



Implementazione dell'interprete di un nucleo di linguaggio funzionale



Linguaggio funzionale didattico

- Consideriamo un nucleo di un linguaggio funzionale
 - sottoinsieme 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



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 *)
```



Valori esprimibili e ambiente

- Valori esprimibili

```
type evT = Int of int
          | Bool of bool
          | Unbound
```
- Ambiente: `evT env`

Let(x, e1, e2)

- 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 e 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

Semanica operativa di Let

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

```

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

```

Analisi

```
let rec eval((e: exp), (r: evT env)) =  
  match e with  
  .....
```

| Let(i, e1, e2) ->
 eval(e2, bind (r , i, eval(e1, r)))

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

Esempio di valutazione

```
# eval(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));;
```

```
-: evT = 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
```



Funzioni

- Passiamo ora a esaminare gli ingredienti fondamentali della programmazione nel paradigma funzionale
 - astrazione funzionale
 - applicazione di funzione

Sintassi

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

Astrazione

Applicazione

Funzioni

- Identificatori (**parametri formali**) nel costrutto di **astrazione**
Fun of ide list * exp
- Espressioni (**parameri attuali**) nel costrutto di **applicazione**
Appl of exp * exp list
- Per ora non ci occupiamo del modo del passaggio parametri
 - le espressioni parametro attuale sono valutate (**eval** o **dval**) e i valori ottenuti legati nell'ambiente al corrispondente parametro formale
- Per ora ignoriamo le funzioni ricorsive
- Assumeremo solitamente di avere funzione unarie
- Introducendo le funzioni, il linguaggio funzionale è completo
 - un linguaggio funzionale reale (tipo ML) ha in più i tipi, il pattern matching e le eccezioni

Giochiamo con la semantica (1)

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

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

- La definizione di **efun** mostra che una astrazione funzionale è una **chiusura**, che comprende
 - nome del parametro formale
 - codice della funzione dichiarata
 - ambiente al momento della dichiarazioneI riferimenti non locali dell'astrazione verranno risolti nell'ambiente di dichiarazione



Astrazione e applicazione di funzione: scoping statico

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

$$env \triangleright e1 \Rightarrow v1 \quad v1 = Funval(x, e, env1)$$

$$env \triangleright e2 \Rightarrow v2 \quad env1[v2 / x] \triangleright e \Rightarrow v$$

$$env \triangleright Apply(e1, e2) \Rightarrow v$$

Semantica eseguibile: scoping statico

```
let rec eval((e: exp), (r: evT env)) =  
  match e with  
  | ...  
  | Fun(i, a) -> Funval(i, a, r)  
  | Apply(e1, e2) -> match eval(e1, r) with  
    | Funval(i, a, r1) ->  
      eval(a, bind(r1, i, eval(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**, nel quale era stata valutata l'astrazione

Semantica operativa vs. eseguibile

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

$$env \triangleright Apply(e1, e2) \Rightarrow v$$

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


Giochiamo con la semantica (2)

- Vediamo ora lo **scoping dinamico**. Dobbiamo modificare **evT**

```
type evT = | Int of int | Bool of bool | Unbound
           | Funval of efun
and efun = ide * exp
```

- 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 nel quale avviene la applicazione



Astrazione e applicazione di funzione: scoping dinamico

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

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

$$env \triangleright e2 \Rightarrow v2 \quad env[v2 / x] \triangleright e \Rightarrow v$$

$$env \triangleright Apply(e1, e2) \Rightarrow v$$

Semantica eseguibile: scoping dinamico

```
let rec eval((e: exp), (r: evT env)) =  
  match e with  
  | ...  
  | Fun(i, a) -> Funval(i, a)  
  | Apply(e1, e2) -> match eval(e1, r) with  
    | Funval(i, a) ->  
      eval(a, bind(r, i, eval(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 , quello nel quale viene effettuata la chiamata

Ricapitolando: regole di scoping

```
type efun = ide * exp * eval env
| Apply(e1, e2) -> match eval(e1, r) with
  | Funval(i, a, r1) ->
    eval(a, bind(r1, i, eval(e2, r)))
```

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

```
type efun = ide * exp
| Apply(e1, e2) -> match eval(e1, r) with
  | Funval(i, a) ->
    eval(a, bind(r, i, seml(e2, r)))
```

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



Definizioni ricorsive

Funzioni ricorsive

- Come è fatta una definizione di funzione ricorsiva?
 - è una espressione $\text{Let}(f, e1, e2)$ nella quale
 - ✓ f è il nome della funzione (ricorsiva)
 - ✓ $e1$ è un'astrazione $\text{Fun}(i, a)$ nel cui corpo occorre una applicazione di $\text{Den } f$

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!!!

Guardiamo la semantica

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

- Il corpo **a** (che include **Den "fact"**) è valutato in un ambiente che è quello (**r1**) nel quale si valutano sia l'espressione **Let** che l'espressione **Fun**, esteso con una associazione per il parametro formale **x**. Tale ambiente non contiene il nome **"fact"** pertanto **Den "fact"** restituisce **Unbound!!!**

Morale

- Per permettere la ricorsione bisogna che il corpo della funzione venga valutato in un ambiente nel quale è già stata 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



Problema generale

- Come costruiamo la chiusura per la gestione della ricorsione?
- Il punto importante è che l'ambiente della chiusura deve contenere un binding per la gestione della ricorsione



Modello operativo



Letrec

- Estendiamo la sintassi astratta del linguaggio didattico con un opportuno costruttore

type exp =

:

| **Letrec of ide * ide * exp * exp**



Letrec

- Letrec("f", "x", fbody, letbody)
 - "f" identifica il nome della funzione
 - "x" identifica il parametro formale
 - fbody è il corpo della funzione
 - letbody è il corpo del let

Esempio

```
Letrec("fact",
      "n",
      Ifthenelse(Eq(Den("n"), EInt(0)),
                  EInt(1),
                  Prod(Den("n"),
                      Apply(Den("fact"),
                            Sub(Den("n"), EInt(1))))))
      Apply(Den("fact"), EInt(3)))
```

```
letrec fact n =
  if n = 0 the 1 else n * fact (n-1) in
  fact(3)
```

Tipi esprimibili

- Estendere i tipi esprimibili (**evT**) per avere le astrazioni funzionali ricorsive (**RecFunVal**)

```
type evT = | Int of int | Bool of bool  
          | Unbound | Funval of ide * exp * eval env  
          | Recfunval of ide * ide * exp * evT env
```



RecFunVal

Recfunval of ide * ide * exp * evT env

**Recfunval (funName,
param,
funBody,
staticEnvironment)**

Il codice dell'interprete

```
  :  
| Letrec(f, i, fBody, letBody) ->  
  let benv =  
    bind(r, f, (Recfunval(f, i, fBody, r)))  
in eval(letBody, benv)  
  :
```

Viene associato al nome della funzione ricorsiva una chiusura ricorsiva che contiene il nome della funzione stessa

Il codice dell'interprete (2)

```
:
| Apply(Den f, eArg) ->
  let fclosure = eval(f, r) in
  match fclosure with
  | Funval(arg, fbody, fDecEnv) ->
    ::
  | RecFunVal(f, arg, fbody, fDecEnv) ->
    let aVal = eval(eArg, r) in
      let rEnv = bind(fDecEnv, f, fclosure) in
        let aEnv = bind(rEnv, arg, aVal) in
          eval(fbody, aEnv)
  | _ -> failwith("non functional value")
| Apply(_,_) -> failwith("not function")
```

Passi dell'interprete

- Il valore della chiusura ricorsiva **Recfunval** è recuperato dall'ambiente corrente
- Il parametro attuale è valutato nell'ambiente del chiamante ottenendo il valore **aVal**
- L'ambiente statico **fDecEnv**, memorizzato nella chiusura, è esteso con il legame tra il nome della funzione e la sua chiusura ricorsiva, ottenendo l'ambiente **rEnv**
- L'ambiente effettivo di esecuzione **aEnv** si ottiene estendendo l'ambiente **rEnv** con il binding del passaggio del parametro



```
# let myRP =
  Letrec("fact", "n",
    Ifthenelse(Eq(Den("n"), EInt(0)),
      EInt(1),
      Prod(Den("n"),
        Apply(Den("fact"),
          Sub(Den("n"), CstInt(1))))),
    Apply(Den("fact"), EInt(3)));;

val myRP : exp = ...

# eval myRP emptyEnv;;
- : eval = Int 6
```



Altri casi di ricorsione già visti: while

```
let rec semc(While(e, cl), (r: dval env),
            (s: mval store)) =
  let g = eval(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



Modello denotazionale



Rec

- Estendiamo la sintassi astratta del linguaggio didattico con un opportuno costruttore

type exp =

:

| **Rec of ide * ide * exp**

makefunrec

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

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

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 3]))
```

- Tipico uso di **Rec**

Let("f", Rec("f", arg, body), exp)

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

Definizioni ricorsive

- Consideriamo la funzione $f: \mathbf{N} \rightarrow \mathbf{N}$ definita ricorsivamente nel modo seguente

$$f(n) = \text{if } (n = 0) \text{ then } 0 \text{ else } f(n-1) + 2n - 1$$

o equivalentemente con le seguenti equazioni

$$f(0) = 0$$

$$f(n+1) = f(n) + 2n + 1$$

- Si verifica facilmente che la funzione $g(n) = n * n$ è una soluzione. Ma...
 - come l'abbiamo trovata?
 - ce ne sono altre?

Soluzione del sistema

- Verifichiamo che $g(n) = n * n$ è una soluzione del sistema

$$f(0) = 0$$

$$f(n+1) = f(n) + 2n + 1$$

- Infatti

$$g(0) = 0*0 = 0$$

$$g(n+1) = (n+1)*(n+1) =$$

$$n*n + 2n + 1 = g(n) + 2n + 1$$

- Ma cosa significa precisamente la definizione ricorsiva?
 - come abbiamo trovato la soluzione g ?
 - ce ne sono altre?

Funzionali e punti fissi

- Leggiamo la definizione ricorsiva come la definizione di un **funzionale**, cioè una funzione (di ordine superiore) che trasforma funzioni in funzioni

$$\mathbf{F: (N \rightarrow N) \rightarrow (N \rightarrow N)}$$

- Riscriviamo le due equazioni

$$\mathbf{F(X)(0) = 0}$$

$$\mathbf{F(X)(n+1) = X(n) + 2n + 1}$$

dove \mathbf{X} deve essere di tipo $\mathbf{X: N \rightarrow N}$

- Ora, le **soluzioni** del sistema originale sono esattamente i **punti fissi** del funzionale \mathbf{F} , cioè le funzioni $\mathbf{k: N \rightarrow N}$ tali che $\mathbf{F(k) = k}$

Funzionali e punti fissi

- Riscriviamo le due equazioni

$$F(X)(0) = 0$$

$$F(X)(n+1) = X(n) + 2n + 1$$

- Vediamo infatti che $g(n) = n*n$ è un punto fisso di F , cioè $F(g) = g$

$$F(g)(0) = 0 = 0*0 = g(0)$$

$$F(g)(n+1) = F(g)(n) + 2n + 1 =$$

$$n*n + 2n + 1 = (n+1)*(n+1) = g(n+1)$$

Approssimazioni successive

- 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 \mathbf{f}_0 non definita (in termini di insiemi di coppie, \mathbf{f}_0 è la funzione che non contiene coppie)
- Ad ogni iterazione definiamo $\mathbf{f}_i = \mathbf{F}(\mathbf{f}_{i-1})$
- Il punto fisso sarà il “limite” di questo procedimento

- $f_0(n) = \text{indefinito}$
 - $f_0 = \emptyset$
- $f_1(0) = 0, \quad f_1(n+1) = f_0(n) + 2n + 1$
 - $f_1 = \{(0,0)\}$
- $f_2(0) = 0, \quad f_2(n+1) = f_1(n) + 2n + 1$
 - $f_2 = \{(0,0), (1,1)\}$
- $f_3(0) = 0, \quad f_3(n+1) = f_2(n) + 2n + 1$
 - $f_3 = \{(0,0), (1,1), (2,4)\}$
- \vdots
- Il limite di questa sequenza è la funzione

$$g(x) = x^*x$$
 che soddisfa le equazioni iniziali

Fixpoint iteration

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

$$f = \text{fix}(F)$$

$$= f_0, f_1, f_2, f_3, \dots \dots$$

$$= -, F(f_0), F(f_1), F(f_2), \dots \dots$$

$$= \bigcup_i F^i(-)$$



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
-

... 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.* Nicklaus Wirth (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



Scoping statico e dinamico: valutazione



Anzitutto...

- In presenza del solo costrutto di blocco, non c'è differenza fra le due regole di scoping...
- ...perché non c'è distinzione fra definizione e attivazione
 - un blocco viene “eseguito” immediatamente quando lo si incontra



Scoping statico e dinamico: verifiche

- Un riferimento non locale al nome x nel corpo di un blocco o di una funzione viene risolto
 - se lo scoping è statico, con la (eventuale) associazione per x creata nel blocco o astrazione più interni fra quelli che sintatticamente “contengono”
 - se lo scoping è dinamico, con la (eventuale) associazione per x creata per ultima nella sequenza di attivazioni (a tempo di esecuzione)

Scoping statico

- Guardando il programma (la sua struttura sintattica) siamo in grado di
 - verificare se l'associazione per x esiste
 - identificare la dichiarazione (o il parametro formale) rilevanti e conoscere quindi l'eventuale informazione sul tipo
- Il compilatore può “staticamente”
 - determinare gli errori di nome (identificatore non dichiarato, unbound)
 - fare il controllo di tipo e rilevare gli eventuali errori di tipo



Scoping dinamico

- L'esistenza di una associazione per x e il tipo di x dipendono dalla particolare sequenza di attivazioni
- Due diverse applicazioni della stessa funzione, che utilizza x come non locale, possono portare a risultati diversi
 - errori di nome si possono rilevare solo a tempo di esecuzione
 - non è possibile fare controllo dei tipi statico

Scoping statico: ottimizzazioni

- Un riferimento non locale al nome x nel corpo di un blocco o di una funzione e viene risolto
 - con la (eventuale) associazione per x creata nel blocco o astrazione più interni fra quelli che sintatticamente “contengono” e
- Guardando il programma (la sua struttura sintattica) siamo in grado di
 - verificare se l’associazione per x esiste
 - identificare la dichiarazione (o il parametro formale) rilevanti e conoscere quindi l’eventuale informazione sul tipo
- Il compilatore potrà ottimizzare l’implementazione al prim’ordine (mediante struttura dati) dell’ambiente sia la struttura dati che lo implementa
 - che l’algoritmo che permette di trovare l’entità denotata da un nome
- Tali ottimizzazioni sono impossibili con lo scoping dinamico

Regole di scoping e linguaggi

- Lo scoping statico è decisamente migliore
- L'unico linguaggio importante che ha una regola di scoping dinamico è LISP
- Alcuni linguaggi non hanno regole di scoping
 - l'ambiente è locale oppure globale
 - non ci sono associazioni ereditate da altri ambienti locali
 - PROLOG, JAVA
- Avere soltanto ambiente locale e ambiente non locale con scoping statico crea problemi rispetto alla modularità e alla compilabilità separata
 - PASCAL
- Soluzione migliore
 - ambiente locale, ambiente non locale con scoping statico e ambiente globale basato su un meccanismo di moduli



Implementazione dell'ambiente (ragione)

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 funzionepossiamo eliminare l'ambiente locale (e cose eventualmente associate) insieme a tutte le altre informazioni contenute nel record di attivazione



L'ambiente locale (una implementazione “furba”)

- 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 dalla sintassi del costrutto

Analisi statiche e ottimizzazioni

- Se lo scoping è statico, chi legge il programma e il compilatore possono
 - controllare che ogni riferimento a 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 alla 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 da 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 nomi

- Il numero di passi da fare a tempo di esecuzione 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

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'ambientenon 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