OCaml

案例

链表去重

元素个数统计及

```
# let encode list =
    List.map (fun l -> (List.length l, List.hd l)) (pack list);;
val encode : 'a list -> (int * 'a) list = <fun>

# encode ["a"; "a"; "a"; "a"; "b"; "c"; "c"; "a"; "a"; "d"; "e"; "e";
"e"];;
-: (int * string) list =
[(4, "a"); (1, "b"); (2, "c"); (2, "a"); (1, "d"); (4, "e")]
```

重复元素个数

```
# let replicate list n =
  let rec prepend n acc x =
    if n = 0 then acc else prepend (n-1) (x :: acc) x in
  let rec aux acc = function
  | [] -> acc
  | h :: t -> aux (prepend n acc h) t in
  (* This could also be written as:
       List.fold_left (prepend n) [] (List.rev list) *)
  aux [] (List.rev list);;

# replicate ["a"; "b"; "c"] 3;;
- : string list = ["a"; "a"; "a"; "b"; "b"; "b"; "c"; "c"; "c"]
```

链表元素排列组合

```
# let rec permutation list =
  let rec extract acc n = function
    | [] -> raise Not_found
    | h :: t -> if n = 0 then (h, acc @ t) else extract (h :: acc) (n - 1) t
    in
  let extract_rand list len =
      extract [] (Random.int len) list
  in
```

```
let rec aux acc list len =
    if len = 0 then acc else
    let picked, rest = extract_rand list len in
    aux (picked :: acc) rest (len - 1)
    in
    aux [] list (List.length list);;

# permutation ["a"; "b"; "c"; "d"; "e"; "f"];;
- : string list = ["a"; "e"; "f"; "b"; "d"; "c"]
```

质数检测

```
# let is_prime n =
let n = abs n in
let rec is_not_divisor d =
    d * d > n || (n mod d <> 0 && is_not_divisor (d + 1)) in
n <> 1 && is_not_divisor 2;;
```

二叉树

```
# type 'a binary_tree =
   Empty
    | Node of 'a * 'a binary_tree * 'a binary_tree;;
type 'a binary_tree = Empty | Node of 'a * 'a binary_tree * 'a binary_tree
# let example_tree =
    Node ('a', Node ('b', Node ('d', Empty, Empty), Node ('e', Empty, Empty)),
         Node ('c', Empty, Node ('f', Node ('g', Empty, Empty), Empty)));;
val example_tree : char binary_tree =
  Node ('a', Node ('b', Node ('d', Empty, Empty), Node ('e', Empty, Empty)),
   Node ('c', Empty, Node ('f', Node ('g', Empty, Empty), Empty)))
# let example_int_tree =
    Node (1, Node (2, Node (4, Empty, Empty), Node (5, Empty, Empty)),
         Node (3, Empty, Node (6, Node (7, Empty, Empty), Empty)));;
val example_int_tree : int binary_tree =
  Node (1, Node (2, Node (4, Empty, Empty), Node (5, Empty, Empty)),
   Node (3, Empty, Node (6, Node (7, Empty, Empty), Empty)))
```

修改变量

```
# let r = ref 0;;
val r : int ref = {contents = 0}
# r := 10;;
- : unit = ()
# !r;;
- : int = 10
# |
```

```
# type mypair = {a: int; b: int};;
type mypair = { a : int; b : int; }
# {a = 1; b = 2};;
- : mypair = {a = 1; b = 2}
# {a = 1};;
Error: Some record fields are undefined: b
# |
```

无类型算术表达式

语法树结点个数 size(t):

```
\begin{array}{lll} \textit{size}(\mathsf{true}) & = & 1 \\ \textit{size}(\mathsf{false}) & = & 1 \\ \textit{size}(\mathsf{0}) & = & 1 \\ \textit{size}(\mathsf{succ}\,\mathsf{t}_1) & = & \textit{size}(\mathsf{t}_1) + 1 \\ \textit{size}(\mathsf{pred}\,\mathsf{t}_1) & = & \textit{size}(\mathsf{t}_1) + 1 \\ \textit{size}(\mathsf{iszero}\,\mathsf{t}_1) & = & \textit{size}(\mathsf{t}_1) + 1 \\ \textit{size}(\mathsf{if}\,\mathsf{t}_1\,\mathsf{then}\,\mathsf{t}_2\,\mathsf{else}\,\mathsf{t}_3) & = & \textit{size}(\mathsf{t}_1) + \textit{size}(\mathsf{t}_2) + \textit{size}(\mathsf{t}_3) + 1 \end{array}
```

语法书结点深度:

```
\begin{array}{lll} \textit{depth}(\mathsf{true}) & = & 1 \\ \textit{depth}(\mathsf{false}) & = & 1 \\ \textit{depth}(0) & = & 1 \\ \textit{depth}(\mathsf{succ}\,\mathsf{t}_1) & = & \textit{depth}(\mathsf{t}_1) + 1 \\ \textit{depth}(\mathsf{pred}\,\mathsf{t}_1) & = & \textit{depth}(\mathsf{t}_1) + 1 \\ \textit{depth}(\mathsf{iszero}\,\mathsf{t}_1) & = & \textit{depth}(\mathsf{t}_1) + 1 \\ \textit{depth}(\mathsf{if}\,\mathsf{t}_1\,\mathsf{then}\,\mathsf{t}_2\,\mathsf{else}\,\mathsf{t}_3) & = & \max(\textit{depth}(\mathsf{t}_1), \textit{depth}(\mathsf{t}_2), \textit{depth}(\mathsf{t}_3)) + 1 \\ \end{array}
```

Lambda 演算

三种规约方式:

Full beta-reduction:

$$id (id (\lambda z. \underline{id z}))$$

$$\rightarrow id \underline{(id (\lambda z. z)})$$

$$\rightarrow \underline{id (\lambda z. z)}$$

$$\rightarrow \lambda z. z$$

$$\rightarrow$$

正则顺序:每次选择最左、最外侧的redex

$$\frac{id (id (\lambda z. id z))}{id (\lambda z. id z)}$$

$$\rightarrow \lambda z. \underline{id z}$$

$$\rightarrow \lambda z. z$$

$$\rightarrow \lambda z. z$$

传名: 严格从外侧归约, 函数抽象中不进行归约

$$\frac{id (id (\lambda z. id z))}{id (\lambda z. id z)}$$

$$\rightarrow \lambda z. id z$$

$$\rightarrow$$

传值: 函数参数先归约, 也就是说先参数求值再计算函数

$$id (id (\lambda z. id z))$$

$$\rightarrow id (\lambda z. id z)$$

$$\rightarrow \lambda z. id z$$

$$\rightarrow$$

Church encodes

$$tru = \lambda t. \lambda f. t;$$

 $fls = \lambda t. \lambda f. f;$
 $test = \lambda l. \lambda m. \lambda n. l m n;$

- $and = \lambda b. \lambda c. b \ c \ fls$
- $or = \lambda b. \lambda c. b tru c$
- $not = \lambda b.b fls tru$

```
pair = \lambda f. \lambda s. \lambda b. b f s;
fst = \lambda p. p tru;
snd = \lambda p. p fls;
```

```
c_0 = \lambda s. \lambda z. z;

c_1 = \lambda s. \lambda z. s z;

c_2 = \lambda s. \lambda z. s (s z);

c_3 = \lambda s. \lambda z. s (s (s z));

etc.
```

 $scc = \lambda n. \lambda s. \lambda z. s (n s z);$

- $plus = \lambda m. \lambda n. \lambda s. \lambda z. m s (n s z)$
- $times = \lambda m. \lambda n. m (plus n) c_0$

- $iszero = \lambda m.m (\lambda x. fls) tru$
- $zz = pair c_0 c_0$
- $ss = \lambda p. pair (snd p) (plus c_1 (snd p))$
- $prd = \lambda m. fst (m ss zz)$
- $sub = \lambda m. \lambda n. m prd n$

fix = λf . (λx . f (λy . x x y)) (λx . f (λy . x x y));

替换规则

$$[x \mapsto s]x = s$$

$$[x \mapsto s]y = y \qquad \text{if } y \neq x$$

$$[x \mapsto s](\lambda y.t_1) = \lambda y. [x \mapsto s]t_1 \qquad \text{if } y \neq x \text{ and } y \notin FV(s)$$

$$[x \mapsto s](t_1 t_2) = [x \mapsto s]t_1 [x \mapsto s]t_2$$

简单类型表达式

引理: 类型关系的倒置 (Inversion of the type relation)

- 1. If true: R, then R = Bool.
- 2. If false: R, then R = Bool.
- 3. If if t_1 then t_2 else t_3 : R, then t_1 : Bool, t_2 : R, and t_3 : R.
- 4. If 0 : R, then R = Nat.
- 5. If succ t_1 : R, then R = Nat and t_1 : Nat.
- 6. If pred t_1 : R, then R = Nat and t_1 : Nat.
- 7. If iszero t_1 : R, then R = Bool and t_1 : Nat.

性质:

可靠性/安全性/Soundness/Safty

- Well-typed terms求值不会出错
- 不会到达stuck state

由两个定理来保证可靠性:

• Progress: A well-typed term is not stuck. 要么是一个value,要么存在一条规则进行求值

• Preservation: 一个well-typed term经过若干次求值得到的新term也是well-typed的

例: 推断 (λx:Bool .x) true的类型

 $\frac{x:Bool \in x:Bool}{x:Bool \vdash x:Bool} \xrightarrow{T-VAR} \frac{}{\vdash \lambda x:Bool.x:Bool \rightarrow Bool} \xrightarrow{T-ABS} \frac{}{\vdash true:Bool} \xrightarrow{T-APP}$ $\vdash (\lambda x:Bool.x) true:Bool$

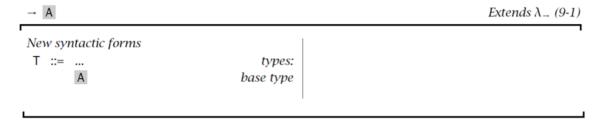
引理: 类型关系的倒置

- 1. If $\Gamma \vdash x : R$, then $x : R \in \Gamma$.
- 2. If $\Gamma \vdash \lambda x : T_1 \cdot t_2 : R$, then $R = T_1 \rightarrow R_2$ for some R_2 with $\Gamma, x : T_1 \vdash t_2 : R_2$.
- 3. If $\Gamma \vdash \mathsf{t}_1 \; \mathsf{t}_2 : \mathsf{R}$, then there is some type T_{11} such that $\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_{11} \to \mathsf{R}$ and $\Gamma \vdash \mathsf{t}_2 : \mathsf{T}_{11}$.
- 4. If $\Gamma \vdash \mathsf{true} : \mathsf{R}$, then $\mathsf{R} = \mathsf{Bool}$.
- 5. If $\Gamma \vdash \mathsf{false} : \mathsf{R}$, then $\mathsf{R} = \mathsf{Bool}$.
- 6. If $\Gamma \vdash \text{if } \mathsf{t}_1 \text{ then } \mathsf{t}_2 \text{ else } \mathsf{t}_3 : \mathsf{R}, \text{ then } \Gamma \vdash \mathsf{t}_1 : \mathsf{Bool} \text{ and } \Gamma \vdash \mathsf{t}_2, \mathsf{t}_3 : \mathsf{R}$. \square

简单扩展

基本类型

使用抽象的A,B,C这样的符号表示不同的基本类型



例:

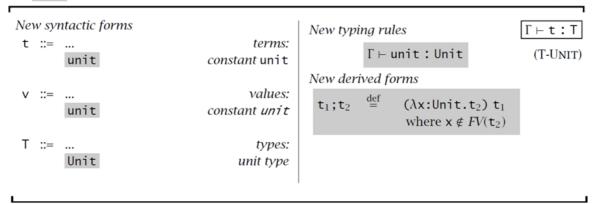
λχ:Α. χ

λx:B. x

 $\lambda f: A \rightarrow A. \ \lambda x: A. \ f(f(x))$

The Unit Types

 \rightarrow Unit Extends λ_{-} (9-1)

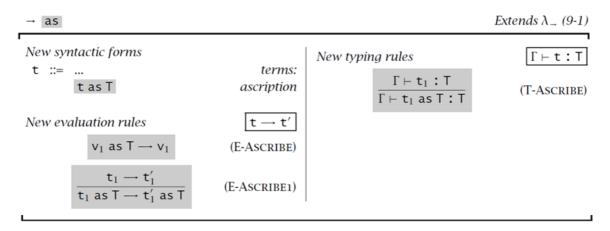


 $t_1;t_2$ 可以看作 $(\lambda x:Unit.\ t_2)t_1$ 的缩写,是一种派生型(derived form),又称为语法糖(syntactic sugar)

通配符(Wildcard)

- 一种语法糖
- 表示不会在函数抽象中使用到的参数
- 用通配符"_"表示这样的参数
- $\lambda_-: S.t \neq \lambda x: S.t$ 的缩写,其中 x 不在 t 中出现

类型归属(Ascription)



Let 绑定 (Let Bindings)

$$- let x = t_1 in t_2 \stackrel{\text{def}}{=} (\lambda x: T_1.t_2) t_1?$$

$$- T_1 的类型从type checker得到$$

$$\frac{\vdots}{\frac{\Gamma + \mathsf{t}_1 : \mathsf{T}_1}{\Gamma + \mathsf{t}_1 : \mathsf{T}_1}} \frac{\vdots}{\frac{\Gamma, x : \mathsf{T}_1 + \mathsf{t}_2 : \mathsf{T}_2}{\Gamma + \mathsf{let} \ x = \mathsf{t}_1 \ in \ \mathsf{t}_2 : \mathsf{T}_2}} \underbrace{\mathsf{T-Let}}_{\Gamma - \mathsf{Let}}$$

$$\frac{\vdots}{\frac{\Gamma, x : \mathsf{T}_1 + \mathsf{t}_2 : \mathsf{T}_2}{\Gamma + \lambda x : \mathsf{T}_1.\mathsf{t}_2 : \mathsf{T}_1 \to \mathsf{T}_2}} \underbrace{\mathsf{T-ABS}}_{\Gamma + \mathsf{L}_1 : \mathsf{T}_1} \frac{\vdots}{\Gamma + \mathsf{L}_1 : \mathsf{T}_1}}_{\Gamma - \mathsf{LPD}}$$

Pairs

Tuples

 \rightarrow {}

 $\frac{\mathsf{t}_1 \longrightarrow \mathsf{t}'_1}{\mathsf{t}_1 \cdot \mathsf{i} \longrightarrow \mathsf{t}'_1 \cdot \mathsf{i}}$ $\frac{\mathsf{t}_j \longrightarrow \mathsf{t}'_j}{\{\mathsf{V}_i^{i \in I..j-l}, \mathsf{t}_j, \mathsf{t}_k^{k \in j+l.n}\}}$ $\longrightarrow \{\mathsf{V}_i^{i \in I..j-l}, \mathsf{t}'_j, \mathsf{t}_k^{k \in j+l.n}\}$ (E-Tuple)

New typing rules $\Gamma \vdash t : T$

 $\frac{\text{for each } i \quad \Gamma \vdash \mathsf{t}_i : \mathsf{T}_i}{\Gamma \vdash \{\mathsf{t}_i^{i \in l..n}\} : \{\mathsf{T}_i^{i \in l..n}\}}$ (T-TUPLE)

 $\frac{\Gamma \vdash \mathsf{t}_1 : \{\mathsf{T}_i^{\ i \in I..n}\}}{\Gamma \vdash \mathsf{t}_1 . \, \mathsf{j} : \mathsf{T}_j} \tag{T-Proj}$

记录

 \rightarrow {}

New syntactic forms $t ::= ... \qquad terms: \\ \{1_i = t_i^{i \in I..n}\} \qquad record \\ t.1 \qquad projection$

V ::= ... values: $\{1_i = V_i \stackrel{i \in I...n}{=} \}$ record value

T ::= ... types: $\{1_i: T_i \stackrel{i \in I..n}{=} \}$ type of records

 $\frac{\mathsf{t}_1 \longrightarrow \mathsf{t}_1'}{\mathsf{t}_1.\mathsf{1} \longrightarrow \mathsf{t}_1'.\mathsf{1}} \tag{E-Proj}$

 $\frac{\mathsf{t}_{j} \longrightarrow \mathsf{t}'_{j}}{\{\mathsf{l}_{i} = \mathsf{V}_{i} \stackrel{i \in I...j-1}{,} \mathsf{l}_{j} = \mathsf{t}_{j}, \mathsf{l}_{k} = \mathsf{t}_{k} \stackrel{k \in j+I.n}{,} \}}{\longrightarrow \{\mathsf{l}_{i} = \mathsf{V}_{i} \stackrel{i \in I...j-1}{,} \mathsf{l}_{j} = \mathsf{t}'_{j}, \mathsf{l}_{k} = \mathsf{t}_{k} \stackrel{k \in j+I.n}{,} \}}$ (E-RCD)

New typing rules $\Gamma \vdash t : T$

 $\frac{\text{for each } i \quad \Gamma \vdash \mathsf{t}_i : \mathsf{T}_i}{\Gamma \vdash \{\mathsf{1}_i = \mathsf{t}_i \ ^{i \in 1..n}\} : \{\mathsf{1}_i : \mathsf{T}_i \ ^{i \in 1..n}\}}$ (T-RCD)

 $\frac{\Gamma \vdash \mathsf{t}_1 : \{\mathsf{l}_i : \mathsf{T}_i \stackrel{i \in I..n}{=}\}}{\Gamma \vdash \mathsf{t}_1 . \mathsf{l}_j : \mathsf{T}_j} \tag{T-Proj}$

Sums

 \rightarrow + Extends λ_{-} (11-9)

New syntactic forms
t ::= ...

v ::= ... values: inl v as T tagged value (left) inr v as T tagged value (right)

New evaluation rules $t \rightarrow t'$

case (inl v_0 as T_0) of inl $x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2$ (E-CASEINL) $\longrightarrow [x_1 \mapsto v_0]t_1$ case (inr v_0 as T_0) of inl $x_1 \Rightarrow t_1 \mid \text{inr } x_2 \Rightarrow t_2$ (E-CASEINR) $\rightarrow [x_2 \mapsto v_0]t_2$

 $\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{inl} \; \texttt{t}_1 \; \texttt{as} \; \texttt{T}_2 \; \longrightarrow \; \texttt{inl} \; \texttt{t}_1' \; \texttt{as} \; \texttt{T}_2} \tag{E-INL}$

 $\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{inr}\, \texttt{t}_1 \, \texttt{as}\, \mathsf{T}_2 \, \longrightarrow \, \texttt{inr}\, \texttt{t}_1' \, \texttt{as}\, \mathsf{T}_2} \qquad (\text{E-INR})$

New typing rules $\Gamma \vdash t : T$

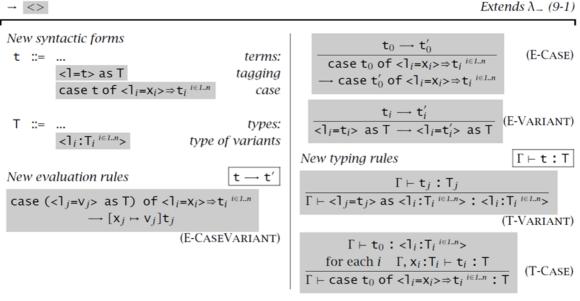
 $\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_1}{\Gamma \vdash \mathsf{inl} \; \mathsf{t}_1 \; \mathsf{as} \; \mathsf{T}_1 + \mathsf{T}_2 \; : \; \mathsf{T}_1 + \mathsf{T}_2} \tag{T-INL}$

 $\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_2}{\Gamma \vdash \mathsf{inr} \; \mathsf{t}_1 \; \mathsf{as} \; \mathsf{T}_1 + \mathsf{T}_2 \; : \; \mathsf{T}_1 + \mathsf{T}_2} \tag{T-INR}$

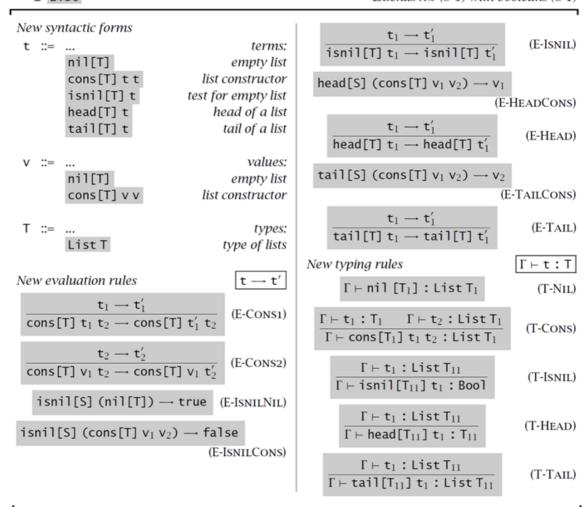
Variants

将sum types加上标签,可以泛化为variants

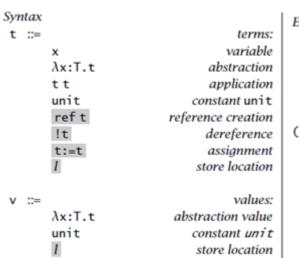
$$T_1 + T_2 \Longrightarrow < l_1: T_1, l_2: T_2 >$$
 $inl \ t \ as \ T_1 + T_2 \Longrightarrow < l_1 = t > as < l_1: T_1, l_2: T_2 >$
 \longrightarrow



链表







$$\begin{array}{ccc} \Gamma & ::= & & \textit{contexts:} \\ & \oslash & & \textit{empty context} \\ & \Gamma, x{:}T & \textit{term variable binding} \end{array}$$

$$\mu$$
 ::= stores:
 \varnothing empty store
 $\mu, l = v$ location binding

$$\Sigma$$
 ::= store typings:
 \varnothing empty store typing
 Σ , l :T location typing

$$\frac{\mathsf{t}_1 \mid \mu \longrightarrow \mathsf{t}_1' \mid \mu'}{\mathsf{t}_1 \; \mathsf{t}_2 \mid \mu \longrightarrow \mathsf{t}_1' \; \mathsf{t}_2 \mid \mu'} \tag{E-APP1}$$

$$\frac{\mathsf{t}_2 \mid \mu \longrightarrow \mathsf{t}_2' \mid \mu'}{\mathsf{v}_1 \; \mathsf{t}_2 \mid \mu \longrightarrow \mathsf{v}_1 \; \mathsf{t}_2' \mid \mu'} \tag{E-App2}$$

$$(\lambda x: T_{11}.t_{12}) v_2 \mid \mu \longrightarrow [x \mapsto v_2]t_{12} \mid \mu$$
(E-AppAbs)

$$\frac{l \notin dom(\mu)}{\mathsf{ref}\,\mathsf{v}_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto \mathsf{v}_1)} \tag{E-RefV}$$

$$\frac{\mathsf{t}_1 \mid \mu \longrightarrow \mathsf{t}_1' \mid \mu'}{\mathsf{ref}\,\mathsf{t}_1 \mid \mu \longrightarrow \mathsf{ref}\,\mathsf{t}_1' \mid \mu'} \tag{E-Ref}$$

$$\frac{\mu(l) = \mathsf{v}}{!l \mid \mu \longrightarrow \mathsf{v} \mid \mu} \tag{E-DerefLoc}$$

$$\frac{\mathsf{t}_1 \mid \mu \longrightarrow \mathsf{t}_1' \mid \mu'}{\mathsf{!t}_1 \mid \mu \longrightarrow \mathsf{!t}_1' \mid \mu'}$$
 (E-Deref)

$$l:=v_2 \mid \mu \longrightarrow unit \mid [l \mapsto v_2]\mu$$
 (E-Assign)

$$\frac{\mathsf{t}_1 \mid \mu \longrightarrow \mathsf{t}_1' \mid \mu'}{\mathsf{t}_1 \colon = \mathsf{t}_2 \mid \mu \longrightarrow \mathsf{t}_1' \colon = \mathsf{t}_2 \mid \mu'} \quad \text{(E-Assign1)}$$

$$\frac{\mathsf{t}_2 \mid \mu \longrightarrow \mathsf{t}_2' \mid \mu'}{\mathsf{v}_1 \colon= \! \mathsf{t}_2 \mid \mu \longrightarrow \mathsf{v}_1 \colon= \! \mathsf{t}_2' \mid \mu'} \quad \text{(E-Assign2)}$$

continued...

Typing
$$\begin{array}{c|c} x:T \in \Gamma \\ \hline \Gamma \mid \Sigma \vdash x:T \\ \hline \Gamma, x:T_1 \mid \Sigma \vdash t_2:T_2 \\ \hline \end{array}$$
 (T-VAR)

$$\frac{\Gamma \mid \Sigma \vdash \lambda x : T_1 \cdot t_2 : T_1 \rightarrow T_2}{\Gamma \mid \Sigma \vdash t_1 : T_{11} \rightarrow T_{12}} \frac{\Gamma \mid \Sigma \vdash t_2 : T_{11}}{\Gamma \mid \Sigma \vdash t_1 : t_2 : T_{12}}$$

(T-App)

$$\Gamma \mid \Sigma \vdash \mathsf{unit} : \mathsf{Unit}$$
 (T-UNIT)

$$\frac{\Sigma(l) = \mathsf{T}_1}{\Gamma \mid \Sigma \vdash l : \mathsf{Ref} \, \mathsf{T}_1} \tag{T-Loc}$$

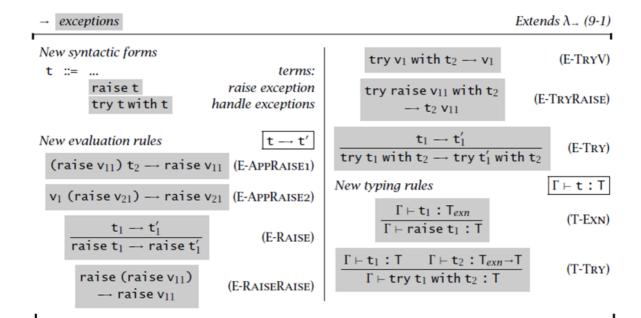
$$\frac{\Gamma \mid \Sigma \vdash \mathsf{t}_1 : \mathsf{T}_1}{\Gamma \mid \Sigma \vdash \mathsf{ref} \; \mathsf{t}_1 : \mathsf{Ref} \; \mathsf{T}_1} \tag{T-Ref}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathsf{t}_1 : \mathsf{Ref} \, \mathsf{T}_{11}}{\Gamma \mid \Sigma \vdash ! \, \mathsf{t}_1 : \mathsf{T}_{11}} \tag{T-Deref}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathsf{t}_1 : \mathsf{Ref}\,\mathsf{T}_{11} \qquad \Gamma \mid \Sigma \vdash \mathsf{t}_2 : \mathsf{T}_{11}}{\Gamma \mid \Sigma \vdash \mathsf{t}_1 := \mathsf{t}_2 : \mathsf{Unit}}$$

(T-Assign)

异常处理



子类型

性质

引理:

- 如果 $S <: T_1 \to T_2$, 那么 S 具有 $S_1 \to S_2$ 的形式,其中 $T_1 <: S_1$, $S_2 <: T_2$ 。
 如果 $S <: \{l_i : T_i^{(i \in 1..n)}\}$,那么S具有 $\{k_j : S_j^{(j \in 1..m)}\}$ 的形式,其中 $\{l_i^{(i \in 1..n)}\} \subseteq \{k_j^{(j \in 1..m)}\}$,且对每个共同的标签 $l_i=k_j$ 都有 $S_j<:T_i$ 。

利用运算符µ表示递归与重复的部分

【例】

- $NatList = \mu X. \langle nil : Unit, const \{Nat, X\} \rangle;$
- 令 NatList 为一个无穷类型,满足方程 $X = < nil : Unit, const \{Nat, X\} >$

```
sumlist = fix (\lambdas:NatList\rightarrowNat. \lambdal:NatList.
                     if isnil 1 then 0 else plus (hd 1) (s (tl 1)));
```

► sumlist : NatList → Nat

【例】

一个hungry函数,接收一个参数,然后返回一个函数,返回的函数仍然是一个hungry函数: $Hungry = \mu A. Nat \rightarrow A;$

```
f = fix (\lambda f: Nat \rightarrow Hungry. \lambda n: Nat. f);
   ► f : Hungry
      f 0 1 2 3 4 5;
   ► <fun> : Hungry
【例】
Stream = \mu A. Unit \rightarrow \{Nat, A\};
     hd = \lambdas:Stream. (s unit).1;
   ► hd : Stream → Nat
     tl = \lambda s:Stream. (s unit).2;
   ► tl : Stream → Stream
     upfrom0 = fix (\lambdaf: Nat\rightarrowStream. \lambdan:Nat. \lambda_:Unit. {n,f (succ n)}) 0;
   ► upfrom0 : Stream
     hd upfrom0;
   ▶ 0 : Nat
     hd (tl (tl upfrom0)));
   ▶ 3 : Nat
    - Streams可以用来表示进程
     - Process = \mu A. Nat \rightarrow \{Nat, A\};
        p = fix (\lambda f: Nat \rightarrow Process. \lambda acc: Nat. \lambda n: Nat.
                       let newacc = plus acc n in
                       {newacc, f newacc}) 0;
      ▶ p : Process
        curr = \lambdas:Process. (s 0).1;
      ► curr : Process → Nat
        send = \lambda n:Nat. \lambda s:Process. (s n).2;
      ▶ send : Nat → Process → Process
```

●对象

- 数据及数据上的操作
- 纯函数式

```
Counter = \muC. {get:Nat, inc:Unit\rightarrowC, dec:Unit\rightarrowC};

c = let create = fix (\lambdaf: {x:Nat}\rightarrowCounter. \lambdas: {x:Nat}.

{get = s.x,

inc = \lambda_:Unit. f {x=succ(s.x)},

dec = \lambda_:Unit. f {x=pred(s.x)}})

in create {x=0};

c1 = c.inc unit;

c2 = c1.inc unit;

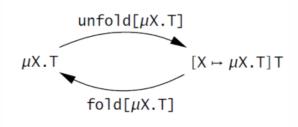
c2.get;

let 2 : Nat
```

形式化定义

- 增加一对函数unfold和fold

unfold[μ X.T] : μ X.T \rightarrow [X \mapsto μ X.T]T fold[μ X.T] : [X \mapsto μ X.T]T \rightarrow μ X.T



【例】

 $\mu X.< nil:Unit,cons:\{Nat,X\}>,$

unfolds to

<nil:Unit, cons:{Nat, μ X.<nil:Unit,cons:{Nat,X}>}>.

多态

类型变量与替换。为了实现多态,可以使用一些变量作为占位符,需要的时候再替换成实际的类型。 类型的替换由两部分组成:

声明一个映射σ,将类型变量映射到具体的类型

• 应用一次映射T,得到一个实例σ T

【例】

$$egin{aligned} \sigma &= [X \mapsto Bool, Y \mapsto Nat, Z \mapsto Nat
ightarrow Bool] \ \sigma(X
ightarrow X) &= Bool
ightarrow Bool \end{aligned}$$

System F

【例】多态链表

▶ nil : ∀X. List X

▶ 7 : Nat

cons : $\forall X. X \rightarrow List X \rightarrow List X$

isnil : $\forall X$. List $X \rightarrow Bool$

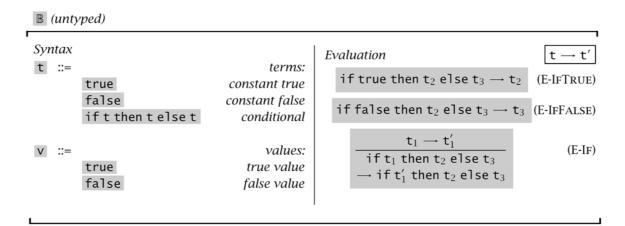
head : $\forall X$. List $X \rightarrow X$

tail : $\forall X$. List $X \rightarrow List X$

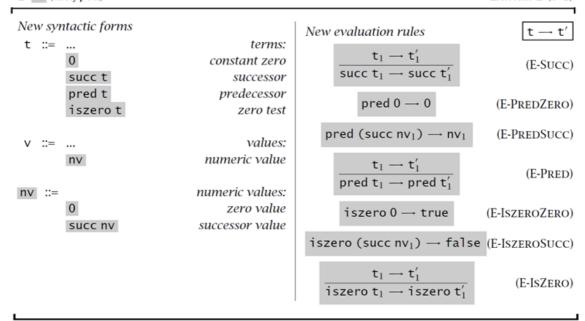
```
map = \lambda X. \lambda Y.
               \lambda f: X \rightarrow Y.
                  (fix (\lambda m: (List X) \rightarrow (List Y).
                             \lambda1: List X.
                                if isnil [X] 1
                                   then nil [Y]
                                   else cons [Y] (f (head [X] 1))
                                                        (m (tail [X] 1)));
 ▶ map : \forall X. \forall Y. (X \rightarrow Y) \rightarrow List X \rightarrow List Y
    1 = cons [Nat] 4 (cons [Nat] 3 (cons [Nat] 2 (nil [Nat])));
 ▶ 1 : List Nat
    head [Nat] (map [Nat] [Nat] (\lambda x:Nat. succ x) 1);
 ▶ 5 : Nat
【例】二叉树
          Tree = \mu X. <leaf:Unit, node:{Nat,X,X}>;
          leaf = <leaf=unit> as Tree;
        ▶ leaf : Tree
          node = \lambda n:Nat. \lambda t_1:Tree. \lambda t_2:Tree. < node={n,t_1,t_2}> as Tree;
        ▶ node : Nat → Tree → Tree → Tree
          isleaf = \lambda1:Tree. case 1 of <1eaf=u> \Rightarrow true | <node=p> \Rightarrow false;
        ▶ isleaf : Tree → Bool
          label = \lambda1:Tree. case 1 of <leaf=u> \Rightarrow 0 | <node=p> \Rightarrow p.1;
        ▶ label : Tree → Nat
          left = \lambda1:Tree. case 1 of <leaf=u> \Rightarrow leaf | <node=p> \Rightarrow p.2;
        ▶ left : Tree → Tree
          right = \lambda1:Tree. case 1 of <1eaf=u> \Rightarrow 1eaf | <node=p> \Rightarrow p.3;
        ▶ right : Tree → Tree
          append = fix (\lambda f: NatList \rightarrow NatList \rightarrow NatList.
                             \lambda11:NatList. \lambda12:NatList.
                              if isnil 11 then 12 else
                              cons (hd 11) (f (t1 11) 12));
```

```
► append : NatList → NatList
  preorder = fix (\lambdaf:Tree\rightarrowNatList. \lambdat:Tree.
                    if isleaf t then nil else
                     cons (label t)
                          (append (f (left t)) (f (right t))));
▶ preorder : Tree → NatList
  t1 = node 1 leaf leaf;
  t2 = node 2 leaf leaf;
 t3 = node 3 t1 t2;
  t4 = node 4 t3 t3;
  1 = preorder t4;
 hd 1;
▶ 4 : Nat
 hd (t1 1);
▶ 3 : Nat
 hd (tl (tl l));
▶ 1 : Nat
```

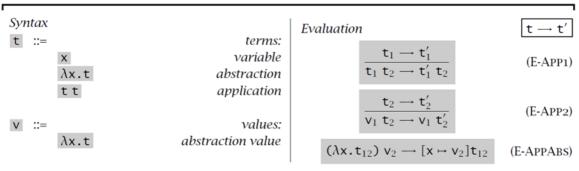
求值规则

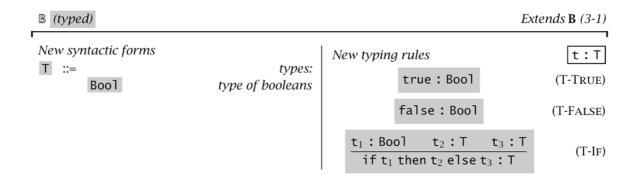


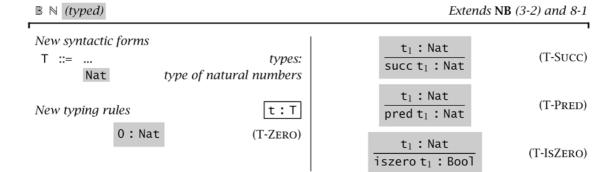




→ (untyped)

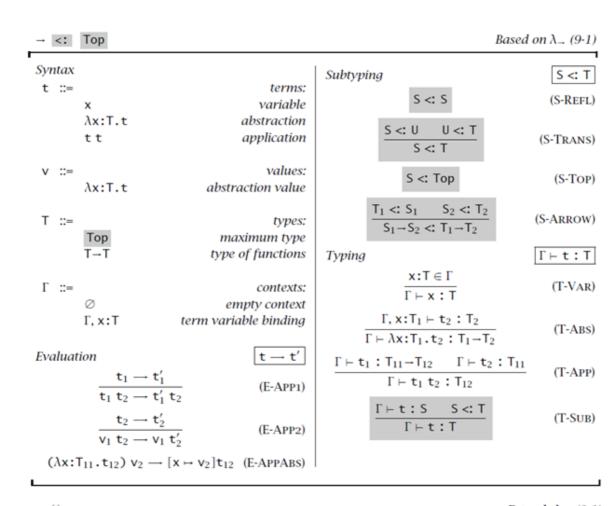


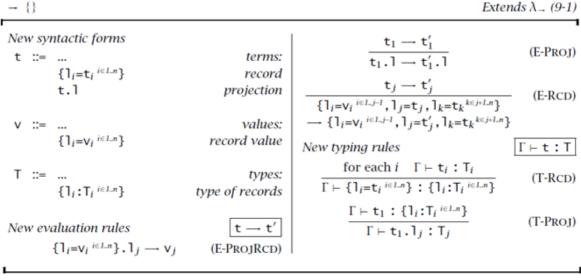


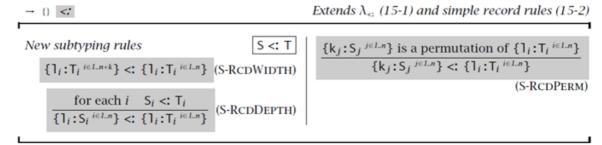


Based on λ (5-3) → (typed) Syntax Evaluation t ::= terms: $\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1 \ \mathtt{t}_2 \longrightarrow \mathtt{t}_1' \ \mathtt{t}_2}$ variable (E-APP1) $\lambda x : T.t$ abstraction $\frac{\texttt{t}_2 \longrightarrow \texttt{t}_2'}{\texttt{v}_1 \; \texttt{t}_2 \longrightarrow \texttt{v}_1 \; \texttt{t}_2'}$ application t t (E-APP2) values: v ::= $(\lambda x : T_{11} . t_{12}) v_2 \rightarrow [x \mapsto v_2] t_{12}$ (E-APPABS) $\lambda x : T.t$ abstraction value **Typing** $\Gamma \vdash \mathsf{t} : \mathsf{T}$ T ::= types: $x\!:\!T\in\Gamma$ $T \rightarrow T$ type of functions (T-VAR) $\Gamma \vdash x : \mathsf{T}$ Γ ::= contexts: Γ , x: $T_1 \vdash t_2$: T_2 empty context (T-ABS) $\overline{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2}$ $\Gamma, x:T$ term variable binding $\underline{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_{11} \!\rightarrow\! \mathsf{T}_{12} \qquad \Gamma \vdash \mathsf{t}_2 : \mathsf{T}_{11}}$ (T-APP) $\Gamma \vdash \mathsf{t}_1 \; \mathsf{t}_2 : \mathsf{T}_{12}$

子类型







Iso-recursive

 $\rightarrow \mu$ Extends λ_{-} (9-1)

约束求解

$$\frac{x:\mathsf{T}\in\Gamma}{\Gamma \vdash x:\mathsf{T} \mid_{\varnothing}\{\}} \qquad (\mathsf{CT-VAR}) \qquad \frac{\Gamma \vdash \mathsf{t}_1:\mathsf{T} \mid_{X} C}{\Gamma \vdash x:\mathsf{T}_1 \vdash \mathsf{t}_2:\mathsf{T}_2 \mid_{X} C} \qquad (\mathsf{CT-ABS}) \qquad \frac{\Gamma \vdash \mathsf{t}_1:\mathsf{T} \mid_{X} C}{\Gamma \vdash \mathsf{t}_1:\mathsf{T}_1 \mid_{X_1} C_1 \qquad \Gamma \vdash \mathsf{t}_2:\mathsf{T}_2 \mid_{X} C} \qquad (\mathsf{CT-ABS}) \qquad \Gamma \vdash \mathsf{t}_1:\mathsf{T} \mid_{X_1} C_1 \qquad \Gamma \vdash \mathsf{t}_2:\mathsf{T}_2 \mid_{X_2} C_2 \qquad \Gamma \vdash \mathsf{t}_1:\mathsf{T} \mid_{X_1} C_1 \qquad \Gamma \vdash \mathsf{t}_2:\mathsf{T}_2 \mid_{X_2} C_2 \qquad \Gamma \vdash \mathsf{t}_1:\mathsf{T} \mid_{X_1} C_1 \qquad \Gamma \vdash \mathsf{t}_2:\mathsf{T}_2 \mid_{X_2} C_2 \qquad \Gamma \vdash \mathsf{t}_1:\mathsf{T} \mid_{X_1} C_1 \qquad (\mathsf{CT-ISZERO}) \qquad \Gamma \vdash \mathsf{t}_1:\mathsf{T} \mid_{X_2} C_1, C_2, \Gamma, \mathsf{t}_1, \text{ or } \mathsf{t}_2 \qquad \Gamma \vdash \mathsf{t}_1:\mathsf{T} \mid_{X_1} C_1 \qquad \Gamma \vdash \mathsf{t}_2:\mathsf{T}_2 \mid_{X_2} C_2 \qquad \Gamma \vdash \mathsf{t}_3:\mathsf{T}_3 \mid_{X_3} C_3 \qquad X_1, X_2, X_3 \text{ nonoverlapping} \qquad C' = C_1 \cup C_2 \cup C_3 \cup \{\mathsf{T}_1 = \mathsf{Bool}, \mathsf{T}_2 = \mathsf{T}_3\} \qquad \Gamma \vdash \mathsf{if} \mathsf{t}_1 \mathsf{then} \mathsf{t}_2 \mathsf{else} \mathsf{t}_3:\mathsf{T}_2 \mid_{X_1 \cup X_2 \cup X_3} C' \qquad \Gamma \vdash \mathsf{if} \mathsf{t}_1 \mathsf{then} \mathsf{t}_2 \mathsf{else} \mathsf{t}_3:\mathsf{T}_2 \mid_{X_1 \cup X_2 \cup X_3} C' \qquad (\mathsf{CT-IF})$$

```
unify(C) = \text{ if } C = \emptyset, \text{ then } [\ ]
\text{else let } \{S = T\} \cup C' = C \text{ in }
\text{ if } S = T
\text{ then } unify(C')
\text{else if } S = X \text{ and } X \notin FV(T)
\text{ then } unify([X \mapsto T]C') \circ [X \mapsto T]
\text{else if } T = X \text{ and } X \notin FV(S)
\text{ then } unify([X \mapsto S]C') \circ [X \mapsto S]
\text{else if } S = S_1 \rightarrow S_2 \text{ and } T = T_1 \rightarrow T_2
\text{ then } unify(C' \cup \{S_1 = T_1, S_2 = T_2\})
\text{else}
fail
```

 $\rightarrow \forall$ Based on λ_{-} (9-1)

Syntax t ::= terms: variable λx:T.t abstraction t t application λX.t type abstraction type application t [T] values: v ::= λx:T.t abstraction value λX.t type abstraction value

 $\begin{array}{cccc} \Gamma & ::= & & \textit{contexts:} \\ & \varnothing & & \textit{empty context} \\ & \Gamma, x \colon T & \textit{term variable binding} \\ & \Gamma, X & \textit{type variable binding} \end{array}$

Evaluation $t \rightarrow t'$

$$\frac{\texttt{t}_1 \, \longrightarrow \, \texttt{t}_1'}{\texttt{t}_1 \, \texttt{t}_2 \, \longrightarrow \, \texttt{t}_1' \, \texttt{t}_2} \tag{E-App1)$$

$$\frac{\mathsf{t}_2 \longrightarrow \mathsf{t}_2'}{\mathsf{v}_1 \; \mathsf{t}_2 \longrightarrow \mathsf{v}_1 \; \mathsf{t}_2'} \tag{E-APP2}$$

$$(\lambda x:T_{11}.t_{12}) v_2 \rightarrow [x \mapsto v_2]t_{12}$$
 (E-APPABS)

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1 \ [\mathtt{T}_2] \longrightarrow \mathtt{t}_1' \ [\mathtt{T}_2]} \tag{E-TAPP}$$

$$(\lambda X.t_{12})$$
 $[T_2] \rightarrow [X \mapsto T_2]t_{12}$ (E-TAPPTABS)

Typing
$$\Gamma \vdash t : T$$

$$\frac{\mathsf{x} : \mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{x} : \mathsf{T}} \tag{T-VAR}$$

$$\frac{\Gamma, x \colon\! T_1 \vdash t_2 \colon T_2}{\Gamma \vdash \lambda x \colon\! T_1 \cdot t_2 \colon T_1 \!\rightarrow\! T_2} \tag{T-Abs}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_{11} \! \rightarrow \! \mathsf{T}_{12} \qquad \Gamma \vdash \mathsf{t}_2 : \mathsf{T}_{11}}{\Gamma \vdash \mathsf{t}_1 \: \mathsf{t}_2 : \mathsf{T}_{12}} \qquad \text{(T-APP)}$$

$$\frac{\Gamma, \mathsf{X} \vdash \mathsf{t}_2 : \mathsf{T}_2}{\Gamma \vdash \lambda \mathsf{X}. \mathsf{t}_2 : \forall \mathsf{X}. \mathsf{T}_2} \tag{T-TABS}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \forall \mathsf{X}.\mathsf{T}_{12}}{\Gamma \vdash \mathsf{t}_1 \; [\mathsf{T}_2] : [\mathsf{X} \mapsto \mathsf{T}_2]\mathsf{T}_{12}} \tag{T-TAPP}$$