CS 162 Programming languages

Lecture 8: Operational Semantics II

Yu Feng

Winter 2023

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

$$\frac{}{\mathbb{E} \vdash \textsf{lambda}\, x.\ e \Downarrow \textsf{lambda}\, x.\ e}\ \textsc{Lambda}$$

Lambda abstractions just evaluate to themselves

$$\frac{\mathbb{E} \vdash e_1 \Downarrow \textsf{lambda}\, x.\ e_1' \qquad \mathbb{E} \vdash [x \mapsto e_2]e_1' \Downarrow v}{\mathbb{E} \vdash (e_1\ e_2) \Downarrow v}\ \textsc{App}$$

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 $[x \rightarrow e_2]e'_1$ as in β-reduction in lambda calculus.

The Lambda rule

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

$$\frac{\mathbb{E} \vdash e_1 \overset{=}{\cancel{\Downarrow}} \textsf{lambda}\; x.\; e_1' \qquad \mathbb{E} \vdash [x \mapsto e_2]e_1' \Downarrow v}{\mathbb{E} \vdash (e_1\; e_2) \Downarrow v}\ \textsc{App}$$

- 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

$$\frac{\mathbb{E} \vdash e_1 \Downarrow \textsf{lambda}\ x.\ e_1' \qquad e_2 \Downarrow v_2 \qquad \mathbb{E} \vdash [x \mapsto \overset{v_2}{\cancel{e_2}}]e_1' \Downarrow v}{\mathbb{E} \vdash (e_1\ e_2) \Downarrow v}\ \textsc{App}$$

- Answer: This is also correct: you will eagerly evaluate e_2 before passing it to the lambda abstraction (call-by-value)

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 uses k times, it will be evaluated k times!

Booleans: implementation

Boolean implementation

- let TRUE = λx y. x   -- Returns its first argument
- let FALSE = λx y. y   -- Returns its second argument
- let ITE   = λb x y. b x y -- Applies condition to branches

Why they are correct?

Booleans: examples

eval ite_true:

$$
\begin{aligned}
& \text{ITE TRUE } e_1\ e_2 \\
&= (\lambda b\ x\ y.\ b\ \ x\ \ y)\ \text{TRUE}\ \ e_1\ e_2 && \text{-- expand def ITE} \\
&=_\beta\ (\lambda x\ y.\ \text{TRUE}\ x\ \ y)\ \ e_1\ e_2 && \text{-- beta-step} \\
&=_\beta\ (\lambda y.\ \text{TRUE}\ e_1\ y)\ \ e_2 && \text{-- beta-step} \\
&=_\beta\ \text{TRUE}\ e_1\ e_2 && \text{-- expand def TRUE} \\
&=\ (\lambda x\ y.\ \ x)\ e_1\ e_2 && \text{-- beta-step} \\
&=_\beta\ (\lambda y.\ e_1)\ \ e_2 && \text{-- beta-step} \\
&=_\beta\ e_1
\end{aligned}
$$

Other boolean API:

let NOT = λb. ITE b FALSE TRUE

let AND = $\lambda b_1\ b_2.$ ITE $b_1\ b_2$ FALSE

let OR = $\lambda b_1\ b_2.$ ITE b_1 TRUE b_2

λ-calculus:Numbers

- Church numerals: a number N is encoded as a combinator that calls a function on an argument N times

let ONE = $\lambda f\ \lambda x.\ f\ x$

let TWO = $\lambda f\ \lambda x.\ f\ (f\ x)$

let THREE = $\lambda f\ \lambda x.\ f\ (f\ (f\ x))$

let FOUR = $\lambda f\ \lambda x.\ f\ (f\ (f\ (f\ x)))$

let FIVE = $\lambda f\ \lambda x.\ f\ (f\ (f\ (f\ (f\ x))))$

let SIX = $\lambda f\ \lambda x.\ f\ (f\ (f\ (f\ (f\ (f\ x)))))$

let ZERO = $\lambda f\ \lambda x.\ x$

https://en.wikipedia.org/wiki/Church_encoding

λ-calculus:Numbers API

- Numbers API
  - let INC = (λn λf λx. f (n f x)) -- Call \`f\` on \`x\` one more time than \`n\` does
  - let ADD = λn λm. n INC m. -- Call \`f\` on \`x\` exactly \`n + m\` times

eval inc_zero :
INC ZERO
= (λn λf λx. f (n f x)) ZERO
$=_\beta$ λf λx. f (ZERO f x)
= λf λx. f x
= ONE

eval add_one_zero :
ADD ONE ZERO = ONE

https://en.wikipedia.org/wiki/Church_encoding

Recursion

- Recursion can not be directly applied with β-reduction

$$(\lambda x\,.\,x\ \ x)\ (\lambda x\,.\,x\ \ x)\ \ \rightarrow (\lambda x\,.\,x\ \ x)\ (\lambda x\,.\,x\ \ x)$$

- Fixed-point combinator is defined to evaluate recursive functions

$$\mathsf{fix} = \lambda f\,.\,(\lambda x\,.f\ (\lambda y\,.\,x\ \ x\ \ y))\ (\lambda x\,.f\ (\lambda y\,.\,x\ \ x\ \ y))$$

- If we define a recursive function g, then invoking function g on argument n is equivalent to applying fixed-point combinator on g:

$$\mathsf{factorial} : \mathsf{g}\ n = \mathsf{fix}\ g\ n$$

The Fix-point operator

- A fixed-point combinator is a higher-order function that returns some fixed point of its argument function

$$\mathsf{fix}\ f = f\ (\mathsf{fix}\ f) \qquad\qquad \mathsf{fix}\ f = f(f(\ldots f(\mathsf{fix}\ f)\ldots))$$

- To evaluate a fixed-point expression *fix e*, we first evaluate e to a lambda expression lambda f.e′, which is the generator of a recursive function that refers to itself as f. We then apply the lambda expression to a copy of the fixed-point expression (i.e. substituting fix (lambda f. e′) for f in e′), essentially unrolling the body of the recursive function once.

$$\frac{e \Downarrow \mathsf{lambda}\ f.\ e' \qquad [f \mapsto \mathsf{fix}\ (\mathsf{lambda}\ f.\ e')]e' \Downarrow v}{\mathsf{fix}\ \ e \Downarrow v}\ \textsc{Fix}$$

Operational Semantics

$$\frac{\mathbb{E}(x) = v}{\mathbb{E} \vdash x \Downarrow v}\ \textsc{Var}$$

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

$$\frac{\mathbb{E} \vdash e_1 \Downarrow v_1 \qquad \mathbb{E}[x \mapsto v_1] \vdash e_2 \Downarrow v_2}{\mathbb{E} \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow v_2}\ \textsc{Let}$$

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 ← 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

$$\frac{\mathbb{E} \vdash e_1 \Downarrow v_1 \qquad \mathbb{E}[x \mapsto v_1] \vdash e_2 \Downarrow v_2}{\mathbb{E} \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \Downarrow v_2}\ \textsc{Let} \qquad \frac{\mathbb{E}(x) = v}{\mathbb{E} \vdash x \Downarrow v}\ \textsc{Var}$$

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

$$\frac{E \vdash 3 : 3 \qquad \dfrac{E[x \leftarrow 3](\mathsf{x}) = 3}{E[x \leftarrow 3] \vdash \mathsf{x} : 3}}{E \vdash \mathsf{let}\ x = 3\ \mathsf{in}\ x : 3}$$

Operational Semantics

$$\frac{}{\mathbb{E} \vdash \mathsf{Nil} \Downarrow \mathsf{Nil}}\ \textsc{Nil} \qquad \frac{\mathbb{E} \vdash e_1 \Downarrow v_1 \qquad \mathbb{E} \vdash e_2 \Downarrow v_2}{\mathbb{E} \vdash e_1 @ e_2 \Downarrow v_1 @ v_2}\ \textsc{Cons}$$

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.

$$\frac{\mathbb{E} \vdash e \Downarrow \mathsf{Nil}}{\mathbb{E} \vdash \mathsf{isnil}\ e \Downarrow 1}\ \textsc{IsNilTrue} \qquad \frac{\mathbb{E} \vdash e \Downarrow v_1 @ v_2}{\mathbb{E} \vdash \mathsf{isnil}\ e \Downarrow 0}\ \textsc{IsNilFalse}$$

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.

$$\frac{\mathbb{E} \vdash e \Downarrow v_1 @ v_2}{\mathbb{E} \vdash\ !e \Downarrow v_1}\ \textsc{Head} \qquad \frac{\mathbb{E} \vdash e \Downarrow v_1 @ v_2}{\mathbb{E} \vdash \# e \Downarrow v_2}\ \textsc{Tail}$$

We define similar rules for head and tail

Congratulations!

- You can now understand every page in the λ^+ reference manual
- For HW3&4, 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

$$\frac{\mathbb{E} \vdash e_1 \Downarrow i_1 \qquad \mathbb{E} \vdash e_2 \Downarrow i_2}{\mathbb{E} \vdash e_1 + e_2 \Downarrow i_1 + i_2} \textsc{Add}$$

- 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 "→") 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

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

Small step operational semantics

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

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

- 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:
- $\langle(2+4)+(6-1), E\rangle \rightarrow \langle 6+(6-1), E\rangle \rightarrow \langle 6+5, E\rangle \rightarrow \langle 11, E\rangle$

$\boxed{\text{One atomic step at a time!}}$

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

- Will switch to type checking next week