Formal Operational Semantics

So far, we’ve presented operational semantics using interpreters: precise and executable, but verbose and too concrete.

One alternative: describe semantics using state transition judgments.

Example language:

\[
\text{exp ::= var | int} \\
\text{| ' ( ' '+' exp exp ' ) ' } \\
\text{| ' ( ' 'let' var exp exp ' ) ' } \\
\text{| ' ( ' ':'= var exp ' ) ' } \\
\text{| ' ( ' 'if' exp exp exp ' ) ' } \\
\text{| ' ( ' 'while' exp exp ' ) ' } \\
\text{| etc.}
\]
State machine transition judgments

\[ \langle e, E, S \rangle \Downarrow \langle v, S' \rangle \]

- **current expression**
- **result value**
- **current environment** maps vars \( x \) to locations \( l \)
- **final store** maps locations \( l \) to values \( v \)
- **initial store** maps locations \( l \) to values \( v \)

Evaluating expression \( e \) in environment \( E \) and store \( S \) yields value \( v \) and (possibly) changed store \( S' \)
Evaluation by inference

- To describe the machine’s operation, we give rules of inference that say when a judgment can be derived from judgments about sub-expressions.

\[
\frac{\text{premises}}{\text{conclusion}} \quad \text{(Name of rule)}
\]

- We can view evaluation of the program as the process of building an inference tree.

- Notation has similarities to axiomatic semantics: idea of derivation is that same, but contents of judgments is different.
ENVIRONMENTS AND STORES, FORMALLY

- We write $E(x)$ means the result of looking up $x$ in environment $E$. (This notation is because an environment is like a function taking a name as argument and returning a meaning as result.)
- We write $E + \{x \mapsto v\}$ for the environment obtained from existing environment $E$ by extending it with a new binding from $x$ to $v$. If $E$ already has a binding for $x$, this new binding replaces it.

The **domain** of an environment, $dom(E)$, is the set of names bound in E.

Analogously with environments, we’ll write
- $S(l)$ to mean the value at location $l$ of store $S$
- $S + \{l \mapsto v\}$ to mean the store obtained from store $S$ by extending (or updating) it so that location $l$ maps to value $v$.
- $dom(S)$ for the set of locations bound in store $S$.

Also, we’ll write
- $S - \{l\}$ to mean the store obtained from store $S$ by removing the binding for location $l$. 
### Evaluation Rules (1)

\[
\begin{align*}
  l &= E(x) \quad v = S(l) \quad \text{(Var)} \\
  \langle x, E, S \rangle &\downarrow \langle v, S \rangle
\end{align*}
\]

\[
\begin{align*}
  \langle i, E, S \rangle &\downarrow \langle i, S \rangle \quad \text{(Int)}
\end{align*}
\]

\[
\begin{align*}
  \langle e_1, E, S \rangle &\downarrow \langle v_1, S' \rangle \quad \langle e_2, E, S' \rangle &\downarrow \langle v_2, S'' \rangle \\
  \langle (+ e_1 e_2), E, S \rangle &\downarrow \langle v_1 + v_2, S'' \rangle \quad \text{(Add)}
\end{align*}
\]

\[
\begin{align*}
  \langle e_1, E, S \rangle &\downarrow \langle v_1, S' \rangle \quad l \notin \text{dom}(S') \\
  \langle e_2, E + \{x \mapsto l\}, S' + \{l \mapsto v_1\} \rangle &\downarrow \langle v_2, S'' \rangle \\
  \langle \text{let } x e_1 e_2, E, S \rangle &\downarrow \langle v_2, S'' - \{l\} \rangle \quad \text{(Let)}
\end{align*}
\]

\[
\begin{align*}
  \langle e, E, S \rangle &\downarrow \langle v, S' \rangle \quad l = E(x) \\
  \langle (:= x e), E, S \rangle &\downarrow \langle v, S' + \{l \mapsto v\} \rangle \quad \text{(Assgn)}
\end{align*}
\]
\[
\begin{align*}
\langle e_1, E, S \rangle \Downarrow \langle v_1, S' \rangle & \quad v_1 \neq 0 \quad \langle e_2, E, S' \rangle \Downarrow \langle v_2, S'' \rangle \\
\langle \text{(if } e_1 \quad e_2 \quad e_3), E, S \rangle \Downarrow \langle v_2, S'' \rangle & \quad \text{(If-nzero)}
\end{align*}
\]

\[
\begin{align*}
\langle e_1, E, S \rangle \Downarrow \langle 0, S'' \rangle & \quad \langle e_3, E, S' \rangle \Downarrow \langle v_3, S''' \rangle \\
\langle \text{(if } e_1 \quad e_2 \quad e_3), E, S \rangle \Downarrow \langle v_3, S''' \rangle & \quad \text{(If-zero)}
\end{align*}
\]

\[
\begin{align*}
\langle e_1, E, S \rangle \Downarrow \langle v_1, S' \rangle & \quad v_1 \neq 0 \quad \langle e_2, E, S' \rangle \Downarrow \langle v_2, S'' \rangle \\
\langle \text{(while } e_1 \quad e_2), E, S' \rangle \Downarrow \langle v_3, S''' \rangle & \quad \langle \text{(while } e_1 \quad e_2), E, S \rangle \Downarrow \langle v_3, S''' \rangle \\
\langle \text{(while } e_1 \quad e_2), E, S \rangle \Downarrow \langle v_3, S''' \rangle & \quad \text{(While-nzero)}
\end{align*}
\]

\[
\begin{align*}
\langle e_1, E, S \rangle \Downarrow \langle 0, S' \rangle & \quad \langle e_2, E, S' \rangle \Downarrow \langle 0, S'' \rangle \\
\langle \text{(while } e_1 \quad e_2), E, S \rangle \Downarrow \langle 0, S'' \rangle & \quad \text{(While-zero)}
\end{align*}
\]
Example Derivation Trees

where \( E_1 = \{ x \mapsto L_1 \} \), \( S_1 = \{ L_1 \mapsto 10 \} \), \( S_2 = \{ L_1 \mapsto 21 \} \).

where \( E_1 = \{ x \mapsto L_1 \} \), \( S_1 = \{ L_1 \mapsto -1 \} \), \( S_2 = \{ L_1 \mapsto 0 \} \).
About the rules

- As usual in inference systems, a rule only applies if all the premises above the line can be shown.

- Structure of rules guarantees that at most one rule is applicable at any point.

- Store relationships constrain the order of evaluation of premises.
  
  (For simplicity here, we use just a single global store)

- If no rules apply, the evaluation gets stuck; this corresponds to an (unchecked) runtime error.
Rules vs. Interpreter

We can view a recursive interpreter as implementing a bottom-up exploration of the inference tree.

A function like

\[
\text{Value eval(Exp } e, \text{ Env env) \{ .... \}}
\]

returns a value \( v \) and has side effects on a global store \( \text{store} \) such that

\[
\langle e, \text{env, store} \rangle \downarrow \langle v, \text{store} \rangle
\]

The implementation of \( \text{eval} \) dispatches on the syntactic form of \( e \), choosing the appropriate rule,

and makes recursive calls on \( \text{eval} \) corresponding to the premises of that rule.

Question: How deep can the derivation tree get?