# Extensions to Simply-Typed Lambda Calculus (II)

# Recall Sums (semantics)

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

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

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

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

# Sums (Typing)

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

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

- (T-Inl) and (T-Inr) is problematic! Why?
- Given e of a fixed type, 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 ::= … | | expressions: |
| | \| inl[t] e | injection (left) |
| | \| inr[t] e | injection (right) |

- The typing rules are modified as:

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

# 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
    - 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 $\text{in}_1[t_1+t_2]\ e$.
- $e ::= .. \mid \text{in}_i\ e_i$
- $t ::= .. \mid t_1 + \ldots + t_n$
- Detailed rules left as an exercise.

# Recursive Functions

- Divergent combinator:
  - omega = (\x. x x) (\x. x x)
    → (\x. x x) (\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):
  - fix = \f. (\x. f (\y. x x y)) (\x. f (\y. x x y))
  - We explain how it works by factorial example

# Fix-point Combinator (Factorial Example)

- A naïve definition of factorial function:

```
factorial = \ n. if n=0 then 1
                 else n * (if n-1=0 then 1
                           else (n-1) * (if n-2=0 then 1)
                                        else (n-2) * …
```

- We can use the fix-point combinator instead:

  g = \fct. \n. if n=0 then 1 else n * (fct (n-1))

  factorial = fix g

  factorial: int → int  fct: int → int

  g: (int → int) → int → int

  *which is equivalent to*: (int → int) → (int → int)

# Fix-point Combinator (Factorial Example)

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


fix g 3
→ h h 3
-- where h = \x. g (\y. x x y)
→g fct 3
-- where fct = \y. h h y (Notice we abuse "fct" a bit here.)
→ \n. if n=0 then 1 else n * (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

# General Recursion

**Syntax:**

e ::= …  expressions:

| fix e  fix point of e

**Evaluation:**

fix (\x: t. e) → e [(fix (\x: t. e)) / x]  (E-FixBeta)

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

**Typing:**

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

# Another Recursive Example: iseven

- ff = \ ie: int → bool.

```
        \x: int .
            if x = 0 then true
            else if x > 0 then
                    if x = 1 then false
                    else ie (x – 2)
                else
                    if x = (~1) then false
                    else ie (x + 2)
```

  - ff : (int → bool) → int → bool
- iseven = fix ff
  - iseven : int → bool
- iseven 7 →* false
- iseven (~6) →* true

# Lists

- List is a common recursive data structure

**Syntax:**

| | |
|---|---|
| e ::= … | **expressions:** |
| \| nil[t] | empty list |
| \| e1::e2 | list constructor |
| \| case e of nil => e1 | |
| \| x1::x2 => e2 | list destructor |
| v ::= … | **values:** |
| \| nil | empty list |
| \| v1 :: v2 | list constructor |
| t ::= … | **types:** |
| \| t list | type of lists |

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

# List (Evaluation)

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

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

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

$$\frac{e_1 \rightarrow e_1'}{e_1 :: e_2 \rightarrow e_1' :: e_2} \quad \text{(E-Cons1)} \qquad \frac{e_2 \rightarrow e_2'}{v_1 :: e_2 \rightarrow v_1 :: e_2'} \quad \text{(E-Cons2)}$$

# List (Typing)

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

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

- 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 = \sl : int list → int.

```
        \l : int list.
            case l of nil => 0
                    | x::l => x + (sl l)
```

  - ff : (int list → int) → int list → int
- sum = fix ff
  - sum : int list → int

- E.g. sum (4::3::2::1) →* 10

# Another Example: Reverse a List

- gg = \ap: int list→int→int list.
  \l : int list. \n : int.
  case l of nil => n::nil
  | x :: l => x :: (ap l n)
- append = fix gg : int list → int → int list
- ff = let append = fix gg in
  \rev: int list → int list.
  \l : int list.
  case l of nil => nil
  | x :: l => append (rev l) x
- reverse = fix ff : int list → int list
- reverse (4::3::2::1::nil) →* 1::2::3::4::nil

# Function Implementations

- Function application is implemented by "substitution" so far:

  (\x.e1)  e2 → e1 [e2/x]

- This is not efficient because:
  - Search through e1 for free occurrences of x during substitution
  - Go though e1 again to evaluate it: e1 →* 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 ::= . | 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  if x = x'

  or E, x' v', x v  if x ≠ 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 ::= … | {\x.e, E}

- The new multi-step evaluation judgment:

  (E, e) →* v

# Environment Model (evaluation)

$$\frac{E(x) = v}{(E, x) \to^* v} \text{ (E - var)} \qquad \frac{}{(E, \lambda x.e) \to^* \{\lambda x.e, E\}} \text{ (E - fun)}$$

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

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

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

  (\x.\y.\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 = \y. y + x in
let g = (\x. f x) + 1 in
g (f x)
```

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

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

$(x \mapsto 1, f \mapsto C_f, g \mapsto C_g,\ g\ (f\ x)) \rightarrow^* \ldots$

$(x \mapsto 1, f \mapsto C_f, g \mapsto C_g,\ f\ x) \rightarrow^* \ldots$

$(x \mapsto 1, f \mapsto C_f, g \mapsto C_g,\ x) \rightarrow^* 1$

$(x \mapsto 1, y \mapsto 1,\ y+x) \rightarrow^* 2$

$(x \mapsto 1, f \mapsto C_f, g \mapsto C_g,\ g\ 2) \rightarrow^* \ldots$

$(x \mapsto 2, f \mapsto C_f,\ (f\ x) + 1) \rightarrow^* \ldots$

$(x \mapsto 2, f \mapsto C_f,\ f\ x) \rightarrow^* \ldots$

$(x \mapsto 2, f \mapsto C_f,\ x) \rightarrow^* 2$

$(x \mapsto 2, f \mapsto C_f,\ f\ 2) \rightarrow^* \ldots$

$(x \mapsto 1, y \mapsto 2,\ y+x) \rightarrow^* 3$

$(x \mapsto 2, f \mapsto C_f,\ 3 + 1) \rightarrow^* 4$

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

# Environment Model (Capturing)

- Environment automatically fixes capturing problem:
- By substitution without alpha conversion:

  (\z.\x. z + x) x 5 → (\x. x + x) 5 → 10

- By environment:

  (., (\z.\x. z + x) x 5) →

  (z ↦ x, (\x.z + x) 5) →

  (z ↦ x, x ↦ 5, z + x) →

  x + 5