow$ is a deterministic *relation*, in the sense that $t \longrightarrow t_1$ and $t \longrightarrow t_2$ implies $t_1 = t_2$, it can be shown that *every* derivation of $t_1 \longrightarrow^* t_2$ has one derivation.[1] Since $\longrightarrow$ is deterministic, there can only be at most one string of steps (i.e. "path") that goes from any term t_1 to t_n. Thus this definition of $\longrightarrow^*$ is a deterministic *inductive definition*, so we can state a Principle of Definition by Recursion for multi-stepping according to the new definition.

Proposition 2. *Let S be a set, $H_{zero} : \text{TERM} \to S$ be a function and $H_{plus-one} : \text{TERM} \times \text{TERM} \times S \to S$ be a function. Then there is a unique function $F : (\longrightarrow^*) \to S$ such that*

1. *$F(t, t) = H_{zero}(t)$; and*
2. *$F(t_1, t_3) = H_{plus-one}(t_1, t_2, F(t_2, t_3))$ if $t_1 \longrightarrow t_2$.*

This recursion principle looks a bit different than the one we defined for TERM, especially because the structure of derivations does not match the syntactic structure of pairs of TERM that multi-step. In contrast, our derivations of $r \in Term$ mirror the structure of the terms that they define. So here we can see that the structure of functions follows the structure of derivations, not necessarily the structure of the judgments that the derivations justify.

The most natural first example of a function defined using this principle is one that counts the number of steps involved in a multi-step relation.

$$\begin{aligned} \mathit{steps} &: (\longrightarrow^*) \to \mathbb{N} \\ \mathit{steps}(t, t) &= 0 \\ \mathit{steps}(t_1, t_3) &= 1 + \mathit{steps}(t_2, t_3) \text{ if } t_1 \longrightarrow t_2 \end{aligned}$$

From this function we can show that

$$\mathit{steps}(\mathsf{if}\ (\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{true})\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{true}, \mathsf{true}) = 2.$$

A second function defined over multi-step collects the terms that arise during multi-stepping.

$$\begin{aligned} \mathit{collect} &: (\longrightarrow^*) \to \mathcal{P}(\text{TERM}) \\ \mathit{collect}(t, t) &= \{\, t \,\} \\ \mathit{collect}(t_1, t_3) &= \{\, t_1 \,\} \cup \mathit{collect}(t_2, t_3) \text{ if } t_1 \longrightarrow t_2 \end{aligned}$$

[1] it's inductive definition is also deterministic, but that's a different property

From this function we can show that

$$collect(\text{if } (\text{if true then false else true}) \text{ then false else true}, \text{true}) = \left\{ \begin{array}{l} \text{if } (\text{if true then false else true}) \text{ then false else true}, \\ \text{if false then false else true}, \\ \text{true} \end{array} \right\}.$$

Exercise 1. What are the components S, H_{zero}, and $H_{plus-one}$ that correspond to these definitions?

4 Frames: a simplifying abstraction

In our definition of the single-step relation, there seem to be two kinds of rules. Rules like (sif) don't really capture interesting computation: they just facilitate computation over some small term inside of some larger terms. In particular, the (sif) in general serves just to find the spot in the program where an interesting computation will happen:

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

It basically says that if we can do interesting work in the predicate position of the if expression, then do so. A larger programming language, with more features, would have more of these "structural" rules, which describe those subexpressions of a term that can be stepped.

The really interesting computations happen in those rules that have no premises, namely (sif-t) and (sif-f).

To make the definition of larger languages a bit more concise, we distinguish between the interesting and uninteresting rules. In particular, we keep the rules that perform interesting computations, like (sif-t), but we replace the myriad of structural rules that simply point to places in the program with a syntax for representing "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:

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

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.

For instance, suppose we added a short-circuiting conjunction to the language

$$t ::= \ldots \mid t \text{ and } t$$

And gave it some evaluation rules:

$$\frac{t_1 \longrightarrow t_1}{t_1 \text{ and } t_2 \longrightarrow t_1' \text{ and } t_2} \text{ (sand)} \qquad \frac{}{\text{false and } t_2 \longrightarrow \text{false}} \text{ (sand-f)} \qquad \frac{}{\text{true and } t_2 \longrightarrow t_2} \text{ (sand-t)}$$

Then we could represent the (sand) rule using the frame $\square$ and t which says "you can evaluate the first argument." This leads us to the following correspondence between structural rules and frames.

rule	context frame
(sif)	if $\square$ then t_1 else t_2
(sand)	$\square$ and t

This leads us to define the set of frames:

$$f \in \textsc{Frame}$$
$$f ::= \textsf{if } \square \textsf{ then } t_1 \textsf{ else } t_2 \mid \square \textsf{ and } t$$

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.

$$\cdot[\cdot] : \textsc{Frame} \times \textsc{Term} \to \textsc{Term}$$
$$(\textsf{if } \square \textsf{ then } t_1 \textsf{ else } t_2)[t] = \textsf{if } t \textsf{ then } t_1 \textsf{ else } t_2$$
$$(\square \textsf{ and } t_2)[t] = t \textsf{ and } t_2$$

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 structural operational 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.