CS 162 Programming languages

Lecture 8: Operational Semantics II

Yu Feng
Winter 2024

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

$$\frac{}{\textsf{lambda}\ x.\ e \Downarrow \textsf{lambda}\ x.\ e}\ \textsc{Lambda}$$

Lambda abstractions just evaluate to themselves

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

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

The Lambda rule

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

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

- Answer: This would still give semantics to ((lambda x.x) 3), but no longer to (((lambda x. lambda y. x) 3) 4)

The Lambda rule

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

$$\frac{e_1 \Downarrow \mathsf{lambda}\ x.\ e_1' \qquad \cancel{e_2 \Downarrow v} \qquad [x \mapsto \overset{\mathbf{e_2}}{\cancel{v}}]e_1' \Downarrow v'}{(e_1\ e_2) \Downarrow v'}\ \textsc{App}$$

- 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 uses k times, it will be evaluated k times!

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:

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

Exercise

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

factorial g :$\lambda f\ \lambda n =$ if (n = = 0) 1 else $n * f(n-1)$

factorial : g $1 =$ fix $g\ 1 = ?$

Please show me one iteration

... Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, ... neural networks. Neural Cleanse: Identifying and mitigating backdoor attacks in neural networks. Neural Cleanse: Identifying and Mitigating Backdoor Attacks in Neural Networks, page 0, 2019.
T. Gu, K. Liu, B. Dolan-Gavitt, and S. Garg. Badnets: Evaluating backdooring attacks on deep neural networks. IEEE Access, 7:47230–47244, 2019.
Alexander Turner, Dimitris Tsipras, and Aleksander Madry. Label-consistent backdoor attacks. arXiv preprint arXiv:1912.02771, 2019.

$g = \lambda fact\ \lambda n\ \text{if } n==0\ 1\ \text{else}\ n \times fact(n-1).$

$fix\ g\ 1 = \lambda f.(\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ g\ 1$

$= \underbrace{\lambda x.\ g\ (\lambda y\ x\ x\ y)}_{h}\ \underbrace{\lambda x.\ g\ (\lambda y\ x\ x\ y))}_{h}\ 1$

$= g\ (\lambda y\ h\ h\ y)\ 1$

$= \lambda fact\ \lambda n\ (\ ---\ )\ (\lambda y\ h\ h\ y)\ 1$

$= 1 * ((\lambda y\ h\ h\ y)\ 0)$

$= 1 * (h\ h\ 0)$

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 "$\mathsf{fix}\ f\ \mathsf{is}\ e$", we simply unrolling its definition by replacing any recursive call with a copy of itself

$$\frac{e[f \mapsto \mathsf{fix}\ f\ \mathsf{is}\ e] \Downarrow v}{\mathsf{fix}\ f\ \mathsf{is}\ e \Downarrow v}\ \textsc{Fix}$$

Operational Semantics

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

To evaluate a let expression let x = e_1 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

$$\frac{}{\mathsf{Nil} \Downarrow \mathsf{Nil}}\ \textsc{Nil} \qquad\qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 :: e_2 \Downarrow v_1 :: v_2}\ \textsc{Cons}$$

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.

Operational Semantics

$$\frac{e_1 \Downarrow \mathsf{Nil} \qquad e_2 \Downarrow v}{\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{Nil} \to e_2 \mid x :: y \to e_3\ \mathsf{end} \Downarrow v}\ \textsc{MatchNil}$$

$$\frac{e_1 \Downarrow v_1 :: v_2 \qquad e_3[x \mapsto v_1][y \mapsto v_2] \Downarrow v_3}{\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{Nil} \to e_2 \mid x :: y \to e_3\ \mathsf{end} \Downarrow v_3}\ \textsc{MatchCons}$$

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

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

- 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 "→") 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: 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, $(2 + 4) + (6 + 1) \rightarrow^{\star} 13$

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

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 of $e \Downarrow v$ is a tree.

- In small-step semantics, each rule only performs one step of computation

- This means any derivation of $e \rightarrow^{\star} v$ is a line

TODOs by next lecture

- Will switch to type checking next week