

CS 162 Programming languages

Lecture 7: Operational Semantics II

Yu Feng
Winter 2021

What we have by far

- Given a program as an input string
- First, we separate a string into words (Lexer)
- Second, we understand sentence structure by diagramming the string (Parser)
- Finally, we assign meanings to the structure sentence (Operational semantics)

Operational Semantics

$$\frac{}{\mathbb{E} \vdash \text{lambda } x. e \Downarrow \text{lambda } x. e} \text{ LAMBDA}$$

Lambda abstractions just evaluate to themselves

$$\frac{\mathbb{E} \vdash e_1 \Downarrow \text{lambda } x. e'_1 \quad \mathbb{E} \vdash [x \mapsto e_2]e'_1 \Downarrow v}{\mathbb{E} \vdash (e_1 e_2) \Downarrow v} \text{ APP}$$

To evaluate the application $(e_1 e_2)$, we first evaluate the expression e_1 . The operational semantics “**get stuck**” if e_1 is not a lambda abstraction. This notion of “getting stuck” in the operational semantics corresponds to a **runtime error**. Assuming the expression e_1 evaluates to a lambda expression, we evaluate the application expression by binding e_2 to x and then evaluating the expression $[x \mapsto e_2]e'_1$ as in β -reduction in lambda calculus.

The Lambda rule

- Question: What would change if we write the hypothesis as

$$\frac{\mathbb{E} \vdash e_1 \overset{=}{\not\Downarrow} \text{lambda } x. e'_1 \quad \mathbb{E} \vdash [x \mapsto e_2]e'_1 \Downarrow v}{\mathbb{E} \vdash (e_1 \ e_2) \Downarrow v} \text{APP}$$

- **Answer:** This would still give semantics to (lambda x.x 3), but no longer to let y=lambda x.x in (y 3)

The Lambda rule

- Question: What would change if we write the hypothesis as

$$\frac{\mathbb{E} \vdash e_1 \Downarrow \text{lambda } x. e'_1 \quad \mathbb{E} \vdash [x \mapsto \overbrace{e_2}^{v_2}] e'_1 \Downarrow v}{\mathbb{E} \vdash (e_1 \ e_2) \Downarrow v} \text{APP}$$

- **Answer:** This is also correct: you will eagerly evaluate e_2 before passing it to the lambda abstraction (call-by-value)
- You need to implement this semantics for HW3

Call-by-name v.s. call-by-value

- Not evaluating the argument before substitution is known as call-by-name, evaluating the argument before substitution as call-by-value.
- Languages with call-by-name: classic lambda calculus, ALGOL 60
- Languages with call-by-value: C, C++, Java, Python, FORTRAN, . . .
- Advantage of call-by-name: If argument is not used, it will not be evaluated
- Disadvantage: If argument is used k times, it will be evaluated k times!

Operational Semantics

$$\frac{\mathbb{E}(x) = v}{\mathbb{E} \vdash x \Downarrow v} \text{VAR}$$

If variable x is bound to some value v in the environment E , then x evaluates to its bound value.

$$\frac{\mathbb{E} \vdash e_1 \Downarrow v_1 \quad \mathbb{E}[x \mapsto v_1] \vdash e_2 \Downarrow v_2}{\mathbb{E} \vdash \text{let } x = e_1 \text{ in } e_2 \Downarrow v_2} \text{LET}$$

First evaluate the initial expression e_1 in environment E , which yields value v_1 . Then we obtain a new environment E' by binding identifier x to value v_1 , i.e., $E' = E[x \leftarrow v_1]$. Next, we evaluate the body e_2 in this new environment E' , which yields value v_2 , which is also the result of evaluating the entire let expression.

Environments example

$$\frac{\mathbb{E} \vdash e_1 \Downarrow v_1 \quad \mathbb{E}[x \mapsto v_1] \vdash e_2 \Downarrow v_2}{\mathbb{E} \vdash \text{let } x = e_1 \text{ in } e_2 \Downarrow v_2} \text{LET} \qquad \frac{\mathbb{E}(x) = v}{\mathbb{E} \vdash x \Downarrow v} \text{VAR}$$

- Consider the λ^+ program: let $x = 3$ in x
- Here is the proof that this program evaluates to 3:

$$\frac{E \vdash 3 : 3 \quad \frac{E[x \leftarrow 3](x) = 3}{E[x \leftarrow 3] \vdash x : 3}}{E \vdash \text{let } x = 3 \text{ in } x : 3}$$

Operational Semantics

$$\frac{}{\mathbb{E} \vdash \text{Nil} \Downarrow \text{Nil}} \text{NIL} \qquad \frac{\mathbb{E} \vdash e_1 \Downarrow v_1 \quad \mathbb{E} \vdash e_2 \Downarrow v_2}{\mathbb{E} \vdash e_1 @ e_2 \Downarrow v_1 @ v_2} \text{CONS}$$

a list is either the empty list Nil, or it is a cons cell($e_1 @ e_2$) where e_1 is the head of the list and e_2 is the tail of the list.

$$\frac{\mathbb{E} \vdash e \Downarrow \text{Nil}}{\mathbb{E} \vdash \text{isnil } e \Downarrow 1} \text{ISNILTRUE} \qquad \frac{\mathbb{E} \vdash e \Downarrow v_1 @ v_2}{\mathbb{E} \vdash \text{isnil } e \Downarrow 0} \text{ISNILFALSE}$$

Since any list value can either be Nil or a cons cell, we have two cases which rule matches triggered will depend on whether e evaluates to Nil or not. If e is not a list, then the evaluation will **get stuck**.

$$\frac{\mathbb{E} \vdash e \Downarrow v_1 @ v_2}{\mathbb{E} \vdash !e \Downarrow v_1} \text{HEAD} \qquad \frac{\mathbb{E} \vdash e \Downarrow v_1 @ v_2}{\mathbb{E} \vdash \#e \Downarrow v_2} \text{TAIL}$$

We define similar rules for **head** and **tail**

Congratulations!

- You can now understand every page in the λ^+ reference manual
- For HW3, you will need to refer to the operational semantics of λ^+ in the manual to implement your interpreter
- The manual is the official source for the semantics of λ^+

Operational semantics

- The rules we have written are known as **big-step** operational semantics
- They are called big step because each rule completely evaluates an expression, taking *as many steps as necessary*.

- Example: The plus rule
$$\frac{\mathbb{E} \vdash e_1 \Downarrow i_1 \quad \mathbb{E} \vdash e_2 \Downarrow i_2}{\mathbb{E} \vdash e_1 + e_2 \Downarrow i_1 + i_2} \text{ADD}$$

- Here, we evaluate both e_1 and e_2 to compute the final value in one (**big**) step
- Alternate formalism for giving semantics: **small-step** operational semantics

Small step operational semantics

- Small step operational semantics (denoted as “ \rightarrow ”) perform *only one step* of computation *per rule* invocation
- You can think of SSOS as “decomposing” all operations that happen in one rule in LSOS into individual steps
- This means: Each rule in SSOS has at most one precondition

$$t \longrightarrow^* v \text{ iff } t \Downarrow v.$$

Small step operational semantics

- Consider the plus rule in λ^+ written in SSOS
- Rule 1: Adding two integers

$$\overline{\langle c_1 + c_2, E \rangle \rightarrow \langle c_1 + c_2, E \rangle}$$

- Rule 2: Reducing first expression to an integer

$$\frac{\langle e_1, E \rangle \rightarrow \langle c, E' \rangle}{\langle e_1 + e_2, E \rangle \rightarrow \langle c + e_2, E' \rangle}$$

- Rule 3: Reducing second expression to an integer

$$\frac{\langle e, E \rangle \rightarrow \langle c_2, E' \rangle}{\langle c_1 + e, E \rangle \rightarrow \langle c_1 + c_2, E' \rangle}$$

SSOS in action

- Let's use these rules to prove what the value of $(2+4)+(6-1)$ is:
- $\langle (2+4)+(6-1), E \rangle \rightarrow \langle 6+(6-1), E \rangle \rightarrow \langle 6+5, E \rangle \rightarrow \langle 11, E \rangle$

One atomic step at a time!

Small-step v.s. Big-step

- In big-step semantics, any rule may invoke any number of other rules in the hypothesis
- This means any derivation is a **tree**.
- In small-step semantics, each rule only performs one step of computation
- This means any derivation is a **line**

TODOs by next lecture

- HW3 is out. Please start ASAP!
- Will switch to type checking next week