Big-step semantics

Stefan Muller

CS 5095: Types and Programming Languages, Fall 2025
Lecture 21

Let's consider again STLC with general recursion (or, if you prefer, the system PCF of partial computable functions plus sum and product types). We'll also make values (a subset of expressions) explicit in the syntax, which we haven't done before.

$$\begin{array}{llll} \textit{Expressions} & e & ::= & v \mid x \mid e\ e \mid (e,e) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \{x.e; y.e\} \mid \mathsf{fix}\ x = e\ \mathsf{xif} \\ \textit{Types} & \tau & ::= & \mathsf{unit} \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \\ \textit{Values} & v & ::= & () \mid \lambda x : \tau.e \mid (v,v) \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \end{array}$$

Today we're going to consider a new form of evaluation judgment, $e \Downarrow v$ which says that e "evaluates to" the value v. This is kind of like saying $e \mapsto^* v$ (the two will turn out to be equivalent, though we have to prove that), but it ignores all the intermediate steps.

Here are the rules for the judgment:

$$\frac{}{v \Downarrow v}\ (\textsc{EvalV}) \qquad \frac{e_1 \Downarrow \lambda x : \tau.e \qquad e_2 \Downarrow v \qquad [v/x]e \Downarrow v'}{e_1\ e_2 \Downarrow v'}\ (\textsc{EvalApp}) \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(e_1, e_2) \Downarrow (v_1, v_2)}\ (\textsc{EvalPair})$$

$$\frac{e \Downarrow (v_1, v_2)}{\mathsf{fst}\ e \Downarrow v_1}\ (\textsc{EvalFst}) \qquad \frac{e \Downarrow (v_1, v_2)}{\mathsf{snd}\ e \Downarrow v_2}\ (\textsc{EvalSnd}) \qquad \frac{e \Downarrow v}{\mathsf{inl}\ e \Downarrow \mathsf{inl}\ v}\ (\textsc{EvalInL}) \qquad \frac{e \Downarrow v}{\mathsf{inr}\ e \Downarrow \mathsf{inr}\ v}\ (\textsc{EvalInR})$$

$$\frac{e_1 \Downarrow \mathsf{inl}\ v \qquad [v/x]e_2 \Downarrow v'}{\mathsf{case}\ e_1\ \mathsf{of}\ \{x.e_2; y.e_3\} \Downarrow v'}\ (\textsc{EvalCaseL}) \qquad \frac{e_1 \Downarrow \mathsf{inr}\ v \qquad [v/y]e_3 \Downarrow v'}{\mathsf{case}\ e_1\ \mathsf{of}\ \{x.e_2; y.e_3\} \Downarrow v'}\ (\textsc{EvalCaseR})$$

$$\frac{[\mathsf{fix}\ x = e\ \mathsf{xif}/x]e \Downarrow v}{\mathsf{fix}\ x = e\ \mathsf{xif} \Downarrow v}\ (\textsc{EvalFix})$$

One thing to note is that there are a lot fewer rules, since the search rules are folded into the rules that perform the operations like beta reductions. With small-step semantics, an evaluation of a program took many steps and each step was justified (implicitly) by a full derivation tree with usually many search rules. Here, an entire evaluation of a program is one derivation using these "big-step" rules (so called because they evaluate a program in one big step).

$$\dfrac{\dfrac{\dfrac{\dfrac{}{((),()) \Downarrow ((),())}\ (\textsc{EvalVal})}{\mathsf{fst}\ ((),()) \Downarrow ()}\ (\textsc{EvalFst})}{\mathsf{inr}\ (\mathsf{fst}\ ((),())) \Downarrow \mathsf{inr}\ ()}\ (\textsc{EvalInR}) \quad \dfrac{\dfrac{}{\mathsf{inr}\ () \Downarrow \mathsf{inr}\ ()}\ (\textsc{EvalVal}) \qquad \dfrac{}{() \Downarrow ()}\ (\textsc{EvalVal})}{\mathsf{case}\ \mathsf{inr}\ ()\ \mathsf{of}\ \{a.a\ (); b.b\} \Downarrow ()}\ (\textsc{EvalCase})}{(\lambda x : (\mathsf{unit} \to \mathsf{unit}) + \mathsf{unit}.\mathsf{case}\ x\ \mathsf{of}\ \{a.a\ (); b.b\})\ (\mathsf{inr}\ (\mathsf{fst}\ ((),()))) \Downarrow ()}\ (\textsc{EvalApp})$$

$$\dfrac{}{(\lambda x : (\mathsf{unit} \to \mathsf{unit}) + \mathsf{unit}.\mathsf{case}\ x\ \mathsf{of}\ \{a.a\ (); b.b\}) \Downarrow (\lambda x : (\mathsf{unit} \to \mathsf{unit}) + \mathsf{unit}.\mathsf{case}\ x\ \mathsf{of}\ \{a.a\ (); b.b\})}\ (\textsc{EvalVal})$$

We still want to prove type safety, but of course the Progress and Preservation lemmas that we've used throughout the semester assume a small-step semantics. It's not too hard to figure out what the analog of Preservation for a big-step semantics should be.

Lemma 1 (Preservation)**.** *If* $\bullet \vdash e : \tau$ *and* $e \Downarrow v$ *then* $\bullet \vdash v : \tau$.

Proof. By induction on the derivation of $e \Downarrow v$.

- EvalV. By assumption.
- EvalApp. Then $e = e_1\ e_2$ and $e_1 \Downarrow \lambda x : \tau_1.e_0$ and $e_2 \Downarrow v'$ and $[v'/x]e_0 \Downarrow v$. By inversion, we have $\tau = \tau_2$ and $\bullet \vdash e_1 : \tau_1 \to \tau_2$ and $\bullet \vdash e_2 : \tau_1$. By induction, $\bullet \vdash \lambda x : \tau_1.e_0 : \tau_1 \to \tau_2$ and $\bullet \vdash v' : \tau_1$. By inversion, $x : \tau_1 \vdash e_0 : \tau_2$. By substitution, $\bullet \vdash [v'/x]e_0 : \tau_2$. By induction, $\bullet \vdash v : \tau_2$.
- EvalPair. Then $e = (e_1, e_2)$ and $v = (v_1, v_2)$. and $e_1 \Downarrow v_1$ and $e_2 \Downarrow v_2$. By inversion, $\tau = \tau_1 \times \tau_2$ and $\bullet \vdash e_1 : \tau_1$ and $\bullet \vdash e_2 : \tau_2$. By induction, $\bullet \vdash v_1 : \tau_1$ and $\bullet \vdash v_2 : \tau_2$. Apply $\times$-I.
- EvalFst. Then $e = \mathsf{fst}\ e_0$ and $e_0 \Downarrow (v, v_2)$. By inversion, $\bullet \vdash e_0 : \tau \times \tau_2$. By induction, $\bullet \vdash (v, v_2) : \tau \times \tau_2$. By inversion, $\bullet \vdash v : \tau$.
- EvalSnd. Similar.
- EvalInL. Then $e = \mathsf{inl}\ e_0$ and $v = \mathsf{inl}\ v_0$ and $e_0 \Downarrow v_0$. By inversion, $\tau = \tau_1 + \tau_2$ and $\bullet \vdash e_0 : \tau_1$. By induction, $\bullet \vdash v_0 : \tau_1$. Apply $+ - I_1$
- EvalInR. Similar.
- EvalCaseL. Then $e = \mathsf{case}\ e_1\ \mathsf{of}\ \{x.e_2; y.e_3\}$ and $e_1 \Downarrow \mathsf{inl}\ v'$ and $[v'/x]e_2 \Downarrow v$. By inversion, $\bullet \vdash e_1 : \tau_1 + \tau_2$ and $x : \tau_1 \vdash e_2 : \tau$. By induction, $\bullet \vdash \mathsf{inl}\ v' : \tau_1 + \tau_2$. By inversion, $\bullet \vdash v' : \tau_1$. By substitution, $\bullet \vdash [v'/x]e_2 : \tau$. By induction, $\bullet \vdash v : \tau$.
- EvalCaseR. Similar.
- EvalFix. Then $e = \mathsf{fix}\ x = e_0\ \mathsf{xif}$ and $[\mathsf{fix}\ x = e_0\ \mathsf{xif}/x]e_0 \Downarrow v$. By inversion, $x : \tau \vdash e_0 : \tau$. By substitution, $\bullet \vdash [\mathsf{fix}\ x = e_0\ \mathsf{xif}/x]e_0 : \tau$. By induction, $\bullet \vdash v : \tau$.

□

The analog of Progress is less clear. We could say that if $\bullet \vdash e : \tau$ then there exists v such that $e \Downarrow v$. This is true in STLC (we've shown it if you accept that $e \mapsto^* v$ implies $e \Downarrow v$), but it's not true in most real programming languages, and it's not true in the language we're considering above. For example, if we let $e = \mathsf{fix}\ x = x\ \mathsf{xif}$, then there is no v such that $e \Downarrow v$ even though evaluation will never get stuck. In other words, if a program doesn't terminate, big-step semantics doesn't say anything about it. And it's pretty hard to have a real discussion about type safety using big-step semantics.[1]

On the other hand, big-step semantics have a few advantages. First of all, there are a lot fewer rules and we abstract away details like evaluation order.

Here's the theorem we've motivated a few times:

Theorem 1. $e \mapsto^* v$ *if and only if* $e \Downarrow v$.

We won't prove this in these notes (see the textbooks/internet if you want to see the proof) but here's a general outline:

- For the forward direction (If $e \mapsto^* v$ then $e \Downarrow v$), it suffices to show the following lemma sometimes called "closure under converse evaluation", which we can prove by induction on the derivation of $e \mapsto e'$:

  Lemma 2. *If* $e \mapsto e'$ *and* $e' \Downarrow v$*, then* $e \Downarrow v$.

  We then just apply this lemma for each step in the evaluation of $e \mapsto^* v$, going backward from $v \mapsto^0 v$.

[1] A more promising but more complicated direction would be to construct a separate big-step semantics that says that a program reaches a stuck state and show that a well-typed program never reaches such a state. This can work but is a little cumbersome.

- The reverse direction is fairly straightforward but we need a bunch of annoying lemmas that let us "glue" together evaluations using $\mapsto^*$ (these are the same kinds of lemmas we used but didn't actually show when proving normalization of STLC). For example, for EvalApp, induction gives us that $e_1 \mapsto^* \lambda x : \tau.e.$ and $e_2 \mapsto^* v$. We can append the appropriate search rules onto these evaluations to get $e_1\ e_2 \mapsto^* (\lambda x : \tau.e)\ e_2$ and $(\lambda x : \tau.e)\ e_2 \mapsto^* (\lambda x : \tau.e)\ v$, then glue those sequences of steps together. We then have a beta reduction to show $(\lambda x : \tau.e)\ v \mapsto [v/x]e$ which, by induction on the last premise of EvalApp, steps to v'.