

# EXTENSIONS TO SIMPLY-TYPED LAMBDA CALCULUS (II)

1

# RECALL SUMS (SEMANTICS)

$$\frac{}{\text{case (inl } v) \text{ of inl } x_1 \Rightarrow e_1 \mid \text{inr } x_2 \Rightarrow e_2 \rightarrow e_1[v/x_1]} \quad (\text{E - CaseInl})$$

$$\frac{}{\text{case (inr } v) \text{ of inl } x_1 \Rightarrow e_1 \mid \text{inr } x_2 \Rightarrow e_2 \rightarrow e_2[v/x_2]} \quad (\text{E - CaseInr})$$

$$\frac{e \rightarrow e'}{\text{case } e \text{ of inl } x_1 \Rightarrow e_1 \mid \text{inr } x_2 \Rightarrow e_2 \rightarrow \text{case } e' \text{ of inl } x_1 \Rightarrow e_1 \mid \text{inr } x_2 \Rightarrow e_2} \quad (\text{E - Case})$$

$$\frac{e \rightarrow e'}{\text{inl } e \rightarrow \text{inl } e'} \quad (\text{E - Inl})$$

$$\frac{e \rightarrow e'}{\text{inr } e \rightarrow \text{inr } e'} \quad (\text{E - Inr})$$

# SUMS (TYPING)

$$\frac{\Gamma \mid - e : t_1}{\Gamma \mid - \text{inl } e : t_1 + t_2} \quad (\text{T-Inl})$$

$$\frac{\Gamma \mid - e : t_2}{\Gamma \mid - \text{inr } e : t_1 + t_2} \quad (\text{T-Inr})$$

$$\frac{\Gamma \mid - e : t_1 + t_2 \quad \Gamma, x_1 : t_1 \mid - e_1 : t \quad \Gamma, x_2 : t_2 \mid - e_2 : t}{\Gamma \mid - \text{case } e \text{ of inl } x_1 \Rightarrow e_1 \mid \text{inr } x_2 \Rightarrow e_2 : t} \quad (\text{T-Case})$$

- (T-Inl) and (T-Inr) is problematic! Why?
- Given  $e$  of a fixed type,  $\text{inl } e$  is of type  $t_1 + t_2$ , for any  $t_2$ !
- This breaks the “uniqueness lemma”.

## SUMS (WITH UNIQUE TYPING)

- We can annotate sums with a unique type:

$e ::= \dots$                       expressions:  
|  $\text{inl}[t] e$                       injection (left)  
|  $\text{inr}[t] e$                       injection (right)

- The typing rules are modified as:

$$\frac{\Gamma \mid - e : t_1}{\Gamma \mid - \text{inl}[t_1 + t_2] e : t_1 + t_2} \quad (\text{T-Inl})$$

$$\frac{G \mid - e : t_2}{G \mid - \text{inr}[t_1 + t_2] e : t_1 + t_2} \quad (\text{T-Inr})$$

# MORE COMPLEX EXAMPLE: ADDRESSES

- Types:
  - `userid = string`
  - `ip = int * int * int * int`
  - `host = {machine: string, org: string, country: string}`
  - `domain = host + ip`
  - `email_address = userid * domain`
  - `home_address = {number: int, street: string, city : string, state : string, country: string}`
  - `address = email_address + home_address`
  - Examples:
    - [john@gala.amazon.cn](mailto:john@gala.amazon.cn)
    - [ben@192.168.1.1](mailto:ben@192.168.1.1)
    - 123 Main Street, Seattle, WA, USA.
- Function to extract the country from an address:

```
\x. case x of
  inl email =>
    let d = email.2 in
      case d of inl host => host.country
                | inr ip => "NA"
  | inr home => home.country
```

# VARIANTS

- Binary sums generalizes to variants just like pairs generalized to labeled records.
- Instead of using  $\text{inl}[t_1+t_2] e$ ,  
we use  $\text{in}_1[t_1+t_2] e$ .
- $e ::= .. \mid \text{in}_i e_i$
- $t ::= .. \mid t_1 + \dots + t_n$
- Detailed rules left as an exercise.

# RECURSIVE FUNCTIONS

- Divergent combinator:

- $\text{omega} = (\lambda x. x x) (\lambda x. x x)$   
→  $(\lambda x. x x) (\lambda x. x x)$   
→ ...

- Infinite loop and no normal form: hence the term *divergent*.

- More generally, fix-point combinator (a.k.a. call-by-value Y-combinator):

- $\text{fix} = \lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))$
- We explain how it works by factorial example

# FIX-POINT COMBINATOR (FACTORIAL EXAMPLE)

- A naïve definition of factorial function:

factorial =  $\lambda n. \text{if } n=0 \text{ then } 1$

          else  $n * (\text{if } n-1=0 \text{ then } 1$

                  else  $(n-1) * (\text{if } n-2=0 \text{ then } 1)$

                          else  $(n-2) * \dots$

- We can use the fix-point combinator instead:

$g = \lambda \text{fct}. \lambda n. \text{if } n=0 \text{ then } 1 \text{ else } n * (\text{fct } (n-1))$

factorial = fix g

factorial:  $\text{int} \rightarrow \text{int}$             fct:  $\text{int} \rightarrow \text{int}$

g:  $(\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int}$

*which is equivalent to:*  $(\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})$

# FIX-POINT COMBINATOR (FACTORIAL EXAMPLE)

- $g = \lambda \text{fct. } \lambda n. \text{ if } n=0 \text{ then } 1 \text{ else } n * (\text{fct } (n-1))$   
factorial = fix g (Recall:  $\text{fix} = \lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))$ )
- E.g., factorial 3 =

fix g 3

→ h h 3

-- where  $h = \lambda x. g (\lambda y. x x y)$

→ g fct 3

-- where  $\text{fct} = \lambda y. h h y$  (Notice we abuse “fct” a bit here.)

→  $\lambda n. \text{ if } n=0 \text{ then } 1 \text{ else } n * (\text{fct } (n-1))$  3

→ if 3=0 then 1 else 3 \* (fct (3-1))

→ \* 3 \* (fct 2)

→ 3 \* (h h 2)

→ 3 \* (g fct 2)

→ \* 3 \* 2 \* (g fct 1)

→ \* 3 \* 2 \* 1 \* (g fct 0)

→ \* 6

Recursion happens!



# GENERAL RECURSION

## Syntax:

$e ::= \dots$                       expressions:  
|  $\text{fix } e$                       fix point of  $e$

## Evaluation:

$\text{fix } (\lambda x: t. e) \rightarrow e [(\text{fix } (\lambda x: t. e)) / x]$  (E-FixBeta)

$$\frac{e \rightarrow e'}{\text{fix } e \rightarrow \text{fix } e'} \quad (\text{E - Fix})$$

## Typing:

$$\frac{\Gamma \mid - e : t_1 \rightarrow t_1}{\Gamma \mid - \text{fix } e : t_1} \quad (\text{T - Fix})$$

# ANOTHER RECURSIVE EXAMPLE: ISEVEN

○  $ff = \lambda ie: int \rightarrow bool.$

$\lambda x: int .$

        if  $x = 0$  then true

        else if  $x > 0$  then

            if  $x = 1$  then false

            else  $ie (x - 2)$

        else

            if  $x = (\sim 1)$  then false

            else  $ie (x + 2)$

    •  $ff : (int \rightarrow bool) \rightarrow int \rightarrow bool$

○  $iseven = fix\ ff$

    •  $iseven : int \rightarrow bool$

○  $iseven\ 7 \rightarrow^* false$

○  $iseven\ (\sim 6) \rightarrow^* true$

# QUIZ

- Using fix point combinator, implement a recursive function  $\text{sum}: \text{int} \rightarrow \text{int}$ , such that given input  $N$ , returns  $\sum_{n=1}^N n$ .

hint: define a function  $\text{ss}: (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})$

and then  $\text{sum} = \text{fix ss}$ .

**Evaluation:**

$$\text{fix } (\lambda x: t. e) \rightarrow e [(\text{fix } (\lambda x: t. e)) / x] \quad (\text{E-FixBeta})$$

$$\frac{e \rightarrow e'}{\text{fix } e \rightarrow \text{fix } e'} \quad (\text{E - Fix})$$

**Typing:**

$$\frac{\Gamma \mid - e : t_1 \rightarrow t_1}{\Gamma \mid - \text{fix } e : t_1} \quad (\text{T - Fix})$$

# LISTS

- List is a common recursive data structure

## Syntax:

$e ::= \dots$

|  $\text{nil}[t]$

|  $e_1::e_2$

| case  $e$  of  $\text{nil} \Rightarrow e_1$

|  $x_1::x_2 \Rightarrow e_2$

$v ::= \dots$

|  $\text{nil}$

|  $v_1 :: v_2$

$t ::= \dots$

|  $t$  list

## expressions:

empty list

list constructor

list destructor

## values:

empty list

list constructor

## types:

type of lists

- $[1, 2, 3, 4]$  is written as  $1::(2::(3::(4::\text{nil})))$ .
- In above list, 1 is the head of list,  $(2::(3::(4::\text{nil})))$  is the tail.
- Every list ends with  $\text{nil}$ .

# LIST (EVALUATION)

$$\frac{}{\text{case nil of nil } \Rightarrow e_1 \mid x_1 :: x_2 \Rightarrow e_2 \rightarrow e_1} \quad (\text{E - CaseNil})$$

$$\frac{}{\text{case } v_1 :: v_2 \text{ of nil } \Rightarrow e_1 \mid x_1 :: x_2 \Rightarrow e_2 \rightarrow e_2[v_1 / x_1][v_2 / x_2]} \quad (\text{E - CaseCons})$$

$$\frac{e \rightarrow e'}{\text{case } e \text{ of nil } \Rightarrow e_1 \mid x_1 :: x_2 \Rightarrow e_2 \rightarrow \text{case } e' \text{ of nil } \Rightarrow e_1 \mid x_1 :: x_2 \Rightarrow e_2} \quad (\text{E - ListCase})$$

$$\frac{e_1 \rightarrow e_1'}{e_1 :: e_2 \rightarrow e_1' :: e_2} \quad (\text{E - Cons1})$$

$$\frac{e_2 \rightarrow e_2'}{v_1 :: e_2 \rightarrow v_1 :: e_2'} \quad (\text{E - Cons2})$$

# LIST (TYPING)

$$\frac{}{\Gamma \mid - \text{nil}[t] : t \text{ list}} \quad (\text{T - nil}) \qquad \frac{\Gamma \mid - e_1 : t \quad e_2 : t \text{ list}}{\Gamma \mid - e_1 :: e_2 : t \text{ list}} \quad (\text{T - Cons})$$

$$\frac{\Gamma \mid - e : t_1 \text{ list} \quad \Gamma \mid - e_1 : t \quad \Gamma, x_1 : t_1, x_2 : t_1 \text{ list} \mid - e_2 : t}{\Gamma \mid - \text{case } e \text{ of nil}[t_1] \Rightarrow e_1 \mid x_1 :: x_2 \Rightarrow e_2 : t} \quad (\text{T - Case})$$

- Note that only nil needs to be annotated with an explicit type. Types of other expressions can be inferred from the typing rules.

## EXAMPLE: SUM A LIST OF NUMBERS

- $ff = \backslash sl : \text{int list} \rightarrow \text{int}.$ 
  - $\backslash l : \text{int list}.$ 
    - case l of nil  $\Rightarrow$  0
    - |  $x::l \Rightarrow x + (sl l)$
  - $ff : (\text{int list} \rightarrow \text{int}) \rightarrow \text{int list} \rightarrow \text{int}$
- $sum = \text{fix } ff$ 
  - $sum : \text{int list} \rightarrow \text{int}$
- E.g.  $sum (4::3::2::1) \rightarrow^* 10$

## ANOTHER EXAMPLE: REVERSE A LIST

- $gg = \backslash ap: \text{int list} \rightarrow \text{int} \rightarrow \text{int list}.$   
     $\backslash l: \text{int list}. \backslash n: \text{int}.$   
    case l of nil => n::nil  
        | x :: l => x :: (ap l n)
- $append = \text{fix } gg : \text{int list} \rightarrow \text{int} \rightarrow \text{int list}$
- $ff = \text{let } append = \text{fix } gg \text{ in}$   
     $\backslash rev: \text{int list} \rightarrow \text{int list}.$   
     $\backslash l: \text{int list}.$   
    case l of nil => nil  
        | x :: l => append (rev l) x
- $reverse = \text{fix } ff : \text{int list} \rightarrow \text{int list}$
- $reverse (4::3::2::1::\text{nil}) \rightarrow^* 1::2::3::4::\text{nil}$

# FUNCTION IMPLEMENTATIONS

- Function application is implemented by “substitution” so far:

$$(\lambda x.e1) e2 \rightarrow e1 [e2/x]$$

- This is not efficient because:
  - Search through  $e1$  for free occurrences of  $x$  during substitution
  - Go through  $e1$  again to evaluate it:  $e1 \rightarrow^* v1$
  - That’s double the work!
- There’s an alternate way using “environment.”
- Be extremely lazy!
- This is closer to how PL interpreters actually work.

# ENVIRONMENT MODEL

- An environment is a (variable, value) mapping (set of bindings):

$$E ::= . \mid E, x \ v$$

- Define  $E[x \ v]$  (add a binding into the environment):

$$.[x \ v] = x \ v$$

$$(E, x' \ v')[x \ v] = E, x \ v \quad \text{if } x = x'$$

$$\text{or } E, x' \ v', x \ v \quad \text{if } x \neq x'$$

- We define values to be either constants (e.g., true, false, 5, etc.) or *closures*.

- A *closure* is a pair of a function and its environment.

$$v ::= \dots \mid \{\lambda x.e, E\}$$

- The new multi-step evaluation judgment:

$$(E, e) \rightarrow^* v$$

# ENVIRONMENT MODEL (EVALUATION)

$$\frac{E(x) = v}{(E, x) \rightarrow^* v} \quad (\text{E - var})$$

$$\frac{}{(E, \lambda x.e) \rightarrow^* \{\lambda x.e, E\}} \quad (\text{E - fun})$$

$$\frac{(E, e_1) \rightarrow^* \{\lambda x.e, E_1\} \quad (E, e_2) \rightarrow^* v_2 \quad (E_1[x \mapsto v_2], e) \rightarrow^* v,}{(E, (e_1 \ e_2)) \rightarrow^* v} \quad (\text{E - app})$$

$$\frac{(E, e_1) \rightarrow^* v_1 \quad (E[x \mapsto v_1], e_2) \rightarrow^* v_2}{(E, \text{let } x = e_1 \text{ in } e_2) \rightarrow^* v_2} \quad (\text{E - let})$$

- Subtlety: for nested function applications, e.g.

$(\lambda x. \lambda y. \lambda z. x + y + z) \ 1 \ 2 \ 3$ ,

the environment for each function application is organized in a stack, i.e. the call stack. Items on the call stack are called “stack frames” or “activation records.”

# A NON-TRIVIAL EXAMPLE

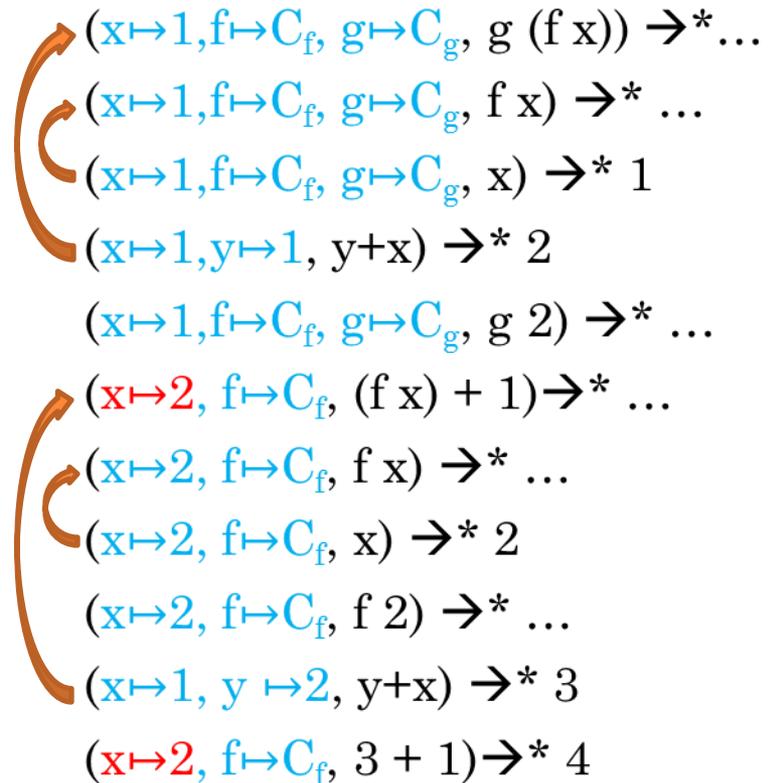
let  $x = 1$  in

let  $f = \lambda y. y + x$  in

let  $g = (\lambda x. f x) + 1$  in  
 $g (f x)$

$$C_f = \{\lambda y. y+x, x \mapsto 1\}$$

$$C_g = \{\lambda x. (f x) + 1, x \mapsto 1, f \mapsto C_f\}$$



Exercise: Think of a better way of presenting the evaluation process?

# ENVIRONMENT MODEL (CAPTURING)

- Environment automatically fixes capturing problem:

- By substitution without alpha conversion:

$$(\lambda z. \lambda x. z + x) x 5 \rightarrow (\lambda x. x + x) 5 \rightarrow 10$$

- By environment:

$$(\cdot, (\lambda z. \lambda x. z + x) x 5) \rightarrow$$

$$(z \mapsto x, (\lambda x. z + x) 5) \rightarrow$$

$$(z \mapsto x, x \mapsto 5, z + x) \rightarrow$$

$$x + 5$$