

Implementazione dell'ambiente (linguaggio funzionale)

Contenuti

- ambiente locale dinamico, scoping statico
 - cosa serve in ogni attivazione (catena statica)
 - un problema con le funzioni esprimibili: la retention
 - implementazione: strutture dati e operazioni
 - cosa cambia nell'interprete iterativo
 - eliminazione del calcolo di punto fisso per la ricorsione
 - un esempio senza retention
- ottimizzazioni eseguibili durante la compilazione
 - traduzione dei riferimenti ed eliminazione dei nomi
- ambiente locale dinamico, scoping dinamico (digressione)
 - l'implementazione standard (deep binding)
 - cenni ad altre implementazioni (shallow binding)
 - sono possibili ottimizzazioni?

Ambiente locale dinamico

- per ogni attivazione
 - entrata in un blocco o applicazione di funzioneabbiamo attualmente nel record di attivazione l'intero ambiente
 - implementato come una funzione
- in accordo con la semantica dell'ambiente locale dinamico 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 funzionepossiamo eliminare l'ambiente locale (e cose eventualmente associate) insieme a tutte le altre informazioni contenute nel record di attivazione
 - ma attenzione a quanto vedremo dopo!

L'ambiente locale

- nel caso del blocco (`Let`)
 - contiene una sola associazione
- 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 denotatila cui dimensione è determinata dalla sintassi del costrutto
- dato che la pila dei record di attivazione è realizzata con varie pile parallele, la pila di ambienti `envstack` è rimpiazzata (per ora) da due pile di ambienti locali
 - `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)

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)
- se lo scoping è dinamico, non mi 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
- studieremo più avanti (digressione) questa implementazione
- se lo scoping è 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)

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)

- ▶ questo fatto risulta evidente guardando la `applyfun` dell'interprete iterativo

```
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"))
```

- ▶ l'ambiente contenuto nella chiusura esiste sempre sulla pila nel momento in cui applichiamo la funzione?

L'ambiente della chiusura

- l'ambiente contenuto nella chiusura esiste sempre sulla pila nel momento in cui applichiamo la funzione?
- la risposta è “non necessariamente”
- è facile convincersi che la risposta è “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
 - per induzione, contenuto nella pila
 - l'ambiente ρ
 - prodotto dalla esecuzione del **Let**
 - che contiene “**ide**”
 - quello contenuto nella chiusura è
 - ρ stesso, se la funzione è ricorsiva
 - quello che precede ρ 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
 - vedi discorso successivo sulla retention

Pila degli ambienti locali e 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
- 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
 - 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

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, associa 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 pileanche all'interno delle chiusure
- ▶ una nuova pila nell'implementazione dell'ambiente

`slinkstack`, pila di (puntatori ad) ambienti

Funzioni esprimibili e *retention*

- in un linguaggio funzionale di ordine superiore (come il nostro), è possibile che la valutazione di una applicazione di funzione

$\text{App1}(e1, e2)$

ritorni una funzione, cioè un valore (chiusura) del tipo $\text{Funval}(e, \rho)$ in cui

- e è un'espressione di tipo Fun
- ρ è l'ambiente in cui la funzione è stata costruita, cioè l'ambiente contenuto nel frame della applicazione $\text{App1}(e1, e2)$, che serve per risolvere eventuali riferimenti non locali di e
- quando la valutazione dell'applicazione $\text{App1}(e1, e2)$ ritorna, normalmente si dovrebbero poppare tutte le pile
 - oltre a cstack e tempvalstack , anche le pile che realizzano l'ambiente
- non è possibile eseguire il pop sulle pile relative all'ambiente, perché l'ambiente ρ 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

L'ambiente e le pile che lo realizzano

```
type 't env = int
let (currentenv: eval env ref) = ref(0)
let namestack = emptystack(stacksize, [| "dummy" |])
let evalstack = emptystack(stacksize, [| Unbound |])
let slinkstack = emptystack(stacksize, !currentenv)
type tag = Retained| Standard
let tagstack = emptystack(stacksize, Standard)
```

► invece di `envstack`, quattro pile parallele

`namestack`, pila di array di identificatori

`evalstack`, pila di array di valori denotati

`slinkstack`, pila di ambienti

`tagstack`, pila di etichette per la retention

```
let svuotaenv() = svuota(namestack); svuota(tagstack); svuota(evalstack); svuota(slinkstack)
let topenv() = !currentenv
let popenv () = pop(namestack); pop(evalstack); pop(slinkstack); pop(tagstack); currentenv := !currentenv - 1
let pushenv(r) = if r = !currentenv then
  (push([| |],namestack); push([| |],evalstack); push(Standard,tagstack); push(r,slinkstack);
  currentenv := lungh(namestack) ) else currentenv := r
```

Le operazioni sull'ambiente 1

```
let applyenv ((x: eval env), (y: ide)) =
  let n = ref(x) in
  let den = ref(Unbound) in
  while !n > -1 do
    let lenv = access(namestack,!n) in
    let nl = Array.length lenv in
    let index = ref(0) in
    while !index < nl do
      if Array.get lenv !index = y then
        (den := Array.get (access(evalstack,!n)) !index;
         index := nl)
      else index := !index + 1
    done;
    if not(!den = Unbound) then n := -1 else n := access(slinkstack,!n)
  done;
  !den
```

- ritrovo l'associazione non locale seguendo all'indietro i puntatori di catena statica

```
let bind ((r:eval env),i,d) =
  push(Array.create 1 i,namestack);
  push(Array.create 1 d,evalstack);
  push(r,slinkstack);
  push(Standard,tagstack);
  (lungh(namestack): eval env)
```

Le operazioni sull'ambiente 2

```
let bindlist(r, il, el) =
  let n = List.length il in
  let ii = Array.create n "dummy" in
  let dd = Array.create n Unbound in
  let ri = ref(il) in
  let rd = ref(el) in
  let index = ref(0) in
  while !index < n do
    let i = List.hd !ri in
    let d = List.hd !rd in
    ( ri := List.tl !ri;
      rd := List.tl !rd;
      Array.set ii !index i;
      Array.set dd !index d;
      index := !index + 1)
  done;
  push(ii, namestack);
  push(dd, evalstack);
  push(r, slinkstack);
  push(Standard, tagstack);
  (lungh(namestack): eval env)

let emptyenv(x) = currentenv := -1; svuota(namestack);
  svuota(evalstack); svuota(slinkstack); svuota(tagstack); !currentenv
```

Gestione (parziale) della retention

```
let retained (n:eval env) = access(tagstack,n) = Retained
```

```
let retain () = setta(tagstack, !currentenv, Retained);  
  let cont = ref(lungh(namestack)) in  
  while !cont > -1 & retained(!cont) do cont := !cont - 1 done; currentenv := !cont
```

```
let to_be_retained (v:eval) = match v with  
  | Funval (e, r1) -> not(r1 < !currentenv)  
  | _ -> false
```

```
let retainorpopenv (r, valore) =  
  if (topenv() = r) then () else  
  if !currentenv < lungh(namestack) then retain() else  
  if to_be_retained(valore) then retain() else  
  (popenv(); let index = ref(!currentenv) in  
    while !index > r do  
      if retained(!index) then (currentenv := !currentenv - 1;  
        index := !index -1) else (index := r)  
    done )
```

```
let collectretained (r) = let index = ref(lungh(namestack)) in  
  while !index > r do  
    pop(namestack); pop(evalstack);  
    pop(slinkstack); pop(tagstack);  
    index := !index -1  
  done;
```

```
currentenv := lungh(namestack)
```

Le novità nell'interprete iterativo: ricorsione

- ▶ il fatto che l'ambiente sia un intero (puntatore nelle pile degli ambienti) ci obbliga a modificare l'implementazione delle funzioni ricorsive
 - ▶ dobbiamo eliminare il calcolo di punto fisso che era necessario per determinare l'ambiente da inserire nella chiusura
 - ▶ possiamo direttamente inserire nella chiusura la prossima posizione nelle pile degli ambienti
 - dove verrà inserita esattamente l'associazione per il nome della funzione ricorsiva
 - l'ambiente corrispondente non esiste ancora, ma ci sarà quando la chiusura verrà utilizzata

```
let makefunrec (f, (a:exp), (x:eval env)) =  
  makefun(a, ((lungh(namestack) + 1): eval env))
```

Le novità nell'interprete iterativo: retention

```
let sem ((e:exp), (r:eval env)) =
  push(emptystack(1,Unbound),tempvalstack);
  newframes(e,r);
  while not(empty(cstack)) do
    while not(empty(top(cstack))) do
      ...
    done;
    let valore= top(top(tempvalstack)) in
    pop(top(tempvalstack));
    pop(cstack);
    pop(tempvalstack);
    push(valore,top(tempvalstack));
    retainorpopenv (r, valore)
  done;
  collectretained(r);
  let valore = top(top(tempvalstack)) in
  pop(tempvalstack); valore
```


Un esempio che non funziona senza retention

```
# sem(  
  Appl(  
    Appl(  
      Fun(["x"],  
        Fun(["y"], Sum(Den("x"), Den("y")))),  
      [Eint 3]),  
    [Eint 5]),  
  emptyenv Unbound);;
```

- l'applicazione rossa pusha sulla pila dei temporanei il valore `Funval(Fun(["y"], Sum(Den("x"), Den("y"))), r)` ed `r` punta ad un ambiente che contiene l'associazione fra `"x"` e `Dint 3`
- senza retention, l'ambiente dell'applicazione rossa viene poppato e rimpiazzato da quello della applicazione nera
 - il cui link statico punta (per errore) a lui stesso
- l'applicazione di tale ambiente ad `"x"` porta in un ciclo infinito
 - `applyenv` quando non trova il nome cerca nell'ambiente puntato dal link statico

Analisi statiche e ottimizzazioni

- se lo scoping è statico, chi legge il programma ed il compilatore possono
 - controllare che ogni riferimento ad un nome abbia effettivamente una associazione
 - oppure segnalare staticamente l'errore
 - inferire-controllare il tipo per ogni espressione e
 - segnalare gli eventuali errori di tipo
- sono possibili anche delle ottimizzazioni legate alle seguente proprietà
- dato che ogni attivazione di funzione avrà come puntatore di catena statica il puntatore all'ambiente locale in cui la funzione è stata definita
 - sempre lo stesso per tutte le attivazioni (anche ricorsive) relative alla stessa definizione
- il numero di passi che a tempo di esecuzione dovrà fare lungo la catena statica per trovare l'associazione (non locale) per l'identificatore "x" è costante
 - non dipende dalla catena delle attivazioni a tempo di esecuzione
 - è esattamente la differenza fra le profondità di annidamento del blocco in cui "x" è dichiarato e quello in cui è usato

Traduzione ed eliminazione dei nomi

- il numero di passi che a tempo di esecuzione dovrò fare lungo la catena statica per trovare l'associazione (non locale) per l'identificatore "x" è esattamente la differenza fra le profondità di annidamento del blocco in cui "x" è dichiarato e quello in cui è usato
- ogni riferimento `Den ide` nel codice può essere staticamente tradotto in una coppia (m, n) di numeri interi
 - m è la differenza fra le profondità di nesting dei blocchi (0 se `ide` si trova nell'ambiente locale)
 - n è la posizione relativa (partendo da 0) della dichiarazione di `ide` fra quelle contenute nel blocco
- l'interprete o il supporto a tempo di esecuzione (la nuova `applyenv`) interpreteranno la coppia (m, n) come segue
 - effettua m passi lungo la catena statica partendo dall'ambiente locale attualmente sulla testa della pila
 - restituisci il contenuto dell'elemento in posizione n nell'array di valori denotati così ottenuto
- l'accesso diventa efficiente (non c'è più ricerca per nome)
- si può economizzare nella rappresentazione degli ambienti locali che non necessitano più di memorizzare i nomi
 - si può eliminare la pila `namestack`

L'utile esercizio di traduzione dei nomi

- dato il programma, per tradurre una specifica occorrenza di `Den ide` bisogna
 - identificare con precisione la struttura di annidamento
 - identificare il blocco o funzione dove occorre l'associazione per `ide` (o scoprire subito che non c'è) e vedere in che posizione è `ide` in tale ambiente
 - contare la differenza delle profondità di nesting
- un modo conveniente per ottenere questo risultato è costruire una analisi statica
 - una esecuzione del programma con l'interprete appena definito
 - che esegue solo i costrutti che hanno a che fare con l'ambiente (ignorando tutti gli altri)
 - e dell'ambiente guarda solo nomi e link statici (`namestack` e `slinkstack`)
 - che costruisce un nuovo ambiente locale seguendo la struttura statica (`Let`, `Fun`, `Rec`) e non quella dinamica (`Let` e `Apply`)
 - facendo attenzione ad associare ad ogni espressione l'ambiente in cui deve essere valutata
- chiaramente diverso dalla costruzione dell'ambiente a tempo di esecuzione
 - basata sulle applicazioni e non sulle definizioni di funzione
- ma sappiamo che per la traduzione dei nomi è la struttura statica quella che conta

Un esempio di traduzione dei nomi

```
Let ("expo",
  Rec
    ("expo",
      Fun (["base";"esp"],
        Let("f",
          Fun
            (["x"],
              Ifthenelse(
                Iszero (Den "esp"),
                Appl(Den "f", [Eint 1]),
                Prod(Den "x",
                  Appl(Den "expo",
                    [Den "x";Diff(Den "esp", Eint 1)]))),),
                Appl (Den "f", [Den "base"]))),),
        Appl (Den "expo", [Den "x"; Eint 3]))
```

Sviluppo dell'esempio 1

0

expo	-1
------	----

```
Let("expo",
  Rec
    ("expo",
      Fun(["base";"esp"],
        Let("f",
          Fun
            (["x"],
              Ifthenelse(
                Iszero(Den "esp"),
                Appl(Den "f", [Eint 1]),
                Prod(Den "x",
                  Appl(Den "expo",
                    [Den "x";Diff(Den "esp", Eint 1)]))),
                Appl(Den "f", [Den "base"]))),
          Appl(Den "expo", [Den "x"; Eint 3]))
```

da valutare nell'ambiente 0

```
Appl(Den "expo", [Den "x"; Eint 3])
```

0,0

errore

Sviluppo dell'esempio 2

```
Let("expo",
  Rec
    ("expo",
      Fun (["base";"esp"],
        Let("f",
          Fun
            (["x"],
              Ifthenelse(
                Iszero (Den "esp"),
                Appl(Den "f", [Eint 1]),
                Prod(Den "x",
                  Appl(Den "expo",
                    [Den "x";Diff(Den "esp", Eint 1)]))),
                Appl (Den "f", [Den "base"]))),
          Appl (Den "expo", [Den "x"; Eint 3]))
```

da valutare nell'ambiente -1

```
Rec
  ("expo",
    Fun (["base";"esp"],
      Let("f",
        Fun
          (["x"],
            Ifthenelse(
              Iszero (Den "esp"),
              Appl(Den "f", [Eint 1]),
              Prod(Den "x",
                Appl(Den "expo",
                  [Den "x";Diff(Den "esp", Eint 1)]))),
              Appl (Den "f", [Den "base"])))
```

Sviluppo dell'esempio 3

0	expo	-1
1	base	0
	esp	

```
Let("expo",
  Rec
    ("expo",
      Fun (["base";"esp"],
        Let("f",
          Fun
            (["x"],
              Ifthenelse(
                Iszero (Den "esp"),
                Appl(Den "f", [Eint 1]),
                Prod(Den "x",
                  Appl(Den "expo",
                    [Den "x";Diff(Den "esp", Eint 1)]))),
                Appl (Den "f", [Den "base"]))),
          Appl (Den "expo", [Den "x"; Eint 3]))
```

da valutare nell'ambiente 0

```
Fun (["base";"esp"],
  Let("f",
    Fun
      (["x"],
        Ifthenelse(
          Iszero (Den "esp"),
          Appl(Den "f", [Eint 1]),
          Prod(Den "x",
            Appl(Den "expo",
              [Den "x";Diff(Den "esp", Eint 1)]))),
          Appl (Den "f", [Den "base"])))
```


Sviluppo dell'esempio 4

0	expo	-1
1	base	0
	esp	
2	f	1

```
Let("expo",
  Rec
    ("expo",
      Fun(["base";"esp"],
        Let("f",
          Fun
            (["x"],
              Ifthenelse(
                Iszero (Den "esp"),
                Appl(Den "f", [Eint 1]),
                Prod(Den "x",
                  Appl(Den "expo",
                    [Den "x";Diff(Den "esp", Eint 1)]))),
                Appl (Den "f", [Den "base"]))),
          Appl (Den "expo", [Den "x"; Eint 3]))
```

da valutare nell'ambiente 2

```
Appl (Den "f", [Den "base"])
0,0      1,0
```

Sviluppo dell'esempio 5

0	expo	-1
1	base	0
	esp	
2	f	1
3	x	1

```
Let("expo",
  Rec
    ("expo",
      Fun (["base";"esp"],
        Let("f",
          Fun
            (["x"],
              Ifthenelse(
                Iszero (Den "esp"),
                Appl(Den "f", [Eint 1]),
                Prod(Den "x",
                  Appl(Den "expo",
                    [Den "x";Diff(Den "esp", Eint 1)]))),
                Appl (Den "f", [Den "base"]))),
          Appl (Den "expo", [Den "x"; Eint 3]))
```

da valutare nell'ambiente 1

```
Fun (["x"], Ifthenelse(
  Iszero (Den "esp"),
  Appl(Den "f", [Eint 1]),
  Prod(Den "x",
    Appl(Den "expo",
      [Den "x";Diff(Den "esp", Eint 1)]))))
```

Sviluppo dell'esempio 6

0	expo	-1
1	base	0
	esp	
2	f	1
3	x	1

```

Let("expo",
  Rec
    ("expo",
      Fun (["base";"esp"],
        Let("f",
          Fun
            (["x"],
              Ifthenelse(
                Iszero (Den "esp"),
                Appl(Den "f", [Eint 1]),
                Prod(Den "x",
                  Appl(Den "expo",
                    [Den "x";Diff(Den "esp", Eint 1)]))),
                Appl (Den "f", [Den "base"]))),
            Appl (Den "expo", [Den "x"; Eint 3]))

```

da valutare nell'ambiente 3

```

Ifthenelse(Iszero (Den "esp"),
  1,1
  Appl(Den "f", [Eint 1]),
  errore
  Prod(Den "x", Appl(Den "expo",
    0,0          2,0
    [Den "x";Diff(Den "esp", Eint 1)]))
    0,0          1,1

```

Scoping dinamico

- con la implementazione vista
 - pila dei record di attivazione
 - che per l'ambiente è realizzata attraverso `namestack` ed `evalstack`
 - con record (ambienti locali) creati e distrutti all'ingresso e uscita da blocchi e applicazioni di funzione
 - l'associazione per il riferimento non locale ad "x" è la prima associazione per "x" che si trova scorrendo la pila all'indietro
 - è una ricerca fatta sul nome in `namestack`
 - i nomi devono essere mantenuti
 - è l'implementazione più comune in LISP (deep binding), dove tuttavia la pila di ambienti locali (`A-list`)
 - è tenuta separata dalla pila dei records di attivazione
 - è rappresentata con una S-espressione (lista di coppie)
 - memorizzata nella heap come tutte le S-espressioni
 - accessibile e manipolabile come S-espressione da parte dei programmi
 - l'implementazione semplice dell'ambiente
 - che è l'unico vantaggio dello scoping dinamico
- si complica se introduciamo le chiusure per trattare gli argomenti funzionali e, soprattutto, i ritorni funzionali à la LISP

Shallow Binding

- ▶ si può semplificare il costo di un riferimento non locale
 - ▶ accesso diretto senza ricercacomplicando la gestione di creazione e distruzione di attivazioni
- ▶ l'ambiente è realizzato con un'unica tabella centrale che contiene tutte le associazioni attive (locali e non locali)
 - ▶ ha una entry per ogni nome utilizzato nel programma
 - ▶ in corrispondenza del nome, oltre all'oggetto denotato, c'è un flag che indica se l'associazione è o non è attiva
- ▶ i riferimenti (locali e non locali) sono compilati in accessi diretti alla tabella a tempo di esecuzione
 - ▶ non c'è più ricerca, basta controllare il bit di attivazione
- ▶ i nomi possono sparire dalla tabella ed essere rimpiazzati dalla posizione nella tabella
- ▶ diventa molto più complessa la gestione di creazione e distruzione di associazioni
 - ▶ la creazione di una nuova associazione locale rende necessario salvare quella corrente (se attiva) in una pila (detta pila nascosta)
 - ▶ al ritorno bisogna ripristinare le associazioni dalla pila nascosta
- ▶ la convenienza rispetto al deep binding dipende dallo "stile di programmazione"
 - ▶ se il programma usa molti riferimenti non locali e pochi `Let` e `Apply`

Scoping dinamico e scoping statico

- con lo scoping dinamico
 - si vede anche dalle implementazioni dell'ambiente

non è possibile effettuare nessuna analisi e nessuna ottimizzazione a tempo di compilazione
- lo scoping statico porta a programmi più sicuri
 - rilevamento statico di errori di nome
 - quando si usa un identificatore si sa a chi ci si vuole riferire
 - verifica e inferenza dei tipi statici
- e più efficienti
 - si possono far sparire i nomi dalle tabelle che realizzano gli ambienti locali
 - i riferimenti possono essere trasformati in algoritmi di accesso che sono essenzialmente indirizzamenti indiretti e non richiedono ricerca
- il problema della non compilabilità separata (ALGOL, PASCAL) si risolve (C) combinando la struttura a blocchi con scoping statico con un meccanismo di moduli separati con regole di visibilità