



EECS 490 – Lecture 13

Operational Semantics

1

Announcements

- ▶ HW3 due tomorrow at 8pm
- ▶ Project 3 due 10/27 at 8pm

Semantics

- Syntax is concerned with the structure of programs, while **semantics** is concerned with their meaning
- Semantics can be described formally
 - **Denotational semantics**: program behavior described using set and domain theory and partial functions over program state
 - **Axiomatic semantics**: concerned with proving logical assertions over program state, so meaning specified with respect to the effect on these logical assertions
 - **Operational semantics**: specifies what each computational step does to the state of a program, and what value it computes
- We will look at a specific form of operational semantics known as **natural** or **big-step** semantics

The textbook describes *small-step* operational semantics, but it is similar enough to big-step semantics to be a useful resource.

Language

- We will use a simple imperative language:

$P \rightarrow S$

Statements

$S \rightarrow$ skip
 $S \rightarrow S; S$
 $S \rightarrow V = A$
 $S \rightarrow \text{if } B \text{ then } S \text{ else } S \text{ end}$
 $S \rightarrow \text{while } B \text{ do } S \text{ end}$

Arithmetic expressions

$A \rightarrow N$
 $A \rightarrow V$
 $A \rightarrow (A + A)$
 $A \rightarrow (A - A)$
 $A \rightarrow (A * A)$

Boolean expressions

$B \rightarrow \text{true}$
 $B \rightarrow \text{false}$
 $B \rightarrow (A \leq A)$
 $B \rightarrow (B \text{ and } B)$
 $B \rightarrow \text{not } B$

Variables

$V \rightarrow \text{Identifier}$

Numbers

$N \rightarrow \text{Integer}$

States

- The **state** of a program is a mapping of variables to values
- State denoted by σ , and value of variable v is $\sigma(v)$
- A new state with a new mapping for variable v to value n is denoted by $\sigma[v := n]$

- Formally,


$$\sigma[v := n](w) = \begin{cases} n, & \text{if } v = w \\ \sigma(w), & \text{if } v \neq w \end{cases}$$

- In other words, the value of other variables are unchanged, but the value of v is now n

Transitions

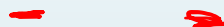
- A transition denotes the result of a computation:

$$\langle s, \sigma \rangle \rightarrow \langle u, \sigma' \rangle$$




- This states that program fragment s , when computed in state σ , yields value u and a new state σ'
- If the fragment does not produce a new state, then the state can be elided from the right-hand side:

$$\langle 3, \sigma \rangle \rightarrow 3$$



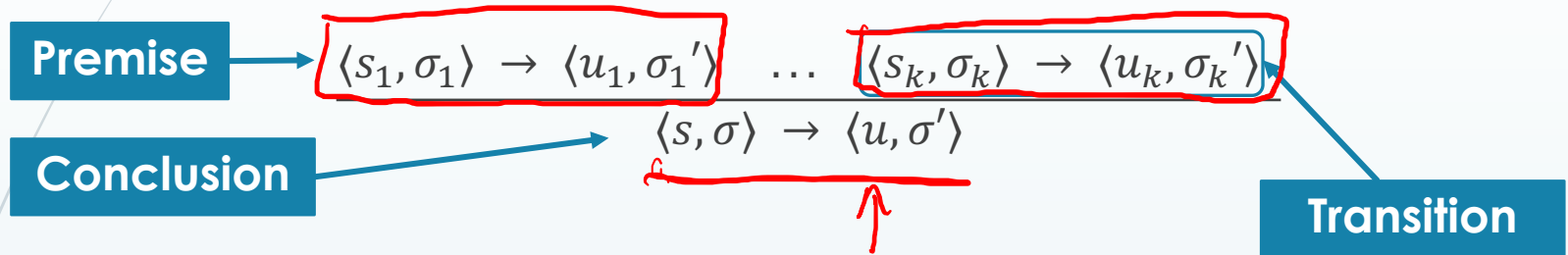
- If the fragment does not produce a value, then the value can be elided from the right-hand side:

$$\langle x = 3, \sigma \rangle \rightarrow \sigma[x := 3]$$



Transition Rules

- Transition rules have the following form:

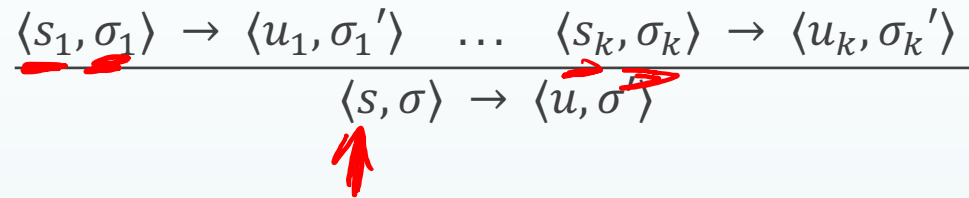


- This is a conditional rule that means:
 - If** s_1 computed in state σ_1 yields value u_1 and modified state σ_1'
 - ...
 - If** s_k computed in state σ_k yields value u_k and modified state σ_k'
 - Then** s computed in state σ yields value u and modified state σ'

In our convention, only transitions can appear in the premises or conclusion of a rule.

Interpretation

- Transition rules have the following form:

$$\frac{\langle s_1, \sigma_1 \rangle \rightarrow \langle u_1, \sigma_1' \rangle \quad \dots \quad \langle s_k, \sigma_k \rangle \rightarrow \langle u_k, \sigma_k' \rangle}{\langle s, \sigma \rangle \rightarrow \langle u, \sigma' \rangle}$$


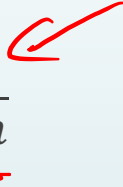
- A transition rule specifies a formula for interpreting a program fragment
 - If the interpreter sees a fragment of the form s , it can compute s by instead computing the fragments s_1, \dots, s_k that are in the premises, in the specified states
 - Computation terminates when no more transition rules can be applied

Expressions

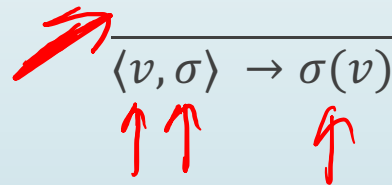
- Expressions do not modify the state in our language, so we will elide the state on the right-hand side of a transition

- Rules with an empty premise are called **axioms**

- Axiom for numbers:

$$\frac{}{\langle n, \sigma \rangle \rightarrow n}$$


- Axiom for variables:

$$\frac{}{\langle v, \sigma \rangle \rightarrow \sigma(v)}$$


Arithmetic Expressions

► Addition:

$$\frac{\langle a_1, \sigma \rangle \rightarrow n_1 \quad \langle a_2, \sigma \rangle \rightarrow n_2}{\langle (a_1 + a_2), \sigma \rangle \rightarrow n}$$

where $n = n_1 + n_2$

► Subtraction:

$$\frac{\langle a_1, \sigma \rangle \rightarrow n_1 \quad \langle a_2, \sigma \rangle \rightarrow n_2}{\langle (a_1 - a_2), \sigma \rangle \rightarrow n}$$

where $n = n_1 - n_2$

► Multiplication:

$$\frac{\langle a_1, \sigma \rangle \rightarrow n_1 \quad \langle a_2, \sigma \rangle \rightarrow n_2}{\langle (a_1 * a_2), \sigma \rangle \rightarrow n}$$

where $n = n_1 \times n_2$

Evaluation

 $\sigma(x)$

- In evaluating a compound expression, the premises are recursively evaluated until axioms are reached

$$\begin{array}{c}
 \frac{\overline{\langle x, \sigma \rangle \rightarrow 1} \quad \overline{\langle 3, \sigma \rangle \rightarrow 3}}{\overline{\langle (x+3), \sigma \rangle \rightarrow 4}} \quad \frac{\overline{\langle y, \sigma \rangle \rightarrow 2} \quad \overline{\langle 5, \sigma \rangle \rightarrow 5}}{\overline{\langle (y-5), \sigma \rangle \rightarrow -3}} \\
 \hline
 \overline{\langle ((x+3) * (y-5)), \sigma \rangle \rightarrow -12}
 \end{array}$$

$a_1 * a_2$

- The result is a **derivation tree**, where the root is the expression to be evaluated and the leaves are axioms
- Evaluation terminates if no more transition rules can be applied, i.e. all leaves are axioms

Example

- Derivation tree for $((x * 3) - 2)$:

Order of Evaluation

- If expressions have side effects, then they produce a new state as well as a new value
- Left-to-right order of evaluation:

$$\frac{\langle a_1, \sigma \rangle \rightarrow \langle n_1, \sigma_1 \rangle \quad \langle a_2, \sigma_1 \rangle \rightarrow \langle n_2, \sigma_2 \rangle}{\langle (a_1 + a_2), \sigma \rangle \rightarrow \langle n, \sigma_2 \rangle} \quad \text{where } n = n_1 + n_2$$

- To specify that either order of evaluation may occur, add the following rule to the above:

$$\frac{\langle a_2, \sigma \rangle \rightarrow \langle n_2, \sigma_2 \rangle \quad \langle a_1, \sigma_2 \rangle \rightarrow \langle n_1, \sigma_1 \rangle}{\langle (a_1 + a_2), \sigma \rangle \rightarrow \langle n, \sigma_1 \rangle} \quad \text{where } n = n_1 + n_2$$

- Implementations can choose which rule to apply for each expression

Boolean Expressions

- True and false:

$$\frac{}{\langle \text{true}, \sigma \rangle \rightarrow \text{true}}$$

$$\frac{}{\langle \text{false}, \sigma \rangle \rightarrow \text{false}}$$

- Comparisons:

$$\frac{\langle a_1, \sigma \rangle \rightarrow n_1 \quad \langle a_2, \sigma \rangle \rightarrow n_2}{\langle (a_1 \leq a_2), \sigma \rangle \rightarrow \text{true}} \quad \text{if } n_1 \leq n_2$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow n_1 \quad \langle a_2, \sigma \rangle \rightarrow n_2}{\langle (a_1 \leq a_2), \sigma \rangle \rightarrow \text{false}} \quad \text{if } n_1 > n_2$$

- Negation:

$$\frac{\langle b, \sigma \rangle \rightarrow \text{true}}{\langle \text{not } b, \sigma \rangle \rightarrow \text{false}}$$

$$\frac{\langle b, \sigma \rangle \rightarrow \text{false}}{\langle \text{not } b, \sigma \rangle \rightarrow \text{true}}$$

Conjunction

- Non-short-circuiting conjunction:

$$\frac{\langle \underline{b_1}, \sigma \rangle \rightarrow t_1 \quad \langle \underline{b_2}, \sigma \rangle \rightarrow t_2}{\langle (\underline{b_1} \text{ and } \underline{b_2}), \sigma \rangle \rightarrow t} \quad \text{where } t = \underline{t_1 \wedge t_2}$$

- Short-circuiting conjunction:

$$\text{⚡} \frac{\langle \underline{b_1}, \sigma \rangle \rightarrow \underline{\text{false}}}{\langle (\underline{b_1} \text{ and } \underline{b_2}), \sigma \rangle \rightarrow \underline{\text{false}}}$$

$$\frac{\langle \underline{b_1}, \sigma \rangle \rightarrow \underline{\text{true}} \quad \langle \underline{b_2}, \sigma \rangle \rightarrow \underline{t_2}}{\langle (\underline{b_1} \text{ and } \underline{b_2}), \sigma \rangle \rightarrow \underline{t_2}}$$

Example

- Derivation tree for $((x \leq 3) \textbf{ and } (0 \leq x))$:

- We'll start again in five minutes.

Statements

- Statements do not have a value in our language, so the right-hand side of a transition will just be the state that results from completely executing a statement
- The **skip** statement does nothing:

$$\frac{}{\langle \text{skip}, \sigma \rangle \rightarrow \sigma}$$

- Assignment produces a new state:

$$\frac{\langle a, \sigma \rangle \rightarrow n}{\langle v = a, \sigma \rangle \rightarrow \sigma[v := n]}$$

Sequencing and Conditionals

- Sequencing executes the second statement in the state produced from executing the first:

$$\frac{\langle s_1, \sigma \rangle \rightarrow \sigma_1 \quad \langle s_2, \sigma_1 \rangle \rightarrow \sigma_2}{\langle s_1; s_2, \sigma \rangle \rightarrow \sigma_2}$$

- Conditionals depend on if the predicate is true or false:

$$\frac{\langle b, \sigma \rangle \rightarrow \text{true} \quad \langle s_1, \sigma \rangle \rightarrow \sigma_1}{\langle \text{if } b \text{ then } s_1 \text{ else } s_2 \text{ end}, \sigma \rangle \rightarrow \sigma_1}$$

$$\frac{\langle b, \sigma \rangle \rightarrow \text{false} \quad \langle s_2, \sigma \rangle \rightarrow \sigma_2}{\langle \text{if } b \text{ then } s_1 \text{ else } s_2 \text{ end}, \sigma \rangle \rightarrow \sigma_2}$$

Loops

- A loop whose predicate is false does nothing:

$$\frac{\langle b, \sigma \rangle \rightarrow \text{false}}{\langle \text{while } b \text{ do } s \text{ end}, \sigma \rangle \rightarrow \sigma}$$

- A loop whose predicate is true is the same as:
 - Executing one iteration of the body
 - Recursively executing the loop in the resulting state

$$\frac{\langle b, \sigma \rangle \rightarrow \text{true} \quad \langle s, \sigma \rangle \rightarrow \sigma_1 \quad \langle \text{while } b \text{ do } s \text{ end}, \sigma_1 \rangle \rightarrow \sigma_2}{\langle \text{while } b \text{ do } s \text{ end}, \sigma \rangle \rightarrow \sigma_2}$$

$\sigma[x := 2][y := 3]$

Example: Loop

```

x = 1;
while x <= 2 do
  x = x + 1
end

```

- Apply transition rule for **while**:

$$\frac{
 \frac{
 \frac{\langle x, \sigma \rangle \rightarrow 1}{\langle x \leq 2, \sigma \rangle \rightarrow \text{true}}
 \quad
 \frac{\langle 2, \sigma \rangle \rightarrow 2}{\langle x = x + 1, \sigma \rangle \rightarrow \sigma[x := 2]}
 }{
 \langle \text{while } x \leq 2 \text{ do } x = x + 1 \text{ end}, \sigma \rangle \rightarrow \sigma'
 }
 \quad
 \frac{
 \frac{\langle x, \sigma \rangle \rightarrow 1}{\langle 1, \sigma \rangle \rightarrow 1}
 \quad
 \frac{\langle x + 1, \sigma \rangle \rightarrow 2}{\langle s_1, \sigma[x := 2] \rangle \rightarrow \sigma'}
 }{
 \langle s_1, \sigma[x := 2] \rangle \rightarrow \sigma'
 }$$

where $s_1 = [\text{while } x \leq 2 \text{ do } x = x + 1 \text{ end}]$

- Recursively executing s_1 in $\sigma[x := 2]$:

$$\frac{
 \langle x \leq 2, \sigma[x := 2] \rangle \rightarrow \text{true} \quad \langle x = x + 1, \sigma[x := 2] \rangle \rightarrow \sigma[x := 3] \quad \langle s_1, \sigma[x := 3] \rangle \rightarrow \sigma'
 }{
 \langle \text{while } x \leq 2 \text{ do } x = x + 1 \text{ end}, \sigma[x := 2] \rangle \rightarrow \sigma'
 }$$

- Another recursive execution of s_1 , now in $\sigma[x := 3]$:

$$\frac{
 \langle x \leq 2, \sigma[x := 3] \rangle \rightarrow \text{false}
 }{
 \langle \text{while } x \leq 2 \text{ do } x = x + 1 \text{ end}, \sigma[x := 3] \rangle \rightarrow \sigma[x := 3]
 }$$

Result: final
state is

$\sigma' = \sigma[x := 3]$

Example: Divergent Loop

```
while true
do
  skip
end
```


- Apply transition rule for **while**:

$$\frac{\overline{\langle \text{true}, \sigma \rangle \rightarrow \text{true}} \quad \overline{\langle \text{skip}, \sigma \rangle \rightarrow \sigma} \quad \langle \text{while true do skip end}, \sigma \rangle \rightarrow \sigma'}{\langle \text{while true do skip end}, \sigma \rangle \rightarrow \sigma'}$$

- Recursive execution of the new loop results in same exact computation as above
- The computation is *divergent*, since it never terminates

Example: Scheme define

- We can define a transition rule that demonstrates the equivalence of two forms of `define` in Scheme:



$$\frac{\langle (\text{define } f \text{ (lambda (params) body)), \sigma \rangle \rightarrow \langle u, \sigma_1 \rangle}{\langle \text{define } (f \text{ params}) \text{ body}, \sigma \rangle \rightarrow \langle u, \sigma_1 \rangle}$$

- This states that the result of evaluating the `define` form in the conclusion is the same as that of evaluating the form in the premise
- An interpreter can translate the form in the conclusion to the form in the premise

Example: Swapping Operands

- We can demonstrate that swapping operands is a legal transformation in our simple language:

$$\frac{\langle x, \sigma \rangle \rightarrow n_x \quad \langle y, \sigma \rangle \rightarrow n_y}{\langle (x + y), \sigma \rangle \rightarrow n} \quad \text{where } n = n_x + n_y$$

$$\frac{\langle y, \sigma \rangle \rightarrow n_y \quad \langle x, \sigma \rangle \rightarrow n_x}{\langle (y + x), \sigma \rangle \rightarrow n} \quad \text{where } n = n_x + n_y$$

By commutativity
of integer addition

Semantics for λ -Calculus

- No state in λ -Calculus, so transitions have the form:

$$e_1 \rightarrow e_2$$

- An expression in normal form evaluates to itself:

$$\frac{}{e \rightarrow e} \quad \text{where } \textit{normal}(e)$$

- We define $\textit{normal}(e)$ as follows:

$$\begin{aligned} \textit{normal}(v) &= \textit{true} \\ \textit{normal}(\lambda v. e) &= \textit{normal}(e) \\ \textit{normal}(v e) &= \textit{true} \\ \textit{normal}((e_1 e_2) e_3) &= \textit{normal}(e_1 e_2) \\ \textit{normal}((\lambda v. e_1) e_2) &= \textit{false} \end{aligned}$$

Semantics for λ -Calculus

- An abstraction is evaluated by reducing its body:

$$\frac{e_1 \rightarrow e_2}{\lambda v. e_1 \rightarrow \lambda v. e_2}$$

- A sequence of function applications is evaluated by computing the first application, followed by the second:

$$\frac{e_1 e_2 \rightarrow e_4 \quad e_4 e_3 \rightarrow e_5}{(e_1 e_2) e_3 \rightarrow e_5}$$

Semantics for λ -Calculus

- An application of an abstraction to an argument does the following:
 - Reduce the body of the abstraction
 - Substitute the argument into the body
 - Evaluate the result of the substitution

$$\frac{e_1 \rightarrow e_3 \quad subst(e_3, v, e_2) \rightarrow e_4}{(\lambda v. e_1) e_2 \rightarrow e_4}$$

- Definition of $subst(body, var, arg)$:

$$\begin{aligned} subst(v_1, v, e) &= \begin{cases} e & \text{if } v = v_1 \\ v_1 & \text{otherwise} \end{cases} \\ subst(\lambda v_1. e_1, v, e) &= \begin{cases} \lambda v_1. e_1 & \text{if } v = v_1 \\ \lambda v_1. subst(e_1, v, e) & \text{otherwise} \end{cases} \\ subst(e_1 e_2, v, e) &= subst(e_1, v, e) subst(e_2, v, e) \end{aligned}$$