

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

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

Lambda abstractions just evaluate to themselves

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

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, and e evaluates to a value v , we evaluate the application expression by binding v to x and then evaluating the expression $[x \mapsto v]e'_1$ as in β -reduction in lambda calculus.

The Lambda rule

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

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

- **Answer:** This would still give semantics to $((\text{lambda } x.x) 3)$, but no longer to $((\text{lambda } x. \text{lambda } y. x) 3) 4)$

The Lambda rule

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

$$\frac{e_1 \Downarrow \text{lambda } x. e'_1 \quad \cancel{e_2 \Downarrow v} \quad [x \mapsto \overset{\mathbf{e}_2}{\cancel{v}}] e'_1 \Downarrow v'}{(e_1 \ e_2) \Downarrow v'} \text{APP}$$

- **Answer:** This is also correct: you will just pass e_2 to the lambda abstraction (call-by-name)

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 λ -calculus, ALGOL 60
- Languages with call-by-value: λ^+ , 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!

Booleans: implementation

Boolean implementation

- `let TRUE = $\lambda x y. x$` -- Returns its first argument
- `let FALSE = $\lambda x y. y$` -- Returns its second argument
- `let ITE = $\lambda b x y. b x y$` -- Applies condition to branches

Why they are correct?

Booleans: examples

eval ite_true:

```
ITE TRUE e1 e2
= (λb x y. b x y) TRUE e1 e2  -- expand def ITE
=β (λx y. TRUE x y) e1 e2  -- beta-step
=β (λy. TRUE e1 y) e2  -- beta-step
=β TRUE e1 e2  -- expand def TRUE
= (λx y. x) e1 e2  -- beta-step
=β (λy. e1) e2  -- beta-step
=β e1
```

Other boolean API:

```
let NOT = λb. ITE b FALSE TRUE
```

```
let AND = λb1 b2. ITE b1 b2 FALSE
```

```
let OR = λb1 b2. ITE b1 TRUE b2
```


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

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

λ -calculus:Numbers API

- Numbers API
- **let** **INC** = $(\lambda n \lambda f \lambda x. f (n f x))$ -- Call `f` on `x` one more time than `n` does
- **let** **ADD** = $\lambda n \lambda m. n \text{ INC } m.$ -- Call `f` on `x` exactly `n + m` times

eval inc_zero :

INC ZERO

= $(\lambda n \lambda f \lambda x. f (n f x)) \text{ ZERO}$

= $_{\beta}$ $\lambda f \lambda x. f (\text{ZERO } f x)$

= $\lambda f \lambda x. f x$

= ONE

eval add_one_zero :

ADD ONE ZERO = ONE

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

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

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

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

The Fix-point operator

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

$$\text{fix } f = f (\text{fix } f)$$

$$\text{fix } f = f(f(\dots f(\text{fix } f)\dots))$$

- To evaluate a fixed-point expression “fix f is e ”, we simply unrolling its definition by replacing any recursive call with a copy of itself

$$\frac{e[f \mapsto \text{fix } f \text{ is } e] \Downarrow v}{\text{fix } f \text{ is } e \Downarrow v} \text{FIX}$$

Operational Semantics

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

To evaluate a let expression $\text{let } x = e_1 \text{ in } e_2$, we first evaluate the initial expression e_1 , which yields value v_1 . Then, to evaluate the body e_2 , we substitute occurrences of identifier x in e_2 with value v_1 , and evaluate the substituted expression, which yields value v_2 , the result of evaluating the entire let expression.

Operational Semantics

$$\frac{}{\text{Nil} \Downarrow \text{Nil}} \text{NIL} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 :: e_2 \Downarrow v_1 :: v_2} \text{CONS}$$

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.

Operational Semantics

$$\frac{e_1 \Downarrow \text{Nil} \quad e_2 \Downarrow v}{\text{match } e_1 \text{ with Nil} \rightarrow e_2 \mid x :: y \rightarrow e_3 \text{ end} \Downarrow v} \text{MATCHNIL}$$

$$\frac{e_1 \Downarrow v_1 :: v_2 \quad e_3[x \mapsto v_1][y \mapsto v_2] \Downarrow v_3}{\text{match } e_1 \text{ with Nil} \rightarrow e_2 \mid x :: y \rightarrow e_3 \text{ end} \Downarrow v_3} \text{MATCHCONS}$$

Since any list value can either be Nil or a cons cell, we have two cases for a pattern-match. Which rule is triggered will depend on whether e_1 evaluates to Nil or not.

- If e_1 evaluates to Nil, then we evaluate the Nil branch, which is e_2 .
- If e_1 evaluates to a cons cell $v_1 :: v_2$, then we evaluate the cons branch e_3 , but we also replace x with v_1 and y with v_2 .
- If e is not a list, then the evaluation will **get stuck**.

Congratulations!

- You can now understand every page in the λ^+ reference manual
- For HW2&3, 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

$$\frac{e_1 \Downarrow i_1 \quad e_2 \Downarrow i_2}{e_1 + e_2 \Downarrow i_1 + i_2} \text{ ADD}$$

- 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: Reduce the first expression

$$\frac{e_1 \longrightarrow e'_1}{e_1 + e_2 \longrightarrow e'_1 + e_2}$$

- Rule 2: Reduce the second expression once the first expression has been reduced to an integer

$$\frac{e_2 \longrightarrow e'_2}{c_1 + e_2 \longrightarrow c_1 + e'_2}$$

- Rule 3: Once both expressions have been reduced to constants, add two constants

$$\frac{c_1 + c_2 = c}{c_1 + c_2 \longrightarrow c}$$

SSOS in action

- Let's use these rules to prove what the value of $(2+4)+(6+1)$ is:
- $(2 + 4) + (6 + 1) \rightarrow 6 + (6 + 1) \rightarrow 6 + 7 \rightarrow 13$
- Thus, $(2 + 4) + (6 + 1) \rightarrow^{\star} 13$

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 of $e \Downarrow v$ is a **tree**.
- In small-step semantics, each rule only performs one step of computation
- This means any derivation of $e \rightarrow^* v$ is a **line**

TODOs by next lecture

- Will switch to type checking next week