Extensions to Simply-Typed Lambda Calculus (II)

RECALL SUMS (SEMANTICS)

$$\frac{}{\text{case (inl v) of inl } \mathrm{x}_1 \text{ => } \mathrm{e}_1 \mid \text{inr } \mathrm{x}_2 \text{ => } \mathrm{e}_2 \rightarrow e_1[v/x_1]} \text{ (E - CaseInl)}$$

$$\frac{}{\text{case (inr v) of inl } \mathrm{x}_1 \text{ => } \mathrm{e}_1 \mid \text{inr } \mathrm{x}_2 \text{ => } \mathrm{e}_2 \rightarrow e_2[v/x_2]} \text{ (E - CaseInr)}$$

$$\frac{e \rightarrow e'}{\begin{array}{l}\text{case e of inl } \mathrm{x}_1 \text{ => } \mathrm{e}_1 \mid \text{inr } \mathrm{x}_2 \text{ => } \mathrm{e}_2 \\ \rightarrow \text{case e' of inl } \mathrm{x}_1 \text{ => } \mathrm{e}_1 \mid \text{inr } \mathrm{x}_2 \text{ => } \mathrm{e}_2\end{array}} \text{ (E - Case)}$$

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

Sums (Typing)

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

$$\frac{\begin{array}{c}\Gamma \mid - e : t_1 + t_2 \\ \Gamma, x_1 : t_1 \mid - e_1 : t \quad \Gamma, x_2 : t_2 \mid - e_2 : t\end{array}}{\Gamma \mid - \text{case e of inl } \mathrm{x}_1 \Rightarrow \mathrm{e}_1 \mid \text{inr } \mathrm{x}_2 \Rightarrow \mathrm{e}_2 : t} \quad \text{(T - Case)}$$

- (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:

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

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 $inl[t_1+t_2]\ e$,

  we use $in_1[t_1+t_2]\ e$.
- $e ::= .. \mid in_i\ e_i$
- $t ::= .. \mid t_1 + \ldots + t_n$
- 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
  $\to$ h h 3
  -- where h = \x. g (\y. x x y)
  $\to$ g fct 3
  -- where fct = \y. h h y (Notice we abuse "fct" a bit here.)
  $\to$ \n. if n=0 then 1 else n * (fct (n-1)) 3
  $\to$ if 3=0 then 1 else 3 * (fct (3-1))
  $\to^*$ 3 * (fct 2)
  $\to$ 3 * (h h 2)
  $\to$ 3 * (g fct 2)
  $\to^*$ 3 * 2 * (g fct 1)
  $\to^*$ 3 * 2 * 1 * (g fct 0)
  $\to^*$ 6

Recursion happens!

General Recursion

Syntax:

e ::= … expressions:

| fix e fix point of e

Evaluation:

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

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

Typing:

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

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

QUIZ

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

  hint: define a function ss: (int→int)→(int→ int)

  and then sum = fix ss.

Evaluation:

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

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

Typing:

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

LISTS

Quiz: Why do we need annotation of *t* here?

- 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)

$$\frac{}{\text{case nil of nil} \Rightarrow \text{e}_1 \mid \text{x}_1 :: \text{x}_2 \Rightarrow \text{e}_2 \rightarrow e_1} \text{(E-CaseNil)}$$

$$\frac{}{\text{case v}_1 :: \text{v}_2 \text{ of nil} \Rightarrow \text{e}_1 \mid \text{x}_1 :: \text{x}_2 \Rightarrow \text{e}_2 \rightarrow e_2[v_1 / x_1][v_2 / x_2]} \text{(E-CaseCons)}$$

$$\frac{\text{e} \rightarrow \text{e'}}{\begin{array}{l}\text{case e of nil} \Rightarrow \text{e}_1 \mid \text{x}_1 :: \text{x}_2 \Rightarrow \text{e}_2 \rightarrow \\ \text{case e' of nil} \Rightarrow \text{e}_1 \mid \text{x}_1 :: \text{x}_2 \Rightarrow \text{e}_2\end{array}} \text{(E-ListCase)}$$

$$\frac{\text{e}_1 \rightarrow \text{e}_1'}{\text{e}_1 :: \text{e}_2 \rightarrow \text{e}_1' :: \text{e}_2} \text{(E-Cons1)} \qquad \frac{\text{e}_2 \rightarrow \text{e}_2'}{\text{v}_1 :: \text{e}_2 \rightarrow \text{v}_1 :: \text{e}_2'} \text{(E-Cons2)}$$

List (Typing)

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

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

- 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 = \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 $x \neq x'$

- 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 ::= … | {\x.e, E}

- The new multi-step evaluation judgment:

  (E, e) →* v

Environment Model (evaluation)

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

$$\frac{(E, \mathrm{e}_1) \to^* \{\lambda x.e, \boxed{E_1}\} \quad (E, \mathrm{e}_2) \to^* v_2 \quad (\boxed{E_1}[x \mapsto v_2], \mathrm{e}) \to^* v,}{(E, (\mathrm{e}_1 \ \ \mathrm{e}_2)) \to^* v} \quad \text{(E - app)}$$

$$\frac{(E, \mathrm{e}_1) \to^* v_1 \quad (E[x \mapsto v_1], \mathrm{e}_2) \to^* v_2}{(E, \text{let x} = \mathrm{e}_1 \text{ in } \mathrm{e}_2) \to^* v_2} \quad \text{(E - let)}$$

- 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 = {\y. y+x, x↦1}

C_g = {\x. (f x) + 1, x↦1, f↦C_f}

(x↦1,f↦C_f, g↦C_g, g (f x)) →* …

(x↦1,f↦C_f, g↦C_g, f x) →* …

(x↦1,f↦C_f, g↦C_g, x) →* 1

(x↦1,y↦1, y+x) →* 2

(x↦1,f↦C_f, g↦C_g, g 2) →* …

(x↦2, f↦C_f, (f x) + 1)→* …

(x↦2, f↦C_f, f x) →* …

(x↦2, f↦C_f, x) →* 2

(x↦2, f↦C_f, f 2) →* …

(x↦1, y ↦2, y+x) →* 3

(x↦2, f↦C_f, 3 + 1)→* 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:

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

- By environment:

  $(.,\ (\backslash z.\backslash x.\ z + x)\ x\ 5) \rightarrow$

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

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

  $x + 5$