Disc 9 - OpSem and Lambda Calculus

Thursday, November 4, 2021 11:09 AM

Operational Semantics

2. Using the rules given below, show: $1 + (2 + 3) \Rightarrow 6$

$$\frac{}{n \Rightarrow n} \qquad \frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad n_3 \textit{ is } n_1 + n_2}{e_1 + e_2 \Rightarrow n_3}$$

$$\cfrac{1 \Rightarrow 1 \quad \cfrac{2 \Rightarrow 2 \quad 3 \Rightarrow 3 \quad 5 \text{ is } 2+3}{2+3 \Rightarrow 5} \quad 6 \text{ is } 1+5}{1 + (2+3) \Rightarrow 6}$$

4. Using the rules given below, show: $A;\ \textit{let } y = 1 \textit{ in let } x = 2 \textit{ in } x \Rightarrow 2$

$$\frac{}{A;\ n \Rightarrow n} \qquad \frac{A(x) = v}{A;\ x \Rightarrow v}$$

$$\frac{A;\ e_1 \Rightarrow v_1 \quad A, x : v_1;\ e_2 \Rightarrow v_2}{A;\ \textbf{let } x = e_1 \textbf{ in } e_2 \Rightarrow v_2} \qquad \frac{A;\ e_1 \Rightarrow n_1 \quad A;\ e_2 \Rightarrow n_2 \quad n_3 \textit{ is } n_1 + n_2}{A;\ e_1 + e_2 \Rightarrow n_3}$$

$$\cfrac{A;\ 1 \Rightarrow 1 \quad \cfrac{A, x{:}1;\ 2 \Rightarrow 2 \quad \cfrac{A, x{:}2;\ (x) = 2}{A, x{:}2;\ x \Rightarrow 2}}{A, x{:}1;\ \text{let } x = 2 \text{ in } x \Rightarrow 2}}{A;\ \text{let } x = 1 \text{ in let } x = 2 \text{ in } x \Rightarrow 2}$$

5) Recall last week we went over lexing and parsing:

```
type expr =
| Int of int
| Plus of expr * expr
```

Implement an expression evaluator, that takes an environment closure and an expression, and returns a value after evaluating it.

Key Notes (Taken from OpSem rules, which will be given on the project)
- Integers evaluate to themselves
- Plus works on integers (throw a TypeError otherwise)

```
let rec eval_expr env e =
```

see below!

Lambda Calculus

a - bound variable
b - unbound variable

λa. a b

1) (λa. a) b

Make the parentheses explicit in the following expressions

2) a b c = ((a b) c)
3) λa. λb. a b = (λa. (λb. (a b))) ← fun (two args)
4) λa. a b λa. a b = (λa. ((a b) (λa. (a b))))

Identify the free variables in the following expressions

1) λa. a b a = (λa. ((a b) a))
2) a (λa. a) a = (a (λa. a) a)
3) λa. (λb. a b) a b = (λa. (λb. (a b)) (a b))

Apply alpha-conversions to the following

1) λa. λa. a = λb. λa. a or λa. λb. b
2) (λa. a) a b - (λc. c) a b
3) (λa. (λa. (λa. a) a) a) a = (λd. (λc. (λb. b) c) d) a

Apply beta-reductions to the following

1) (λa. a b) x b = (((λa. (a b)) x) b) = (x b) b
2) (λa. b) (λa. λb. λc. a b c) = b
3) (λa. a a) (λa. a a) = y y
= (λa. a a) (λa. a a)
= Can't reduce further!

```
let rec eval_expr env e = match e with

        | Int i -> Int i
        | Plus (e1, e2) -> let v1 = eval_expr env e1 in
                           let v2 = eval_expr env e2 in
                           match (v1, v2) with

                           | (Int i1, Int i2) -> i1+i2
                           | (Plus (e3,e4), Int i) -> let val = eval_expr env
                                                                  (Plus (e3,e4))
                                                      in val + i
                           | (Int i, Plus(e3, e4)) -> let val = eval expr env
                                                                  (Plus (e3,e4))
                                                      in val + i

                           | (Plus(e3, e4), Plus(e5, e6)) ->

                                    let val1 = eval_expr env (Plus(e3,e4))
                                 in let val2 = eval_expr env (Plus (e5,e6))
                                 in val1 + val2
```

Correction! we don't need to match Plus in the inner match because that is handled by recursion!

Precise solution below!

```
let rec eval_expr env e = match e with

        | Int i -> Int i
        | Plus (e1, e2) -> let v1 = eval_expr env e1 in
                           let v2 = eval_expr env e2 in
                           match (v1, v2) with

                           | (Int i1, Int i2) -> i1+i2
                           | _ -> raise (TypeError "Invalid!")
```