



---

## IMPLEMENTAZIONE DELL'INTERPRETE DI UN NUCLEO DI LINGUAGGIO FUNZIONALE

1



---

## Linguaggio funzionale didattico

- ✎ Consideriamo un nucleo di un linguaggio funzionale
  - Sottosinsieme di ML senza tipi né pattern matching
- ✎ Obiettivo: esaminare tutti gli aspetti relativi alla implementazione dell'interprete e del supporto a run time per il linguaggio

2

# Linguaggio funzionale didattico



```
type ide = string
type exp = Eint of int
         | Ebool of bool
         | Den of ide
         | Prod of exp * exp
         | Sum of exp * exp
         | Diff of exp * exp
         | Eq of exp * exp
         | Minus of exp
         | Iszero of exp
         | Or of exp * exp
         | And of exp * exp
         | Not of exp
         | Ifthenelse of exp * exp * exp
         | Let of ide * exp * exp      (* Dichiarazione di ide: modifica ambiente *)
         | Fun of ide list * exp    (* Astrazione di funzione *)
         | Appl of exp * exp list  (* Applicazione di funzione *)
         | Rec of ide * exp        (* Ricorsione *)
```

3

## Blocchi-inline (Let)



- ✎ Con il **Let** possiamo cambiare l'ambiente in punti arbitrari all'interno di una espressione
  - facendo sì che l'ambiente "nuovo" valga soltanto durante la valutazione del "corpo del blocco", l'espressione **e2**
  - lo stesso nome può denotare entità distinte in blocchi diversi
- ✎ I blocchi possono essere annidati
  - e l'ambiente locale di un blocco più esterno può essere (in parte) visibile ed utilizzabile nel blocco più interno
    - ✓ come ambiente non locale!
- ✎ Come abbiamo visto, il blocco porta naturalmente a
  - ✓ una semplice gestione dinamica della memoria locale (stack dei record di attivazione)
- si sposa felicemente con la regola di scoping statico
  - ✓ per la gestione dell'ambiente non locale

4

## Let: Semantica operativa



$$\frac{env \triangleright e1 \Rightarrow v1 \quad env[v1/x] \triangleright e2 \Rightarrow v2}{env \triangleright Let(x, e1, e2) \Rightarrow v2}$$

5

```
let rec sem ((e:exp), (r:eval env)) =
  match e with
  | Eint(n) -> Int(n)
  | Ebool(b) -> Bool(b)
  | Den(i) -> applyenv(r,i)
  | Iszero(a) -> iszero(sem(a, r))
  | Eq(a,b) -> equ(sem(a, r),sem(b, r))
  | Prod(a,b) -> mult(sem(a, r), sem(b, r))
  | Sum(a,b) -> plus(sem(a, r), sem(b, r))
  | Diff(a,b) -> diff(sem(a, r), sem(b, r))
  | Minus(a) -> minus(sem(a, r))
  | And(a,b) -> et(sem(a, r), sem(b, r))
  | Or(a,b) -> vel(sem(a, r), sem(b, r))
  | Not(a) -> non(sem(a, r))
  | Ifthenelse(a,b,c) -> let g = sem(a, r) in
    if typecheck("bool",g) then
      (if g = Bool(true) then sem(b, r) else sem(c, r))
    else failwith ("nonboolean guard")
  | Let(i,e1,e2) ->
    sem(e2, bind (r ,i, sem(e1, r)))
```



6

## Analisi



```
let rec sem ((e:exp), (r:eval env)) =  
  match e with  
  ....  
  | Let(i,e1,e2) -> sem(e2, bind (r ,i, sem(e1, r)))
```

L'espressione **e2** (corpo del blocco) è valutata nell'ambiente "esterno" esteso con l'associazione tra il nome **i** ed il valore di **e1**

7

## Esempio di valutazione



```
# sem(Let("x", Sum(Eint 1, Eint 0),  
  Let("y", Ifthenelse(Eq(Den "x", Eint 0),  
    Diff(Den "x", Eint 1),  
    Sum(Den "x", Eint 1)),  
  Let("z", Sum(Den "x", Den "y"), Den "z"))),  
  (emptyenv Unbound));;
```

```
-: eval = Int 3
```

### Sintassi OCaml corrispondente

```
# let x = 1+0 in  
  let y = if x = 0 then x-1 else x+1 in  
    let z = x + y in z;;  
-: int = 3
```

8

# Funzioni



- Passiamo ora a esaminare gli ingredienti fondamentali della programmazione nel paradigma funzionale
  - Astrazione funzionale
  - Applicazione di funzione

9

# Sintassi



```
type exp = ...  
  | Fun of ide list * exp  
  | Appl of exp * exp list  
  | Rec of ide * exp
```

Astrazione

Applicazione

Ricorsione

10



## Funzioni

- Identificatori (**parametri formali**) nel costrutto di **astrazione**  
**Fun of ide list \* exp**
- Espressioni (**parametri attuali**) nel costrutto di **applicazione**  
**Appl of exp \* exp list**
- Per ora non ci occupiamo di modalità di passaggio dei parametri
  - le espressioni parametro attuale sono valutate (**eval** oppure **dval**) e i valori ottenuti sono legati nell'ambiente al corrispondente parametro formale
- Per ora ignoriamo il costrutto **Rec** (funzioni ricorsive)
- Con l'introduzione delle funzioni, il linguaggio funzionale è completo
  - un linguaggio funzionale reale (tipo ML) ha in più i tipi, il pattern matching e le eccezioni

11



## Giochiamo con la semantica (1)

- Come bisogna estendere i tipi esprimibili (**eval**) per comprendere le astrazioni funzionali?
- Assumiamo **scoping statico** (vedremo poi quello **dinamico**)

```
type eval = | Int of int    | Bool of bool    | Unbound  
          | Funval of efun  
and efun = exp * eval env
```

- La definizione di **efun** mostra che una astrazione funzionale è una **chiusura**, comprendente:
  - Codice della funzione dichiarata
  - Ambiente al momento della dichiarazioneI riferimenti non locali della astrazione verranno risolti nell'ambiente di dichiarazione

12

## Semantica operativa di astrazione e applicazione di funzione: **scoping statico**


$$env \triangleright Fun(x,e) \Rightarrow Funval(Fun(x,e), env)$$
$$env \triangleright e1 \Rightarrow v1 \quad v1 = Funval(Fun(x,e), env1)$$
$$\frac{env \triangleright e2 \Rightarrow v2 \quad env1[v2 / x] \triangleright e \Rightarrow v}{env \triangleright Apply(e1,e2) \Rightarrow v}$$

13

## Semantica eseguibile: **scoping statico**



```
let rec sem ((e:exp), (r:eval env)) =  
  match e with  
  | ....  
  | Fun("x", a) -> Funval(e,r)  
  | Apply(e1, e2) -> match sem(e1, r) with  
    | Funval(Fun("x", a), r1) ->  
      sem(a, bind(r1, x, sem(e2, r)))  
    | _ -> failwith("no funct in apply")
```

Il corpo della funzione viene valutato nell'ambiente ottenuto legando i parametri formali ai valori dei parametri attuali nell'ambiente **r1** che è quello in cui era stata valutata l'astrazione

14

## Semantica operativa vs. eseguibile



$$\frac{\begin{array}{l} env \triangleright e1 \Rightarrow v1 \quad v1 = \text{Funval}(\text{Fun}(x,e), env1) \\ env \triangleright e2 \Rightarrow v2 \quad env1[v2/x] \triangleright e \Rightarrow v \end{array}}{env \triangleright \text{Apply}(e1,e2) \Rightarrow v}$$

```
let rec sem ((e:exp), (r:eval env)) =
  match e with
  | ...
  | Fun("x", a) -> Funval(e,r)
  | Apply(e1, e2) -> match sem(e1, r) with
    | Funval(Fun("x", a), r1) ->
      sem(a, bind(r1, x, sem(e2, r)))
    | _ -> failwith("no funct in apply")
```

15

## Giochiamo con la semantica (2)



- ☛ Vediamo ora lo **scoping dinamico**. Dobbiamo modificare **eval**:

```
type eval = | Int of int | Bool of bool | Unbound
            | Funval of efun
and efun = expr
```

- ☛ La definizione di **efun** mostra che l'astrazione funzionale contiene solo il codice della funzione dichiarata
- ☛ Il corpo della funzione verrà valutato nell'ambiente ottenuto
  - ✓ legando i parametri formali ai valori dei parametri attuali
  - ✓ nell'ambiente in cui avviene la applicazione

16

## Semantica operativa di astrazione e applicazione di funzione: **scoping dinamico**



$$env \triangleright Fun(x,e) \Rightarrow Funval(Fun(x,e))$$

$$env \triangleright e1 \Rightarrow v1 \quad v1 = Funval(Fun(x,e))$$

$$\frac{env \triangleright e2 \Rightarrow v2 \quad env[v2/x] \triangleright e \Rightarrow v}{env \triangleright Apply(e1,e2) \Rightarrow v}$$

17

## Semantica eseguibile: scoping dinamico



```
let rec sem (e:exp) (r:eval env) =
  match e with
  | ...
  | Fun("x", a) -> Funval(e)
  | Apply(e1, e2) -> match sem(e1, r) with
    | Funval(Fun("x", a)) ->
      sem(a, bind(r, x, sem(e2, r)))
    | _ -> failwith("no funct in apply")
```

Il corpo della funzione viene valutato nell'ambiente ottenuto legando i parametri formali ai valori dei parametri attuali nell'ambiente *r* che è quello i cui viene effettuata la chiamata

18

## Ricapitolando: regole di scoping



```
type efun = expr * eval env
| Apply(e1, e2) -> match sem(e1, r) with
  | Funval(Fun("x", a), r1) ->
    sem(a, bind(r1, x, sem(e2, r)))
```

- ✦ **Scoping statico (lessicale):** l'ambiente non locale della funzione è quello esistente al momento in cui viene valutata l'astrazione

```
type efun = expr
| Apply(e1, e2) -> match sem(e1, r) with
  | Funval(Fun("x", a)) ->
    sem(a, bind(r, x, sem1(e2, r)))
```

- ✦ **Scoping dinamico:** l'ambiente non locale della funzione è quello esistente al momento in cui avviene l'applicazione
- ✦ Nel **linguaggio didattico**, adottiamo lo **scoping statico**
  - discuteremo lo scoping dinamico successivamente

19



## INTERPRETE

(SCOPING STATICO, FUNZIONI CON PIÙ ARGOMENTI)

20



```

type efun = exp * eval env

and makefun ((a:exp),(x:eval env)) =
  (match a with
   | Fun(ii,aa) ->
       Funval(a,x)
   | _ -> failwith ("Non-functional object"))
and applyfun((ev1:eval),(ev2:eval list)) =
  ( match ev1 with
   | Funval(Fun(ii,aa),r) ->
       sem(aa, bindlist( r, ii, ev2))
   | _ -> failwith ("attempt to apply
                    a non-functional object"))

```

21



```

let rec sem ((e:exp), (r:eval env)) =
  match e with
  | Eint(n) -> Int(n)
  | Ebool(b) -> Bool(b)
  | Den(i) -> applyenv(r,i)
  | Iszero(a) -> iszero(sem(a, r))
  | Eq(a,b) -> equ(sem(a, r),sem(b, r))
  | :
  | Ifthenelse(a,b,c) -> let g = sem(a, r) in
      if typecheck("bool",g) then
        (if g = Bool(true) then sem(b, r) else sem(c, r))
      else failwith ("nonboolean guard")
  | Let(i,e1,e2) -> sem(e2, bind( r ,i, sem(e1, r)))
  | Fun(i,a) -> makefun(Fun(i,a), r)
  | Appl(a,b) -> applyfun(sem(a, r), semlist(b, r))
  | Rec(i,e) -> makefunrec(i, e, r)
and semlist (el, r) = match el with
  | [] -> []
  | e::el1 -> sem(e, r):: semlist(el1, r)
val sem : exp * eval env -> eval = <fun>
val semlist: exp list * eval env -> eval list

```

22



## DEFINIZIONE RICORSIVE

23

## Funzioni ricorsive



☛ Come è fatta una definizione di funzione ricorsiva?

- Espressione `Let (f, e1, e2)` in cui
  - ✓ `f` è il nome della funzione (ricorsiva)
  - ✓ `e1` è una astrazione `Fun(ii, aa)` nel cui corpo `aa` c'è una applicazione di `Den f`

**Esempio:**

```
Let("fact", Fun(["x"],
  Ifthenelse(Eq(Den "x", Eint 0), Eint 1,
  Prod(Den "x",
  Appl (Den "fact", [Diff(Den "x", Eint 1)]))),
  Appl(Den "fact", [Eint 4]))
```

**In OCaml:**

```
let fact x = if (x == 0) then 1 else (x *
  fact(x-1)) in fact(4) ... non funziona!!!
```

24

Guardiamo la semantica

```
let rec sem ((e:exp), (r:eval env)) =
  match e with
  | Let(i,e1,e2) ->
      sem (e2, bind (r ,i, sem(e1, r)))
  | Fun(ii, aa) -> Funval(Fun(ii,aa), r)
  | Appl(a,b) ->
      match sem(a, r) with
      | Funval(Fun(ii,aa), r1) ->
          sem(aa,bindlist(r1, ii, semlist(b r)))
```

Il corpo **aa** (che include **Den "fact"**) è valutato in un ambiente che è quello (**r1**) in cui si valutano sia l'espressione **Let** che l'espressione **Fun**, esteso con una associazione per i parametri formali **ii**. Tale ambiente non contiene il nome **"fact"** pertanto **Den "fact"** restituisce **Unbound!!!**

25



### **MORALE:**

Per permettere la ricorsione bisogna che il corpo della funzione venga valutato in un ambiente in cui è già stato inserita l'associazione tra il nome e la funzione.

Abbiamo bisogno di

- un diverso costrutto per "dichiarare" funzioni ricorsive (come il **let rec** di ML)
- oppure un diverso costrutto di astrazione per le funzioni ricorsive (come facciamo noi)

26





## Il costrutto **Rec**

```
Let("fact",  
  Rec("fact",  
    Fun(["x"], Ifthenelse(Eq(Den "x", Eint 0), Eint 1,  
      Prod(Den "x", Appl (Den "fact", [Diff(Den "x", Eint 1)]))))),  
  Appl(Den "fact",[Eint 4]))
```

Tipico uso di **Rec()**:

```
Let( "f", Rec("f", Fun([args], body), exp))
```

**Letrec(i, e1, e2)** può essere visto come una notazione per **Let(i, Rec(i,e1), e2)**

27



## Tipi esprimibili

Con l'introduzione delle funzioni ricorsive si deve modificare anche il dominio **eval** includendo le funzioni ricorsive.

28

## makefunrec

```
type eval = | Int of int | Bool of bool
            | Unbound | Funval of efun
and efun = expr * eval env
and makefunrec (f, e1, (r:eval env)) =
  let functional (rr: eval env) =
    bind(r, f, makefun(e1,rr)) in
  let rec rfix =
    function x -> functional rfix x
    in makefun(e1, rfix)
```



L'ambiente calcolato da **functional** contiene l'associazione tra il nome della funzione e la chiusura con l'ambiente della definizione

29

## makefunrec, più esplicitamente

```
type eval = | Int of int | Bool of bool
            | Unbound | Funval of efun
and efun = expr * eval env
and makefunrec (f, Fun(args,body), (r:eval env)) =
  let functional (rr: eval env) =
    bind(r, f, Funval(Fun(args,body),rr)) in
  let rec (rfix: string -> eval) =
    function x -> (functional rfix) x
    in Funval(Fun(args,body), rfix)
```



L'ambiente calcolato da **functional** contiene l'associazione tra il nome della funzione e la chiusura con l'ambiente soluzione della definizione

30

## Altri casi di ricorsione già visti: Semantica del while



```
let rec semc(While(e, cl), (r:dval env), (s: mval store)) =
  let g = sem(e, r, s) in
    if typecheck("bool",g) then
      (if g = Bool(true)
        then semcl((cl @ [While(e, cl)]), r, s)
        else s)
      else failwith ("nonboolean guard")
```

Definizione che esprime il comportamento del while in termini di se stesso

→ Equazione ricorsiva

31

## Definizioni ricorsive e punti fissi



- ✎ Consideriamo la funzione  $f: \mathbb{N} \rightarrow \mathbb{N}$  definita nel modo seguente
- ✎  $f(0) = 0$
- ✎  $f(n+1) = f(n) + 2n + 1$
- ✎ Due equazioni, la seconda ricorsiva, che definiscono le proprietà che la funzione deve soddisfare

32

## Soluzione di definizione ricorsiva



✎  $f(0) = 0$

✎  $f(n+1) = f(n) + 2n + 1$

✎ L'unica soluzione di queste equazioni è la funzione  $g(x) = x * x$

33

## Soluzione di equazioni di punto fisso



✎ La soluzione dell'equazione si può ottenere mediante **approssimazioni successive**

✎ Ogni approssimazione si avvicina alla soluzione dell'equazione

✎ Per trovare la soluzione dell'equazione partiamo dalla funzione **f0** non definita (in termini di insiemi di coppie, **f0** è la funzione che non contiene coppie)

34



- $f_0(n) = \text{indefinito}$ 
  - $f_0 = \emptyset$
- $f_1(0) = 0, \quad f_1(n+1) = f_0(n) + 2n + 1$ 
  - $f_1 = \{(0,0)\}$
- $f_2(0) = 0, \quad f_2(n+1) = f_1(n) + 2n + 1$ 
  - $f_2 = \{(0,0), (1,1)\}$
- $f_3(0) = 0, \quad f_3(n+1) = f_2(n) + 2n + 1$ 
  - $f_3 = \{(0,0), (1,1), (2,4)\}$
- $\vdots$
- Limite di questa sequenza è la funzione  
 $g(x) = x^2$
- che soddisfa le equazioni iniziali

35

## Cosa abbiamo fatto?



- Il processo di approssimazione utilizza strumenti funzionali di ordine superiore, ovvero abbiamo usato un funzionale  $F$  ( $F: (N \rightarrow N) \rightarrow (N \rightarrow N)$ ) che prende l'approssimazione  $f_i$  e restituisce l'approssimazione  $f_{i+1}$
- La soluzione è pertanto una funzione  $f$  tale che  $F(f) = f$
- Infatti:  $F(g)(0) = 0 = 0^2 = g(0)$   
 $F(g)(n+1) = F(g)(n) + 2n + 1 =$   
 $n^2 + 2n + 1 = (n+1)^2 = g(n+1)$

36



## Fixpoint iteration

Procedimento per ottenere un  
“minimo” punto fisso di un  
operatore

$$\begin{aligned} f &= \text{fix}(F) \\ &= f_0, f_1, f_2, f_3 \dots\dots \\ &= -, F(f_0), F(f_1), F(f_2), \dots\dots \\ &= \bigcup_i F^i(-) \end{aligned}$$

37



## Fondamenti

- ✎ Diverse costruzioni definiscono le condizioni per l'esistenza dei punti fissi.
- ✎ **Teorema di Tarski:** Una funzione monotona crescente su un reticolo completo ha un reticolo completo di punti fissi.
- ✎ **Teorema di Banach** stabilisce le condizioni per l'esistenza di punti fissi su spazi metrici
- ✎ **Teorema di Kleene** esistenza del minimo punto fisso di funzioni continue su ordine parziali completi
- ✎ .....

38

## ... a noi ci serve?



- ✎ Ci serve, eccome se ci serve!!
- ✎ *The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions.* Wirth, Niklaus (1976). Algorithms + Data Structures = Programs. Prentice-Hall.
- ✎ Una utile lettura:  
[http://en.wikipedia.org/wiki/Recursion\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Recursion_(computer_science))
- ✎ Metodo per costruire la soluzione di una equazione di punto fisso nella definizione della semantica dei linguaggi di programmazione

39

## Ambiente locale dinamico



- ✎ Per ogni attivazione entrata in un blocco o applicazione di funzione abbiamo attualmente nel record di attivazione l'intero ambiente
  - implementato come una funzione
- ✎ Le regole della semantica dell'ambiente locale dinamico non lo richiedono. In realtà possiamo inserire nel record di attivazione
  - una tabella che implementa il solo ambiente locale
  - e tutto quello che ci serve per reperire l'ambiente non locale in accordo con la regola di scoping
- ✎ quando l'attivazione termina
  - uscita dal blocco o ritorno della applicazione di funzione possiamo eliminare l'ambiente locale (e cose eventualmente associate) insieme a tutte le altre informazioni contenute nel record di attivazione

40

## L'ambiente locale



- ✎ nel caso del blocco (Let) contiene una sola associazione (la dichiarazione del let)
- ✎ nel caso della applicazione (Apply) contiene tante associazioni quanti sono i parametri
- ✎ rappresentiamo l'ambiente locale con una coppia di array "corrispondenti"
  - **l'array dei nomi**
  - **l'array dei valori denotati**la cui dimensione è determinata **staticamente** dalla sintassi del costrutto

41

## E l'ambiente non locale?



**namestack, pila di array di identificatori**

**evalstack, pila di array di valori denotati**

- ✎ in ogni istante le pile rappresentano la sequenza di ambienti locali corrispondenti alla catena di attivazioni (catena dinamica)
- ✎ **Scoping dinamico**, non serve altro
  - se un identificatore non esiste nell'ambiente locale, lo cerco negli ambienti locali che lo precedono nella pila
  - il primo che (eventualmente) trovo identifica l'associazione corretta perché è l'ultima creata nel tempo
- ✎ **Scoping statico**, la cosa non è così semplice
  - nel caso delle funzioni, l'ambiente non locale giusto non è quello che precede sulla pila quello locale, ma è quello che esisteva al momento dell'astrazione
  - tale ambiente ci è noto a tempo di esecuzione perché è contenuto nella chiusura (che è la semantica della funzione che applichiamo)

42

## Scoping statico



namestack, pila di array di identificatori

evalstack, pila di array di valori denotati

- in ogni istante le pile rappresentano la sequenza di ambienti locali corrispondenti alla catena di attivazioni (catena dinamica)
- nel caso di una applicazione di funzione, l'ambiente non locale giusto è quello che esisteva al momento dell'astrazione
  - contenuto nella chiusura (che è la semantica della funzione che applichiamo)
- Una possibile implementazione

```
let applyfun ((ev1:eval),(ev2:eval list)) =  
  ( match ev1 with  
  | Funval(Fun(ii,aa),r) -> newframes(aa,bindlist(r, ii, ev2))  
  | _ -> failwith ("attempt to apply a non-functional object"))
```

43

## L'ambiente della chiusura



- Domanda: *l'ambiente contenuto nella chiusura esiste sempre sulla pila nel momento in cui applichiamo la funzione?*
- Si: se ci limitiamo ad applicare funzioni reperite attraverso il loro nome (funzioni denotate!)
  - per la semantica del Let, siamo sicuri che l'applicazione di un Den "ide" può essere eseguita solo se è visibile
    - ✓ Si dimostra per induzione, contenuto nella pila
  - quello contenuto nella chiusura è
    - ✓ r stesso, se la funzione è ricorsiva
    - ✓ quello che precede r nella pila (cioè quello in cui è valutato il Let), se la funzione non è ricorsiva
- l'implementazione dovrà garantire che l'ambiente della chiusura sia comunque presente nella pila

44

## Pila degli ambienti locali e scoping statico



- in ogni istante le pile rappresentano la sequenza di ambienti locali corrispondenti alla catena di attivazioni
- gli ambienti locali visibili come non locali secondo la regola di scoping statico formano (quasi sempre!) una sottosequenza di quella contenuta nella pila
  - tale sottosequenza riproduce a run time la struttura statica di annidamento di blocchi e funzioni

45

## Ambienti locali e scoping statico



- se l'attivazione corrente è un blocco, l'ambiente non locale giusto è quello precedentemente sulla testa della pila
- se l'attivazione corrente è relativa alla applicazione di una funzione di nome "f", deve esserci sulla pila una attivazione del blocco o applicazione in cui "f" è stata dichiarata
- se la applicazione era relativa ad un nome locale, tale attivazione precede quella di "f" sulla pila
- se la applicazione era relativa ad un nome non locale, ci possono essere in mezzo altre attivazioni che non ci interessano
- se l'attivazione corrente è relativa alla applicazione di una funzione senza nome, ottenuta dalla valutazione di una espressione (le funzioni sono esprimibili!), dobbiamo preoccuparci di fare in modo che sulla pila ci sia anche l'ambiente della chiusura

46

## Catena dinamica e catena statica



namestack, pila di array di identificatori

evalstack, pila di array di valori denotati

- in ogni istante le pile rappresentano la sequenza di ambienti locali corrispondenti alla catena di attivazioni (catena dinamica)
- gli ambienti locali visibili come non locali secondo la regola di scoping statico formano (quasi!) una sottosequenza di quella contenuta nella pila
- per ricostruire la struttura statica e determinare correttamente l'ambiente non locale, associamo ad ogni record di attivazione un puntatore (catena statica) al giusto ambiente locale sulla pila
  - quello precedentemente sulla testa della pila, se entro in un blocco
  - quello contenuto nella chiusura, se applico una funzione
- il tipo ambiente ha ora come implementazione un semplice intero
  - indice negli array che realizzano le pile anche all'interno delle chiusure
- una nuova pila nell'implementazione dell'ambiente

**slinkstack**, pila di (puntatori ad) ambienti

47

## Funzioni esprimibili e *retention*



- in un linguaggio funzionale di ordine superiore (come il linguaggio didattico), è possibile che la valutazione di una applicazione di funzione  $\text{Appl}(e_1, e_2)$  ritorni una funzione, cioè un valore (chiusura) del tipo  $\text{Funval}(e, \rho)$  in cui
  - $e$  è un'espressione di tipo Fun
  - $\rho$  è l'ambiente in cui la funzione è stata costruita, cioè l'ambiente contenuto nel frame della applicazione  $\text{Appl}(e_1, e_2)$ , che serve per risolvere eventuali riferimenti non locali di  $e$
- quando la valutazione dell'applicazione  $\text{Appl}(e_1, e_2)$  ritorna, normalmente si dovrebbero eliminare tutte le pile:

48

# retention



- ✎ non è possibile eseguire il pop sulle pile relative all'ambiente, perché l'ambiente  $\rho$  attualmente testa della pila è utilizzato dentro la chiusura che si trova sulla pila dei valori temporanei
- ✎ in tale situazione è necessario ricorrere alla *retention*, in cui l'ambiente locale viene conservato
  - le teste delle pile che realizzano l'ambiente vengono conservate (ed etichettate come Retained, in una ulteriore pila parallela tagstack)
  - la computazione continua in uno stato in cui l'ambiente corrente non è necessariamente quello in testa alle pile
  - gli ambienti conservati devono prima o dopo essere eliminati, riportando le pile di ambienti nello stato normale