

Lecture 19 — MiniOCaml evaluation rules

- Substitution model — rules and examples
- Environment model — rules and examples

Abstract syntax of MiniOCaml

```
type exp =  
  Operation of exp * binary_operation * exp  
| UnaryOperation of unary_operation * exp  
| Var of string | StrConst of string | IntConst of int  
| FloatConst of float | True | False  
| List of exp list | Tuple of exp list  
| If of exp * exp * exp | App of exp * exp  
| Let of string * exp * exp  
| Fun of string * exp  
| Rec of string * exp  
  
and binary_operation = Semicolon | Comma | Equals | LessThan  
| GreaterThan | NotEquals | Assign | And | Or  
| IntPlus | IntMinus | IntDiv | IntMult  
| FloatPlus | FloatMinus | FloatDiv | FloatMult  
| StringAppend | ListAppend  
  
and unary_operation = Not | Head | Tail | Fst | Snd
```

Substitution model evaluation rules

(Const) $\text{Const } x \Downarrow \text{Const } x$

(Fun) $\text{Fun}(a, e) \Downarrow \text{Fun}(a, e)$

(Rec) $\text{Rec}(f, \text{Fun}(a, e)) \Downarrow \text{Fun}(a, e[\text{Rec}(f, \text{Fun}(a, e))/f])$

(δ) $e \text{ op } e' \Downarrow v \text{ OP } v'$
 $e \Downarrow v$
 $e' \Downarrow v'$

(δ) $\text{op } e \Downarrow \text{OP } v$
 $e \Downarrow v$

(If) $\text{If}(e_1, e_2, e_3) \Downarrow v$
 $e_1 \Downarrow \text{True}$
 $e_2 \Downarrow v$

(If) $\text{If}(e_1, e_2, e_3) \Downarrow v$
 $e_1 \Downarrow \text{False}$
 $e_3 \Downarrow v$

(List) $[e_1, \dots, e_n] \Downarrow [v_1, \dots, v_n]$
 $e_1 \Downarrow v_1$
 $\vdots$
 $e_n \Downarrow v_n$

(App) $e \ e' \Downarrow v$
 $e \Downarrow \text{Fun}(a, e'')$
 $e' \Downarrow v'$
 $e''[v'/a] \Downarrow v$

(Let) $\text{Let}(a, e, e') \Downarrow v'$
 $e \Downarrow v$
 $e'[v/a] \Downarrow v'$

Ex. evaluation by substitution

```
let x = 3 in x+1
```

```
(fun x -> x+1) 3
```

Ex. evaluation by substitution

```
((fun x -> fun y -> x+y) 3) 4
```

Ex. evaluation by substitution

```
let f = fun x -> fun y -> x+y  
in let g = f 1 in g 2
```

Concrete syntax for “rec”

- For the following example, pretend we have *concrete* syntax corresponding to abstract syntax operator `Rec`, so we can write:

```
let fac = rec fac (fun x -> if x=0 then 1 else x * fac(x-1)) in fac 2
```

This would translate to

```
Let("fac", Rec("fac", Fun("x", If( ...))), App(Var "fac", IntConst 2))
```

- This allows us to continue to use concrete syntax, which is much more convenient

Ex. evaluation by substitution

Let $\mathcal{F}$ denote “`rec fac (fun x -> if x=0 then 1 else x * fac(x-1))`”.

`let fac =  $\mathcal{F}$  in fac 2` $\Downarrow$ 2

$\mathcal{F} \Downarrow$ `fun x -> if x=0 then 1 else x *  $\mathcal{F}$ (x-1)`

`(fun x -> if x=0 then 1 else x *  $\mathcal{F}$ (x-1)) 2` $\Downarrow$ 2

`fun x -> if x=0 then 1 else x *  $\mathcal{F}$ (x-1)  $\Downarrow$  fun x -> if x=0 then 1 else x *  $\mathcal{F}$ (x-1)`
2 $\Downarrow$ 2

`if 2=0 then 1 else 2 *  $\mathcal{F}$ (2-1)` $\Downarrow$ 2

`2=0` $\Downarrow$ False

`2` $\Downarrow$ 2

`0` $\Downarrow$ 0

`2 *  $\mathcal{F}$ (2-1)` $\Downarrow$ 2

Evaluation in the environment model

- Substitution model is easy to understand, but it is not realistic — it does not reflect how actual implementations work. Also, it cannot be used to evaluate terms with side-effecting operations.
- *Environment model* is based on a simple optimization:
 - To apply function $\text{Fun}(x, e)$ to value v , instead of creating a new copy of e with all the x 's replaced by v 's, just *record* that x has value v in a separate data structure, called an *environment*.
 - All expression evaluation occurs “within” an environment.
 - To remember the values of variables in a function, need to create a *closure* $\langle e, \rho \rangle$.

Environment model evaluation rules

(Const) $\text{Const } c, \rho \Downarrow \text{Const } c$

(Var) $a, \rho \Downarrow \rho(a)$

(Fun) $\text{Fun}(a,e), \rho \Downarrow \langle \text{Fun}(a,e), \rho \rangle$

(Rec) $\text{Rec}(f,e), \rho \Downarrow \langle \text{Rec}(f,e), \rho \rangle$

(δ) $e \text{ op } e', \rho \Downarrow v \text{ OP } v'$

$e, \rho \Downarrow v$

$e', \rho \Downarrow v'$

(δ) $\text{op } e, \rho \Downarrow \text{OP } v$

$e, \rho \Downarrow v$

(If) $\text{If}(e_1, e_2, e_3), \rho \Downarrow v$

$e_1, \rho \Downarrow \text{True}$

$e_2, \rho \Downarrow v$

(If) $\text{If}(e_1, e_2, e_3), \rho \Downarrow v$

$e_1, \rho \Downarrow \text{False}$

$e_3, \rho \Downarrow v$

(List) $[e_1, \dots, e_n], \rho \Downarrow [v_1, \dots, v_n]$

$e_1, \rho \Downarrow v_1$

$\vdots$

$e_n, \rho \Downarrow v_n$

(Let) $\text{Let}(a,e,e'), \rho \Downarrow v'$

$e, \rho \Downarrow v$

$e', \rho[a \mapsto v] \Downarrow v'$

(App) $e \ e', \rho \Downarrow v$

$e, \rho \Downarrow \langle \text{Fun}(a, e''), \rho' \rangle$

$e', \rho \Downarrow v'$

$e'', \rho'[a \mapsto v'] \Downarrow v$

(App) $e \ e', \rho \Downarrow v''$

$e, \rho \Downarrow v,$

where $v = \langle \text{Rec}(f, \text{Fun}(a, e'')), \rho' \rangle$

$e', \rho \Downarrow v'$

$e'', \rho'[a \mapsto v', f \mapsto v] \Downarrow v''$

Evaluation in environment model

- $\emptyset$ denotes the empty environment. We may write $\emptyset[x \mapsto v]$ as $\{x \mapsto v\}$.

`let x = 3 in x+1,  $\emptyset$`

`(fun x -> x+1) 3,  $\emptyset$`

Evaluation in environment model

```
((fun x -> fun y -> x+y) 3) 4,  ∅
```

Evaluation in environment model

```
let f = fun x -> fun y -> x+y,  ∅  
in let g = f 1 in g 2
```

Evaluation in environment model

Let $\mathcal{F}$ denote “`rec fac (fun x -> if x=0 then 1 else x * fac(x-1))`”.

`let fac =  $\mathcal{F}$  in fac 2,  $\emptyset \Downarrow 2$`

`$\mathcal{F}, \emptyset \Downarrow < \mathcal{F}, \emptyset >$`

`fac 2,  $\rho_1 \Downarrow 2$      ( $\rho_1 = \{fac \mapsto < \mathcal{F}, \emptyset >\}$ )`

`fac,  $\rho_1 \Downarrow < \mathcal{F}, \emptyset >$`

`2,  $\rho_1 \Downarrow 2$`

`if x=0 then 1 else x * fac(x-1),  $\rho_1[x \mapsto 2] \Downarrow 2$`

`x=0,  $\rho_1[x \mapsto 2] \Downarrow \text{False}$`

`x,  $\rho_1[x \mapsto 2] \Downarrow 2$`

`0,  $\rho_1[x \mapsto 2] \Downarrow 0$`

`x * fac(x-1),  $\rho_1[x \mapsto 2] \Downarrow 2$`