

CS558 Programming Languages

Winter 2008

Lecture 6

So far, we've presented operational semantics using interpreters. These have the advantage of being **precise** and **executable**. But they are not ideally **compact** or **abstract**.

Another way to present operational semantics is using **state transition judgments**, for appropriately defined machine states.

For example, consider a simple language of imperative expressions, in which variables must be defined before use, using a `local` construct.

```
exp := var | int
     | '(' '+' exp exp ')'
     | '(' 'local' var exp exp ')'
     | '(' ':' var exp ')'
     | '(' 'if' exp exp exp ')'
     | '(' 'while' exp exp ')'
     | etc.
```

Informally, the meaning of `(local x e1 e2)` is: evaluate e_1 to a value v_1 , create a new store location l bound to x and initialized to v_1 , and evaluate e_2 in the resulting environment and store.

STATE MACHINE

To evaluate this language, we choose a machine state consisting of:

- the current **environment** E , which maps each in-scope variable to a location l .
- the current **store** S , which maps each location l to an integer value v .
- the current **expression** e , to be evaluated.

We give the state transitions in the form of **judgments**:

$$\langle e, E, S \rangle \Downarrow \langle v, S' \rangle$$

Intuitively, this says that evaluating expression e in environment E and store S yields the value v and the (possibly) changed store S' .

OPERATIONAL SEMANTICS BY INFERENCE

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

The form of a rule is

$$\frac{\text{premises}}{\text{conclusion}} \text{ (Name of rule)}$$

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

This notation has similarities to axiomatic semantics: the notion of derivation is essentially the same, but the content 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)

$$\frac{l = E(x) \quad v = S(l)}{\langle x, E, S \rangle \Downarrow \langle v, S \rangle} \text{ (Var)}$$

$$\frac{}{\langle i, E, S \rangle \Downarrow \langle i, S \rangle} \text{ (Int)}$$

$$\frac{\langle e_1, E, S \rangle \Downarrow \langle v_1, S' \rangle \quad \langle e_2, E, S' \rangle \Downarrow \langle v_2, S'' \rangle}{\langle (+ \ e_1 \ e_2), E, S \rangle \Downarrow \langle v_1 + v_2, S'' \rangle} \text{ (Add)}$$

$$\frac{\langle e_1, E, S \rangle \Downarrow \langle v_1, S' \rangle \quad l \notin dom(S') \quad \langle e_2, E + \{x \mapsto l\}, S' + \{l \mapsto v_1\} \rangle \Downarrow \langle v_2, S'' \rangle}{\langle (\text{local } x \ e_1 \ e_2), E, S \rangle \Downarrow \langle v_2, S'' - \{l\} \rangle} \text{ (Local)}$$

$$\frac{\langle e, E, S \rangle \Downarrow \langle v, S' \rangle \quad l = E(x)}{\langle (:= \ x \ e), E, S \rangle \Downarrow \langle v, S' + \{l \mapsto v\} \rangle} \text{ (Assgn)}$$

EVALUATION RULES (2)

$$\frac{\langle e_1, E, S \rangle \Downarrow \langle v_1, S' \rangle \quad v_1 \neq 0 \quad \langle e_2, E, S' \rangle \Downarrow \langle v_2, S'' \rangle}{\langle (\text{if } e_1 \ e_2 \ e_3), E, S \rangle \Downarrow \langle v_2, S'' \rangle} \text{ (If-nzero)}$$

$$\frac{\langle e_1, E, S \rangle \Downarrow \langle 0, S' \rangle \quad \langle e_3, E, S' \rangle \Downarrow \langle v_3, S'' \rangle}{\langle (\text{if } e_1 \ e_2 \ e_3), E, S \rangle \Downarrow \langle v_3, S'' \rangle} \text{ (If-zero)}$$

$$\frac{\langle e_1, E, S \rangle \Downarrow \langle v_1, S' \rangle \quad v_1 \neq 0 \quad \langle e_2, E, S' \rangle \Downarrow \langle v_2, S'' \rangle \quad \langle (\text{while } e_1 \ e_2), E, S'' \rangle \Downarrow \langle v_3, S''' \rangle}{\langle (\text{while } e_1 \ e_2), E, S \rangle \Downarrow \langle v_3, S''' \rangle} \text{ (While-nzero)}$$

$$\frac{\langle e_1, E, S \rangle \Downarrow \langle 0, S' \rangle}{\langle (\text{while } e_1 \ e_2), E, S \rangle \Downarrow \langle 0, S' \rangle} \text{ (While-zero)}$$

NOTES ON THE RULES

- The structure of the rules guarantees that at most one rule is applicable at any point.
- The store relationships constrain the order of evaluation.
- If no rules are applicable, the evaluation **gets stuck**; this corresponds to a runtime error in an interpreter.

We can view the interpreter for the language as implementing a bottom-up exploration of the inference tree. A function like

```
Value eval(Exp e, Env env) { ... }
```

returns a value v and has side effects on a global store such that

$$\langle e, env, store_{before} \rangle \Downarrow \langle v, store_{after} \rangle$$

The implementation of `eval` dispatches on the syntactic form of e , chooses the appropriate rule, and makes recursive calls on `eval` corresponding to the premises of that rule.

Question: how deep can the derivation tree get?