PROGRAMMAZIONE 2
23.

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
- Partendo come prima dalla semantica operazionale
- Molti aspetti li abbiamo gia' esaminati
- L'interprete e' parametrico rispetto all'ambiente

Linguaggio funzionale didattico

```
type ide = string
type exp =
     Eint of int                    (*costante intera*)
   | Ebool of bool                  (*costante booleana*)
   | Den of ide                     (*variabile*)
   | 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 *)
   | Apply of exp * exp list    (* Applicazione di funzione *)
```

Come e' fatto l'ambiente?

```
type 't env = ide -> 't;;
let emptyenv (v : 't) = function x -> v;;
let applyenv (r : 't env) (i : ide) = r i;;
let bind (r : 't env) (i : ide) (v : 't)=
                          function x -> if x = i
                                        then v else applyenv r x;;
```

Ambiente polimorfo!!! Come lo usiamo?

Valori esprimibili e ambiente

- Valori esprimibili

```
type evT = Int of int
         | Bool of bool
         | Unbound
```

- Ambiente

```
evT env
```

Primi passi (gia' fatto)

```
let rec eval (e: exp) (r: evT env) : evT =
       match e with
    | Eint n -> Int n
    | Ebool b -> Bool b
    | Den i -> applyenv r i
    | Eq(a, b) -> eq (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 ("non boolean guard")
```

`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!
- 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 operazionale: `Let`

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

Semanica operazionale: `Let`

run-time stack

push RA

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

Usciamo dal blocco
Effettuando pop

Interprete con **Let**

```
let rec eval (e: exp) (r: evT env) : evT =
     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) : evT =
      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

- astrazione funzionale
- applicazione di funzione

Sintassi: astrazione e applicazione

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

La sintassi OCaml `fun x e` viene rappresentata come

`Fun(x, e)`

La chiamata `sum 3` viene rappresentata come

`Apply(Den "sum", Int_e 3)`

Funzioni

- Identificatori (parametri formali) nel costrutto di astrazione

```
Fun of ide list * exp
```

- Espressioni (parameri attuali) nel costrutto di applicazione

```
Apply of exp * exp list
```

- Il passaggio dei parametri avviene per valore
  - le espressioni parametro attuale sono valutate e i valori ottenuti legati nell'ambiente al corrispondente parametro formale
- Per ora ignoriamo le funzioni ricorsive
- Assumeremo (per semplicità) solitamente di avere funzione unarie

Valori esprimibili?

- Come bisogna estendere i tipi esprimibli (**`evT`**) per comprendere le astrazioni funzionali?
  - Quale è il valore di una funzione?
- Assumiamo scoping statico (vedremo poi quello dinamico)

```
type evT = Int of int | Bool of bool | Unbound
         | Funval of efun

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 dichiarazioni

  I riferimenti non locali dell'astrazione verranno risolti nell'ambiente di dichiarazione

Astrazione e applicazione di funzione: scoping statico

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

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

Semantica eseguibile: scoping statico

```
let rec eval (e: exp) (r: evT env) : evT=
    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 la dichiarazione

Semantica operazionale vs. eseguibile

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

```
let rec eval (e: exp) (r: evT env) : evT=
    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")
```

Scoping dinamico

- Dobbiamo modificare **evT**

```
type evT =    Int of int | Bool of  bool | Unbound
            | Funval of efun

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

Semantica operazionale: scoping dinamico

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

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

Semantica eseguibile: scoping dinamico

```
let rec eval (e: exp) (r: evT env) : evT =
    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 qu 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 esis al momento nel quale avviene l'applicazione

- Nel **linguaggio didattico** adottiamo lo **scoping statico**

Funzioni ricorsive

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

Esempio: funzioni ricorsive

In Ocaml

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

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

Guardiamo la semantica

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

Letrec

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

```
type exp =
          :
      | Letrec of ide * ide * exp * exp
```

- `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: solito fattoriale

In Ocaml

```
let rec fact x = if (x == 0) then 1 else (x * fact (x-1))
                                                in fact (4)
```

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

Tipi esprimibili

- Estendiamo i tipi esprimibli (**`evT`**) per avere le astrazioni funzionali ricorsive (**`Recfunval`**)

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

Recfunval

```
Recfunval  of ide * ide * exp * evT env

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

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

Il codice dell'interprete: dichiarazione

```
 let rec eval (e: exp) (r: evT env) : evT =
      match e with
        …….
|Letrec(f, i, fBody, letBody) ->
    let benv =
       bind(r, f, (Recfunval(f, i, fBody, r)))
    in eval(letBody, benv)
```

Al nome della funzione ricorsiva viene associata una chiusura ricorsiva che contiene il nome della funzione stessa

Il codice dell'interprete: chiamata

```
let rec eval (e: exp) (r: evT env) : evT =
   match e with ….
     | Apply(e1, eArg) ->
           let fclosure = eval(e1, 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")
```

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*, ottendo l'ambiente **`rEnv`**
- L'ambiente effettivo di esecuzione **`fbody`** si ottiene estendendo l'ambiente **`rEnv`** con il binding del passaggio del parametro

Esempio

```
# let myRP =
   Letrec("fact",
     Fun("x", Ifthenelse(Eq(Den "x", Eint 0), Eint 1,
 Prod(Den "x", Apply(Den "fact",[Diff(Den "x", Eint 1)])))),
     Apply(Den "fact", [Eint 4]))

val myRP : exp = …

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

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