Lecture Notes on Operational Dynamic Semantics

15-814: Types and Programming Languages
Jan Hoffmann

Lecture 15
Thursday, October 22, 2026

1 Introduction

In the past weeks, we have defined the dynamic semantics using *structural operations semantics (SOS)*. However, there are many different options for defining the dynamic semantics. We usually classify dynamic semantics into *denotational* and *operational* semantics. A denotational semantics assigns mathematical objects to expressions. For example, the denotational semantics of a function could be a mathematical function. An operational semantics describes how an expression or program is evaluated. There are also other forms of dynamic semantics such as axiomatic semantics and game semantics.

In this course, we focus on operational semantics, which is the most popular class of dynamic semantics. The structural operational semantics we have seen so far is an operational semantics that is based on a transition system. The states of the transition system are (closed) expressions and the transitions form the steps for the evaluation. Another term for SOS is small-step semantics.

In this lecture, we introduce two other types of operational semantics: evaluation dynamics and an abstract machine with control stacks (the *K machine*). Other terms for evaluation dynamics are natural semantics and big-step semantics. Unlike SOS, evaluation dynamics does not have a notion of step. Instead, it inductively defines a relation between expressions and values. In contrast, an abstract machine is based on a transition system and steps like SOS. The difference is that states are not expressions but other objects that often contain expressions.

There does not exist a dynamic semantics that is the best choice in all scenarios. The right choice depends on the programming language we are working with and the goals we want to accomplish with the dynamics. So far, we have had two main goals for our dynamics: defining and conveying how programs are evaluated and proving type soundness. SOS is a great choice for these goals and works for a wide range of languages. However, some features like continuations are difficult to express in SOS.

In some cases, it is beneficial to introduce more than one dynamics so that we can pick the best dynamics for a certain task, for instance a proof. In these cases, we want to show that the different dynamic semantics are equivalent.

2 Call-By-Value PCF

In this lecture, we use *call-by-value* PCF. Recall the syntax and static semantics that we previously defined in lecture.

e	::=	x	variable
		$\mathtt{z}$	zero
		$\mathtt{s}(e)$	successor
		$\mathtt{ifz}\{e_0; x.e_1\}(e)$	conditional
		$\mathtt{fun}\{\tau_1; \tau_2\}(f.x.e)$	recursive function
		$\mathtt{ap}(e_1; e_2)$	function application

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ T_{\text{var}} \qquad \frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}}\ T_{\mathtt{z}} \qquad \frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{s}(e) : \mathtt{nat}}\ T_{\mathtt{s}}$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \mathtt{nat} \vdash e_1 : \tau}{\Gamma \vdash \mathtt{ifz}\{e_0; x.e_1\}(e) : \tau}\ T_{\text{ifz}}$$

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{fun}\{\tau_1; \tau_2\}(f.x.e) : \tau_1 \to \tau_2}\ T_{\text{fun}} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{ap}(e_1; e_2) : \tau'}\ T_{\text{ap}}$$

Values The values of call-by-value PCF are defined as follows. This definition is the same for all the dynamic semantics we consider.

$$\frac{}{\mathtt{z}\ \mathsf{val}}\ V_{\text{zero}} \qquad \frac{e\ \mathsf{val}}{\mathtt{s}(e)\ \mathsf{val}}\ V_{\text{succ}} \qquad \frac{}{\mathtt{fun}\{\tau_1; \tau_2\}(f.x.e)\ \mathsf{val}}\ V_{\text{fun}}$$

Structural Operational Semantics The structural operational semantics is inductively defined as follows.

$$\frac{e \longmapsto e'}{\texttt{ifz}\{e_0; x.e_1\}(e) \longmapsto \texttt{ifz}\{e_0; x.e_1\}(e')}\ S_{\text{ifz1}} \qquad \frac{}{\texttt{ifz}\{e_0; x.e_1\}(\texttt{z}) \longmapsto e_0}\ S_{\text{ifz2}}$$

$$\frac{e\ \mathsf{val}}{\texttt{ifz}\{e_0; x.e_1\}(\texttt{s}(e)) \longmapsto [e/x]e_1}\ S_{\text{ifz3}} \qquad \frac{e_1 \longmapsto e_1'}{\texttt{ap}(e_1; e_2) \longmapsto \texttt{ap}(e_1'; e_2)}\ S_{\text{ap1}}$$

$$\frac{e_1\ \mathsf{val} \quad e_2 \longmapsto e_2'}{\texttt{ap}(e_1; e_2) \longmapsto \texttt{ap}(e_1; e_2')}\ S_{\text{ap2}}$$

$$\frac{e_2\ \mathsf{val}}{\texttt{ap}(\texttt{fun}\{\tau_1; \tau_2\}(f.x.e); e_2) \longmapsto [\texttt{fun}\{\tau_1; \tau_2\}(f.x.e), e_2/f, x]e}\ S_{\text{ap3}}$$

3 Evaluation Dynamics

The first alternative to SOS that we consider is evaluation dynamics. An application of the evaluation dynamics of call-by-value PCF is the proof that SOS and the K machine are equivalent; a statement that we make precise later. As it turns out, instead of proving the statement directly, it is much easier to prove the equivalence of evaluation dynamics and SOS and the equivalence of the evaluation dynamics and the K machine.

Another motivation for the evaluation dynamics is that we would like to have an operational semantics that can be directly translated to the implementation of an efficient interpreter. This is not the case for SOS, in which we have to apply *search rules* such as S_{ap1} again and again to find the spot in a (large) expression at which we should apply one of the leaf rules like S_{ap3}.

A natural and more efficient way to evaluating an application $\texttt{ap}(e_1; e_2)$ is the following. First evaluate e_1 to a value $\texttt{fun}\{\tau_1; \tau_2\}(f.x.e)$, then evaluate e_2 to a value v_2, and finally evaluate $[\texttt{fun}\{\tau_1; \tau_2\}(f.x.e), v_2/f, x]e$ to the value that is the result of the application. This leads directly to a recursive implementation that corresponds to the inductive definition of the evaluation dynamics. Note that we only relate expressions to values in the description of the evaluation strategy without introducing a notion of steps.

The judgment

$$e \Downarrow v$$

of the evaluation dynamics states that expression e evaluates to value v. It is inductively defined by the following rules.

$$\frac{}{\mathtt{z} \Downarrow \mathtt{z}}\ E_{\mathrm{z}} \qquad \frac{e \Downarrow v}{\mathtt{s}(e) \Downarrow \mathtt{s}(v)}\ E_{\mathrm{s}} \qquad \frac{}{\mathtt{fun}\{\tau_1;\tau_2\}(f.x.e) \Downarrow \mathtt{fun}\{\tau_1;\tau_2\}(f.x.e)}\ E_{\mathrm{fun}}$$

$$\frac{e \Downarrow \mathtt{z} \quad e_0 \Downarrow v}{\mathtt{ifz}\{e_0; x.e_1\}(e) \Downarrow v}\ E_{\mathrm{ifz1}} \qquad \frac{e \Downarrow \mathtt{s}(v') \quad v\ \mathsf{val}' \quad [v'/x]e_1 \Downarrow v}{\mathtt{ifz}\{e_0; x.e_1\}(e) \Downarrow v}\ E_{\mathrm{ifz2}}$$

$$\frac{e_1 \Downarrow \mathtt{fun}\{\tau_1;\tau_2\}(f.x.e) \quad e_2 \Downarrow v_2 \quad [\mathtt{fun}\{\tau_1;\tau_2\}(f.x.e), v_2/f, x]e \Downarrow v}{\mathtt{ap}(e_1; e_2) \Downarrow v}\ E_{\mathrm{ap}}$$

We can prove the following theorems by induction on the judgment $e \Downarrow v$.

Theorem 1 *If $e : \tau$ and $e \Downarrow v$ then $v : \tau$.*

Theorem 2 *If $e : \tau$ and $e \Downarrow v$ then v val.*

Relation to SOS Now that we have defined another operational semantics, we want to related it to SOS. We can show that the two dynamic semantics are equivalent as formalized by the following theorem. As usual, we identify α-equivalent expressions.

Theorem 3 *Let $e : \tau$ be a program. Then $e \Downarrow v$ if and only if $e \longmapsto^* v$ and v val.*

The direction from left to right is called soundness and can be proved by induction on $e \Downarrow v$.

Lemma 4 (Soundness) *Let $e : \tau$ be a program. If $e \Downarrow v$ then $e \longmapsto^* v$.*

The direction from right to left is called completeness and is more difficult to prove. We could aim to prove the statement by induction on the number of steps n in the evaluation $e \longmapsto^n v$. However, in the induction step, it is not immediately clear how to conclude $e \Downarrow v$ from the premises $e \longmapsto e'$ and $e' \Downarrow v$. Luckily, we can prove the following lemma by induction on the stepping relation.

Lemma 5 *Let $e : \tau$. If $e \longmapsto e'$ and $e' \Downarrow v$ then $e \Downarrow v$.*

Divergence The evaluation dynamics has some advantages over SOS. For instance, the structure of the inference rules is similar to the structure of the type rules because the premises refer to the subexpressions of the expression in the conclusion. This can simplify certain proofs. It is also closer to the implementation of an efficient interpreter, particularly if we replace substitution with a value environment in which we map variables to values.

A major shortcoming of the evaluation dynamics is that it cannot distinguish computations that get stuck from computations that do not terminate. In both cases, there is simply no value v such that $e \Downarrow v$. This is why the evaluation dynamics cannot be used to show type soundness.

It is possible to define a variant of the evaluation dynamics that addresses this shortcoming. We define the judgment

$$e \Downarrow^n v$$

which includes a natural number n that reflects the size of the derivation tree (or the cost of the evaluation). The existing rules are altered as in the following example.

$$\frac{e_1 \Downarrow^{n_1} \mathtt{fun}\{\tau_1;\tau_2\}(f.x.e) \quad e_2 \Downarrow^{n_2} v_2 \quad [\mathtt{fun}\{\tau_1;\tau_2\}(f.x.e), v_2/f, x]e \Downarrow^{n} v}{\mathtt{ap}(e_1;e_2) \Downarrow^{n_1+n_2+n+1} v}\ E_{\text{ap}}$$

Then we add an *abort* rule that can be applied to any expression.

$$\frac{}{e \Downarrow^0 \circ}\ E_{\text{abort}}$$

An aborted computation $e \Downarrow^n \circ$ corresponds to a partial derivation tree of size n. If a part of the derivation is aborted, then the complete derivation is aborted. This is reflected by adding rules such as the following.

$$\frac{e_1 \Downarrow^{n_1} \mathtt{fun}\{\tau_1;\tau_2\}(f.x.e) \quad e_2 \Downarrow^{n_2} \circ}{\mathtt{ap}(e_1;e_2) \Downarrow^{n_1+n_2+1} \circ}\ E_{\text{ap-abort}}$$

Type Safety With this new judgment we can formulate and prove a theorem that is equivalent to progress and preservation.

Theorem 6 (Type Safety) *If $e : \tau$ then either $e \Downarrow^n v$ for some n or $e \Downarrow^m \circ$ for all $m \in \mathbb{N}$.*

4 Control Stacks (K Machine)

In this section, we introduce the *K machine*, an abstract machine for call-by-value PCF that is based on control stacks. Control stacks are used to define the dynamics of continuations and simplify the dynamics of other control constructs, such as exceptions. Like SOS, the K machine is a transition system and evaluation is a sequence of transitions between states. However, states of the K machine consist of a control stack and an expression.

One way to motivate control stacks, is to view them as a technique to avoid *search rules* like the ones we use in SOS to find the spot in an expression at which the evaluation makes progress. This is achieved by splitting the expression into a part that we are currently working on (the expression of the state) and the surrounding expression (the control stack).

We start by defining the states of the transition system. They have two forms.

$K \triangleright e$	evaluate expression e on stack K
$K \triangleleft e$	return value e to stack K

Stacks K are lists of frames defined as follows.

$$\begin{array}{rcl} K & ::= & \epsilon \\ & & K; f \end{array}$$

The definition of frames f depends on the language we consider and the evaluation order. For call-by-value PCF, the frames are defined by the following grammar.

$$\begin{array}{rcl} f & ::= & \mathtt{s}(-) \\ & & \mathtt{ifz}\{e_0; x.e_1\}(-) \\ & & \mathtt{ap}(e_1; -) \\ & & \mathtt{ap}(-; e_2) \end{array}$$

Initial and final states of the transition system are defined by the following rules.

$$\frac{}{\epsilon \triangleright e \ \mathsf{initial}} \qquad \frac{e \ \mathsf{val}}{\epsilon \triangleleft e \ \mathsf{final}}$$

The following rules define the transitions.

$$\frac{}{K \rhd \mathtt{z} \longmapsto K \lhd \mathtt{z}}\ K_{\mathtt{z}} \qquad \frac{}{K \rhd \mathtt{s}(e) \longmapsto K; \mathtt{s}(-) \rhd e}\ K_{\mathtt{s1}}$$

$$\frac{}{K; \mathtt{s}(-) \lhd e \longmapsto K \lhd \mathtt{s}(e)}\ K_{\mathtt{s2}}$$

$$\frac{}{K \rhd \mathtt{ifz}\{e_0; x.e_1\}(e) \longmapsto K; \mathtt{ifz}\{e_0; x.e_1\}(-) \rhd e}\ K_{\mathtt{if1}}$$

$$\frac{}{K; \mathtt{ifz}\{e_0; x.e_1\}(-) \lhd \mathtt{z} \longmapsto K \rhd e_0}\ K_{\mathtt{if2}}$$

$$\frac{}{K; \mathtt{ifz}\{e_0; x.e_1\}(-) \lhd \mathtt{s}(e) \longmapsto K \rhd [e/x]e_1}\ K_{\mathtt{if3}}$$

$$\frac{}{K \rhd \mathtt{fun}\{\tau_1; \tau_2\}(f.x.e) \longmapsto K \lhd \mathtt{fun}\{\tau_1; \tau_2\}(f.x.e)}\ K_{\mathtt{fun}}$$

$$\frac{}{K \rhd \mathtt{ap}(e_1; e_2) \longmapsto K; \mathtt{ap}(-; e_2) \rhd e_1}\ K_{\mathtt{ap1}}$$

$$\frac{}{K; \mathtt{ap}(-; e_2) \lhd \mathtt{fun}\{\tau_1; \tau_2\}(f.x.e) \longmapsto K; \mathtt{ap}(\mathtt{fun}\{\tau_1; \tau_2\}(f.x.e); -) \rhd e_2}\ K_{\mathtt{ap2}}$$

$$\frac{}{K; \mathtt{ap}(\mathtt{fun}\{\tau_1; \tau_2\}(f.x.e); -) \lhd e_2 \longmapsto K \rhd [\mathtt{fun}\{\tau_1; \tau_2\}(f.x.e), e_2/f, x]e}\ K_{\mathtt{ap3}}$$

Type Safety To formulate progress and preservation, we need to extend the typing rules to program states. One problem is that we get stuck, if in a state $K \lhd e$, the expression e is not compatible with the first frame on the stack K. Consider, for example, $\mathtt{ap}(-; \mathtt{s}(\mathtt{z})) \lhd \mathtt{z}$. The only rule that matches this stack is $K_{\mathtt{ap2}}$. However, to make a transition the expression e has to be a function instead of $\mathtt{z}$. So this is a stuck state for which we cannot make progress.

Another problem is that we can reach stuck states if the frames on the stack are not compatible. Consider, for example,

$$\mathtt{ap}(-; \mathtt{s}(\mathtt{z})); \mathtt{s}(-) \lhd \mathtt{z}\,.$$

The expression $\mathtt{z}$ is compatible with the frame but eventually we will step to the stuck state $\mathtt{ap}(-; \mathtt{s}(\mathtt{z})) \lhd \mathtt{s}(\mathtt{z})$.

To ensure the compatibility of expressions, stacks, and successive frames, we define for each frame an input and an output type. The judgment

$f : \tau_1 \rightsquigarrow \tau_2$ states that frame f accepts a value of type τ_1 and eventually produces a value of type τ_2.

$$\frac{}{\mathtt{s}(-) : \mathtt{nat} \rightsquigarrow \mathtt{nat}}\ F_{\mathrm{s}} \qquad \frac{e_0 : \tau \quad x : \mathtt{nat} \vdash e_1 : \tau}{\mathtt{ifz}\{e_0; x.e_1\}(-) : \mathtt{nat} \rightsquigarrow \tau}\ F_{\mathrm{ifz}}$$

$$\frac{e_1 : \tau' \to \tau \quad e_1\ \mathsf{val}}{\mathtt{ap}(e_1; -) : \tau' \rightsquigarrow \tau}\ F_{\mathrm{ap1}} \qquad \frac{e_2 : \tau'}{\mathtt{ap}(-; e_2) : \tau' \to \tau \rightsquigarrow \tau}\ F_{\mathrm{ap2}}$$

A stack is well-typed and expecting a value of type τ, written as $K \lhd : \tau$, if the input and output types of consecutive frames match and the topmost frame is expecting a value of type τ.

$$\frac{}{\epsilon \lhd : \tau} \qquad \frac{K \lhd : \tau' \quad f : \tau \rightsquigarrow \tau'}{K; f \lhd : \tau}$$

Finally, states are well-typed if they consist of well-typed stacks with compatible well-typed expressions.

$$\frac{e : \tau \quad K \lhd : \tau}{K \rhd e\ \mathsf{ok}} \qquad \frac{e : \tau \quad K \lhd : \tau \quad e\ \mathsf{val}}{K \lhd e\ \mathsf{ok}}$$

Now we can formulate type safety.

Theorem 7 (Progress) *If s ok then either s final or there exists an s' such that $s \longmapsto s'$.*

The proof is by induction on s ok.

Theorem 8 (Preservation) *If s ok and $s \longmapsto s'$ then s' ok.*

The proof is by induction on $s \longmapsto s'$.

Finally, we can also show that the K machine is sound and complete with respect to SOS. Proving soundness is not straightforward but can be done with a familiar induction argument if we set it up cleverly as you will show in the homework. The key idea is to relate a state s of the K machine with an expression $e = \mathit{unroll}(s)$ that is the result of combining the frames of the stack with the focus expression into one expression. We can then show (roughly) that a step $s \longmapsto s'$ in the K machine corresponds to a step $\mathit{unroll}(s) \longmapsto \mathit{unroll}(s')$ in SOS.

However, to prove completeness directly, we would need to define a bisimulation, which is more complex than you might think. An approach

like the one for soundness fails because it is not clear how to convert an expression e into a K machine state s since there are (in general) many possibilities.

We can avoid this technical difficulty by proving the K machine sound and complete with respect to the evaluation dynamics. The soundness and completeness of the K machine with respect to SOS then follows from Theorem 3.

Theorem 9 (Soundness) *If* $\epsilon \triangleright e \longmapsto^* \epsilon \triangleleft v$ *then* $e \Downarrow v$.

The proof of the soundness theorem follows the same recipe as the proof of soundness with respect to SOS that you will work out in the homework.

Theorem 10 (Completeness) *If* $e \Downarrow v$ *then* $K \triangleright e \longmapsto^* K \triangleleft v$ *for every stack* K.

The proof of the completeness theorem proceeds by induction on the judgment $e \Downarrow v$.