IMPLEMENTAZIONE DELL'INTERPRETE DI UN NUCLEO DI LINGUAGGIO FUNZIONALE

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

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

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 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 recordi di attivazione)
  - si sposa felicemente con la regola di scoping statico
    - ✓ per la gestione dell'ambiente non locale

4

Semanica operazionale di Let

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

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

Sintassi

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

Astrazione

Ricorsione

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 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 esprimibli (**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 dichiarazione

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

12

Semantica operazionale di astrazione e applicazione di funzione: **scoping statico**

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

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

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 operazionale vs. eseguibile

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

```
let rec 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 operazionale di astrazione e applicazione di funzione: **scoping dinamico**

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

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

Semantica eseguibile: scoping dinamico

```
let rec 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

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, seml(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

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!!!

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 dicorsive (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

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 soluzione della definizione

28

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

29

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

30

Definizioni ricorsive

- Consideriamo la funzione **f: N -> N** definita ricorsivamente nel modo seguente:

  f(n) = if (n = 0) then 0 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.
  - Come la abbiamo trovata?
  - Ce ne sono altre?

31

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?

32

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

  F: (N->N) -> (N->N)

- Riscriviamo le due equazioni:

  F(X)(0) = 0

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

  dove **X** deve essere di tipo **X: N -> N**

- Ora, le **soluzioni** del sistema originale sono esattamente i **punti fissi** del funzionale **F**, cioè le funzioni **k: N -> N** tali che **F(k) = k**

33

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)

34

Punto fisso con 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 $\mathbf{f_i = F(f_{i-1})}$
- Il punto fisso sarà il "limite" di questo procedimento

35

- f0(n) = indefinito
  - f0 = ∅
- f1(0) = 0, f1(n+1) = f0(n) + 2n + 1
  - f1 = {(0,0)}
- f2(0) = 0, f2(n+1) = f1(n) + 2n + 1
  - f2 = {(0,0), (1,1)}
- f3(0) = 0, f3(n+1) = f2(n) + 2n + 1
  - f3 = {(0,0), (1,1),(2,4)}

  :
- Limite di questa sequenza è la funzione

  g (x) = x*x

  che soddisfa le equazioni iniziali

36

Fixpoint iteration

Procedimento per ottenere un "minimo" punto fisso di un operatore

f = fix(F)
= f0, f1, f2, f3 ……
= -, F(f0), F(f1), F(f2), …..
= U_i F^i(-)

37

Fondamenti

- Diverse costruzioni definiscono le condizioni per l'esistenza dei punti fissi.
- **Teorema di Tarski**: Una funziona 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)
- Metodo per costruire la soluzione di una equazione di punto fisso nella definzione della semantica dei linguaggi di programmazione

39