

Classi e oggetti

1

Contenuti

- dai sottoprogrammi alle classi
 - oggetti come attivazioni permanenti, ambienti accessibili ovunque, entità con stato, strutture dati dinamiche
- ereditarietà (semplice) come annidamento di blocchi e sottoprogrammi
 - combinazione di modularità e scoping statico
- le classi sono anche tipi (semantica statica)
 - l'ereditarietà definisce sottotipi
- l'estensione object-oriented del linguaggio imperativo
 - sintassi
 - semantica operativa
 - dominio delle classi, `makeclass` e `applyclass`
 - interprete iterativo

2

Dai sottoprogrammi alle classi

- un sottoprogramma
 - oltre a definire una astrazione procedurale
 - permette di gestire dinamicamente ambiente e memoria
- una chiamata di sottoprogramma provoca la creazione di
 - un ambiente ed una memoria locale che esistono finché l'attivazione non ritorna
- e se vogliamo che l'ambiente e la memoria creati siano permanenti?
 - si può fare (in modo sporco) adottando l'ambiente locale statico, in cui ambiente e memoria
 - sono creati con la definizione della procedura
 - esistono solo per le diverse attivazioni di quella procedura
- in alternativa possiamo definire un meccanismo
 - che permetta di creare ambiente e memoria al momento della attivazione
 - in cui gli ambienti e memoria così creati
 - siano permanenti (sopravvivano alla attivazione)
 - una volta creati, siano accessibili ed utilizzabili da chiunque possieda il loro "manico"
 - l'oggetto che li contiene

3

Classi e loro istanziazione

- il "nuovo" sottoprogramma si chiama *classe*
 - può avere parametri (in Java no)
 - come un normale sottoprogramma contiene un blocco
 - lista di dichiarazioni
 - lista di comandi
- l'istanziazione (attivazione) della classe avviene attraverso il costrutto `new(classe, parametri_attuali)`
 - che può occorrere in una qualunque espressione
 - con cui si passano alla classe gli eventuali parametri attuali
 - che provoca la restituzione di un *oggetto*
- l'ambiente e la memoria locali dell'oggetto sono creati dalla valutazione delle dichiarazioni
 - le dichiarazioni di costanti e di variabili definiscono i campi dell'oggetto
 - se ci sono variabili, l'oggetto ha una memoria e quindi uno stato modificabile
 - le dichiarazioni di funzioni e procedure definiscono i metodi dell'oggetto
 - che vedono (e possono modificare) i campi dell'oggetto, semplicemente per la semantica dei blocchi
- l'esecuzione della lista di comandi è l'inizializzazione dell'oggetto

4

Oggetti

- l'oggetto è un manico che permette di accedere l'ambiente e la memoria locali creati permanentemente
 - attraverso l'accesso ai suoi metodi e campi
 - con l'operazione `Field(oggetto, identificatore)`
- nell'ambiente locale di ogni oggetto il nome speciale `this` denota l'oggetto medesimo

5

Classi, oggetti, tipi di dato

- le classi sono un meccanismo molto naturale per definire tipi di dato
 - soprattutto tipi di dato con stato (modificabile)
- la rappresentazione dei valori del tipo è data dall'insieme di campi
- le operazioni primitive del tipo di dato sono i metodi
 - che operano implicitamente sulla rappresentazione
- la creazione di un oggetto è la creazione di un valore del tipo
 - se ci sono variabili, l'oggetto ha uno stato modificabile
- se i campi non sono accessibili dall'esterno (privati) il tipo di dato è astratto

6

Oggetti e creazione dinamica di strutture dati

- la creazione di oggetti assomiglia molto (anche nella notazione sintattica) alla creazione dinamica di strutture dati
 - per esempio, in PASCAL e Crealizzata con operazioni come `new (tipo)`
 - che provoca l'allocazione dinamica di un valore di tipo `tipo` e la restituzione di un puntatore a tale struttura
- tale meccanismo prevede l'esistenza di una memoria a heap
 - simile a quella utilizzata per l'implementazione delle liste
 - simile a quella che useremo per implementare gli oggetti
- le strutture dati dinamiche sono un caso particolare di oggetti, ma ...
 - hanno una semantica ad-hoc non riconducibile a quella dei blocchi e delle procedure
 - la rappresentazione non è realizzata con campi separati
 - non ci sono metodi
 - non sono davvero permanenti
 - perchè esiste una (pericolosissima) operazione che permette di distruggere la struttura (`dispose`)

7

Ereditarietà 1

- il concetto di ereditarietà non è un componente essenziale del costrutto classe-oggetto
 - nasce in contesti diversi e lontani
 - tassonomie usate in rappresentazione della conoscenza
- ma si sposa bene con il concetto di oggetto arricchendolo in direzioni molto importanti dal punto di vista delle metodologie di programmazione
 - riusabilità, estendibilità, astrazione di insiemi di tipi tra loro collegati
- dal punto di vista dei tipi, permette di introdurre relazioni di sottotipo
 - arricchendo il sistema di tipi del linguaggio
 - rendendo più complessa la semantica statica (inferenza di tipi e/o loro verifica)
- a noi interessa riportare l'ereditarietà (semplice) ai concetti che già conosciamo legati all'ambiente

8

Ereditarietà 2

- la relazione di sottoclasse è semanticamente simile a quella di annidamento tra blocchi e sottoprogrammi
- se c1 è una sottoclasse di c2, le associazioni esistenti in una istanziazione di c1 sono
 - tutte quelle generate dalle dichiarazioni in c1 più
 - tutte quelle generate da dichiarazioni in c2 che non sono state ridefinite in c1
- è come se c1 fosse sintatticamente all'interno di c2 con una regola di scoping statico
 - con le classi la relazione è una relazione di sottoclasse fra frammenti di programmi separati
 - classi che possono essere compilate separatamente
 - con lo scoping statico ordinario per blocchi e procedure la relazione è di vera e propria inclusione sintattica
 - che impedisce la compilazione separata
- con i blocchi o le procedure annidate, se c'è una attivazione di c1, deve esistere già la corrispondente attivazione di c2
- con le classi questo non succede e quindi una istanziazione di c1 deve creare anche la corrispondente istanziazione di c2

9

OGGETTI NEL LINGUAGGIO DIDATTICO

10

Il linguaggio object-oriented: espressioni

```
type ide = string
type exp = Eint of int
         | Ebool of bool
         | :
         | Ifthenelse of exp * exp * exp
         | Let of ide * exp * exp
         | Newloc of exp
         | Fun of ide list * exp
         | Appl of exp * exp list
         | Rec of ide * exp
         | Proc of ide list * decl * com list
         | Field of exp * ide
         | New of ide * exp list
         | This
```

11

Dichiarazioni e comandi

```
and decl = (ide * exp) list * (ide * exp) list
and com =
  | Assign of exp * exp
  | Cifthenelse of exp * com list * com list
  | While of exp * com list
  | Block of decl * com list
  | Call of exp * exp list

and cdecl =
  Class of ide * ide list * (ide * ide list) * block
and prog = cdecl list * block
```

- notare che le dichiarazioni di classe possono solo occorrere nell'ambiente globale
- non c'è annidamento di classi
 - come in OCAML
 - in Java è possibile in forme limitate e comunque complesse

12

```

type exp = ...
  | Field of exp * ide
  | New of ide * exp list
  | This
and cdecl = Class of
  - ide *      nome della classe
  - ide list * lista di parametri formali
  - (ide * ide list) * (nome della superclasse, suoi parametri formali)
  - block     (campi, metodi, inizializzazione)

```

- diversa dalla procedura solo per la superclasse (ereditarietà)
 - "Object" indica la radice della gerarchia delle classi (senza parametri)
- istanziazione (attivazione) New of
 - ide * nome della classe
 - exp list lista di argomenti (parametri attuali)
- restituisce un oggetto che "contiene la semantica del blocco"
- accesso al contenuto dell'oggetto Field of
 - exp * espressione che valuta ad un oggetto
 - ide identificatore di campo o metodo
- la valutazione di This restituisce l'oggetto corrente

13

I valori degli oggetti

```

type exp = ...
  | New of ide * exp list
  | This

```

- un oggetto è interpretato su un dominio di valori `obj` che sono semplicemente *ambienti*
- i valori (denotabili, esprimibili e memorizzabili) con cui ci riferiamo agli oggetti sono di tipo `pointer`
- l'associazione tra `pointer` e `obj` è definita da un nuovo dominio semantico `heap`

```
type heap = pointer -> obj
```

che comparirà nel dominio e nel codominio di quasi tutte le funzioni di valutazione semantica

- Potremmo essere direttamente gli oggetti? No: il costrutto `This` implica la presenza del dominio `heap` (nella semantica!)

14

I valori delle classi

```
type cdecl =  
    Class of ide * ide list *  
           (ide * ide list) * block
```

- le classi possono
 - essere dichiarate
 - essere passate come parametri
 - essere utilizzate nell'espressione New
- le classi non possono essere restituite come valore di una espressione
- Modello semantico

```
type eclass = cdecl * dval env
```

15

I domini semantici: riassunto

```
type pointer = int  
and eval = ... | Object of pointer  
and dval = ... | Dobject of pointer  
           | Classval of eclass  
and mval = ... | Mobject of pointer  
and obj = dval env  
and heap = pointer -> obj  
and efun = expr * dval env  
and proc = expr * dval env  
and eclass = cdecl * dval env
```

16

Il dominio Heap

```
type pointer = int
and obj = dval env
and heap = pointer -> obj

let (newpoint,initpoint) = let count = ref(-1) in
  (fun () -> count := !count +1; !count),
  (fun () -> count := -1)

let emptyheap () = initpoint();
  ((function (x: pointer) -> emptyenv Unbound): heap)

let applyheap ((x: heap), (y:pointer)) = x y

let allocateheap ((x:heap), (i:pointer), (r:obj)) =
  ((function j -> if j = i then r else x j):heap)
```

17

makeclass, applyclass

```
type eclass = cdecl * dval env
let makeclass((c: cdecl), r) = Classval(c, r)
let rec applyclass ((evl:dval),(apars:dval list), s, h) = ( match evl with
| Classval(Class(name, fpars, extends, (b1,b2,b3) ),r) ->
  (match extends with
  | ("Object",_) -> let i = newpoint() in
    (let (r2, s2, h2) = semdl((b1, b2),
      (bindlist(r, fpars @ ["this"], apars @ [Dobject(i)])), s, h) in
    let (s3, h3) = semcl(b3, r2, s2, h2) in
    let r3 = localenv(r2, fpars, r) in
    let newh = allocateheap(h3, i, r3) in (Object i, s3, newh ))
  | (super,superpars) ->
    let (v, s1, h1) = applyclass(applyenv(r,super),
      findsuperargs(fpars, apars, superpars), s, h) in
    let n = (match v with | Object n1 -> n1) in
    let (r2, s2, h2) = semdl((b1, b2),
      (bindlist(eredita( r, v, h1), fpars, apars)), s1, h1) in
    let (s3, h3) = semcl(b3, r2, s2, h2) in
    let newh = allocateheap(h3, n, localenv( r2 ,fpars, r)) in
    (Object n, s3, newh))
| _ -> failwith("not a class"))
```

18

localenv, eredita

```
let localenv ((env1: dval env) ,(li:ide list), (envv: dval env)) =  
function (j:ide) -> (if notoccur(j, li) & applyenv(envv,j) = Unbound  
then env1 j else Unbound):(dval env)
```

- estrae da env1 la funzione che contiene tutte le associazioni (non presenti in envv) che non riguardano i parametri formali della classe
 - dato che le classi sono tutte dichiarate al top level, l'ambiente non locale dell'istanziamento (prima del passaggio di parametri) contiene solo dichiarazioni di classi

```
let eredita ((env1:dval env), Object(n), (h:heap)) =  
let r = applyheap(h, n) in  
function (i:ide) -> (if r i = Unbound then env1 i else r i):(dval env)
```

- eredita nell'ambiente contenuto nell'oggetto puntato da n le associazioni di env1 non ridefinite

19

Semantica delle espressioni 1

```
let rec sem ((e:exp), (r:dval env), (s: mval store), (h: heap)) = match e with  
| Eint(n) -> Int(n), s, h  
| Ebool(b) -> Bool(b), s, h  
| Den(i) -> dvaltoeval(applyenv(r,i)), s, h  
| Iszero(a) -> let (v1, s1, h1) = sem(a, r, s, h) in (iszero(v1), s1, h1)  
| Eq(a,b) -> let (v1, s1, h1) = sem(a, r, s, h) in  
let (v2, s2, h2) = sem(b, r, s1, h1) in (equ(v1, v2), s2, h2)  
| Prod(a,b) -> let (v1, s1, h1) = sem(a, r, s, h) in  
let (v2, s2, h2) = sem(b, r, s1, h1) in (mult(v1, v2), s2, h2)  
| Sum(a,b) -> let (v1, s1, h1) = sem(a, r, s, h) in  
let (v2, s2, h2) = sem(b, r, s1, h1) in (plus(v1, v2), s2, h2)  
| Diff(a,b) -> let (v1, s1, h1) = sem(a, r, s, h) in  
let (v2, s2, h2) = sem(b, r, s1, h1) in (diff(v1, v2), s2, h2)  
| Minus(a) -> let (v1, s1, h1) = sem(a, r, s, h) in (minus(v1), s1, h1)  
| And(a,b) -> let (v1, s1, h1) = sem(a, r, s, h) in  
let (v2, s2, h2) = sem(b, r, s1, h1) in (et(v1, v2), s2, h2)  
| Or(a,b) -> let (v1, s1, h1) = sem(a, r, s, h) in  
let (v2, s2, h2) = sem(b, r, s1, h1) in (vel(v1, v2), s2, h2)  
| Not(a) -> let (v1, s1, h1) = sem(a, r, s, h) in (non(v1), s1, h1)
```

20

Semantica delle espressioni 2

```
| Ifthenelse(a,b,c) -> let (g, s1, h1) = sem(a, r, s, h) in
  if typecheck("bool",g) then
    (if g = Bool(true) then sem(b, r, s, h1) else sem(c, r, s, h1))
  else failwith ("nonboolean guard")
| Val(e) -> let (v, s1, h1) = semden(e, r, s, h) in (match v with
  | Dloc n -> mvaltoeval(applystore(s1, n)), s1, h1
  | _ -> failwith("not a variable"))
| Let(i,e1,e2) -> let (v, s1, h1) = semden(e1, r, s, h) in
  sem(e2, bind (r,i, v), s1, h1)
| Fun(i,e1) -> dvaltoeval(makefun(e,r)), s, h
| Rec(i,e1) -> makefunrec(i, e1, r), s, h
| Appl(a,b) -> let (v1, s1, h1) = semlist(b, r, s, h) in
  applyfun(evaltodval(sem(a, r, s, h1)), v1, s1, h1)
| New(i,ge) -> let (v, s1, h1) = semlist(ge, r, s, h) in
  applyclass(applyenv(r,i), v, s1, h1)
| This -> (dvaltoeval(applyenv(r,"this")), s, h)
| _ -> failwith ("nonlegal expression for sem")

val sem : exp * dval env * mval store * heap -> eval * mval store * heap = <fun>
```

21

Semantica delle espressioni 3

```
and semden((e:exp), (r:dval env), (s: mval store), (h: heap)) = match e
with
  | Den(i) -> (applyenv(r,i), s, h)
  | Fun(i, e1) -> (makefun(e, r), s, h)
  | Proc(i, b) -> (makeproc(e, r), s, h)
  | Newloc(e) -> let (v, s1, h1) = sem(e, r, s, h) in
    let m = evaltomval v in
    let (l, s2) = allocate(s1, m) in (Dloc l, s2, h1)
  | Field(e,i) -> (match sem(e, r, s, h) with
    | (Object i1,s1,h1) -> let r1 = applyheap(h1, i1) in
      let field = applyenv(r1,i) in (field, s1, h1)
    | _ -> failwith("notanobject"))
  | _ -> let (v, s1, h1) = sem(e, r, s, h) in
    let d = evaltodval v in (d, s1, h1)
and semlist(el, r, s, h) = match el with
  | [] -> ([], s, h)
  | e::e1 -> let (v1, s1, h1) = semden(e, r, s, h) in
    let (v2, s2, h2) = semlist(e1, r, s1, h1) in (v1 :: v2, s2, h2)
val semden : exp * dval env * mval store * heap -> dval * mval store * heap
= <fun>
val semlist : exp list * dval env * mval store * heap -> (dval list) * mval
store * heap = <fun>
```

22

Semantica di comandi

```

and semc((c: com), (r:dval env), (s: mval store), (h:heap)) = match c with
| Assign(e1, e2) -> let (v1, s1, h1) = semden(e1, r, s, h) in (match v1 with
| Dloc(n) -> update(s, n, evaltomval(sem(e2, r, s1, h1)))
| _ -> failwith ("wrong location in assignment"))
| Cifthenelse(e, c11, c12) -> let (g, s1, h1) = sem(e, r, s, h) in
if typecheck("bool",g) then
(if g = Bool(true) then semc1(c11, r, s1, h1) else semc1(c12, r, s1, h1))
else failwith ("nonboolean guard")
| While(e, c1) ->
let (g, s1, h1) = sem e r s h in
if typecheck("bool",g) then
(if g = Bool(true) then semc1(c1 @ [While(e, c1)]), r, s, h) else (s, h)
else failwith ("nonboolean guard")
| Call(e1, e2) -> let (p, s1, h1) = semden(e1, r, s, h) in let (v, s2, h2) = semlist(e2, r, s1, h1)
in

applyproc(p, v, s2, h2)
| Block(b) -> semb(b, r, s, h)

and semc1(cl, r, s, h) = match cl with
| [] -> s, h
| c::cl1 -> let (s1,h1) = semc(c, r, s, h) in semc1(cl1, r, s1, h1)

val semc : com * dval env * mval store * heap -> mval store * heap = <fun>
val semc1 : com list * dval env * mval store * heap -> mval store * heap = <fun>

```

23

Semantica di blocchi e dichiarazioni

```

and semb((dl, rdl, cl), r, s, h) =
let (r1, s1, h1) = semdl((dl, rdl), r, s, h) in semc1(cl, r1, s1, h1)
and semdl((dl, rl), r, s, h) = let (r1, s1, h1) = semdv(dl, r, s, h) in
semr(r1, r1, s1, h1)
and semdv(dl, r, s, h) = match dl with
| [] -> (r, s, h)
| (i,e)::dll -> let (v, s1, h1) = semden(e, r, s, h) in
semv(dll, bind(r, i, v), s1, h1)
and semr(rl, r, s, h) =
let functional ((r1: dval env)) = (match rl with
| [] -> r
| (i,e) :: r11 -> let (v, s2, h2) = semden(e, r1, s, h) in
let (r2, s3, h3) = semr(r11, bind(r, i, v), s, h) in r2 in
let rec rfix = function x -> functional rfix x in (rfix, s, h)
val semb : (decl * com list) * dval env * mval store * heap ->
mval store * heap = <fun>
val semdl : decl * dval env * mval store * heap ->
dval env * mval store * heap = <fun>
val semdv : (ide * expr) list * dval env * mval store * heap ->
dval env * mval store * heap = <fun>
val semr : (ide * expr) list * dval env * mval store * heap ->
dval env * mval store * heap = <fun>

```

24

Dichiarazioni di classe e programmi

```
and semclasslist(c1, (r: dval env )) =
  let functional (r1: dval env) = (match c1 with
    | [] -> r
    | Class(nome,x1,x2,x3)::c11 -> semclasslist(c11, bind(r,nome,
      makeclass(Class(nome,x1,x2,x3), r1)))) in
    let rec rfix = function i -> functional rfix i
    in rfix

and semprog((cd1,b), r, s, h) = semb(b, semclasslist(cd1, r), s, h)

val semclasslist : cdecl list * dval env -> dval env = <fun>
val semprog : (cdecl list * block) * dval env * mval store * heap ->
  mval store * heap = <fun>
```

- le dichiarazioni di classe sono trattate come mutuamente ricorsive
 - da ognuna di esse si vedono tutte le altre

25

Un esempio con ereditarietà 1

```
# let (puntil: cdecl list) = [
  Class("point",["a";"b"],("Object",[]),
    [{"x",Newloc(Den "a")};{"y", Newloc(Den "b")}],
    [{"getx", Fun([],Val(Den "x"))};
     {"gety", Fun([],Val(Den "y"))};
     {"move",Proc(["c";"d"],([],[],
       [Assign(Den "x", Sum(Val(Den "x"),Den "c"));
        Assign(Den "y", Sum(Val(Den "y"),Den "d"))]))}],
    []));
  Class("point3",["a";"b";"c"],("point",["a";"b"]),
    [{"z",Newloc(Den "c")}],
    [{"getz", Fun([],Val(Den "z"))};
     {"move", Proc(["c";"d";"e"], ([],[],
       [Assign(Den "x", Sum(Val(Den "x"),Den "c"));
        Assign(Den "z", Sum(Val(Den "z"),Den "e"));
        Assign(Den "y", Sum(Val(Den "y"),Den "d"))]))}],
    [{"projectxy",Fun([], New("point",[Val(Den "x");Val(Den "y")]))}],
    []]);
# let a1 = semclasslist (puntil, emptyenv Unbound);;
val a1 : dval env = <fun>
# let (dichiarazionil: (ide * exp) list) =
  [{"punto3_1", Newloc(New("point3",[Eint 1; Eint 3; Eint 105]))};
   {"punto3_2", Newloc(Val(Den "punto3_1"))}];;
# let (a2,s2,h2) = semdv(dichiarazionil, a1, emptystore Undefined, emptyheap());;
val a2 : dval env = <fun>
val s2 : mval store = <fun>
val h2 : heap = <fun>
```

26

Un esempio con ereditarietà 2

```
# let (comandil: com list) =
  [Call(Field(Val(Den "punto3_1"), "move"), [Eint 38; Eint 43; Eint 1012]));;
# let (s3, h3) = semcl(comandil, a2, s2, h2);;
val s3 : mval store = <fun>
val h3 : heap = <fun>
# let (newd2: (ide * exp) list) =
  [("punto2_1", Appl(Field(Val(Den "punto3_2"), "projectxy"), []))];;
# let (a3, s4, h4) = semdv(newd2, a2, s3, h3);;
val a3 : dval env = <fun>
val s4 : mval store = <fun>
val h4 : heap = <fun>
# let expr1 = Appl(Field(Den "punto2_1", "getx"), []);;
# sem(expr1, a3, s4, h4);;
- : eval * mval store * heap = Int 39, <fun>, <fun>
```

27

Come eliminiamo la ricorsione

- non servono strutture dati diverse da quelle già introdotte per gestire i blocchi e le procedure
 - la istanziazione di classe crea un nuovo frame in cui valutare il corpo della classe
 - dopo avere eventualmente creato frames per le superclassi
- servono nuovi costrutti etichettati per le classi

type labeledconstruct =

```
| Ogg1 of cdecl
| Ogg2 of cdecl
| Ogg3 of cdecl
```

- pila dei records di attivazione realizzata attraverso sei pile gestite in modo "parallelo"
 - envstack, cstack, tempvalstack, tempdvalstack, storestack, labelstack
- la heap è globale ed è gestita da una variabile (di tipo heap) currentheap

28

makeclass, applyclass

```
let makeclass((c: cdecl), r) = Classval(c, r)
let rec applyclass ((ev1:dval),(apars:dval list), s, h) =
  ( match ev1 with
  | Classval(Class(name, fpars, extends, (b1,b2,b3) ),r) ->
    let j = newpoint() in
      newframes(Ogg1(Class(name, fpars, extends,
        (b1,b2,b3) )),
        bindlist(r, fpars @ ["this"], apars @
          [Dobject(j)]), s)
  | _ -> failwith("not a class"))
```

29

L'interprete iterativo 1

```
let itsem() =
  let continuation = top(cstack) in
  let tempstack = top(tempvalstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
  (match top(continuation) with
  | Expr1(x) ->
    (pop(continuation); push(Expr2(x),continuation);
    (match x with
    | ...
    | New(i,ge) -> pushargs(ge, continuation)
    | _ -> ()))
  | Expr2(x) ->
    (pop(continuation); (match x with
    | Eint(n) -> push(Int(n),tempstack)
    | New(i,ge) -> let arg=getargs(ge,tempdstack) in
      applyclass(applyenv(rho,i), arg, sigma, !currentheap)
      | This -> push(dvaltoeval(applyenv(rho,"this")), tempstack)
    | _ -> failwith("no more cases for itsem")))
  | _ -> failwith("no more cases for itsem"))
val itsem : unit -> unit = <fun>
```

30

L'interprete iterativo 2

```
let itsemnden() =
  let continuation = top(cstack) in
  let tempstack = top(tempvalstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
  (match top(continuation) with
   | Exprd1(x) -> (pop(continuation); push(Exprd2(x), continuation);
    match x with
    | ...
    | Field(e,i) -> push(Expr1(e), continuation)
    | _ -> push(Expr2(x), continuation))
   | Exprd2(x) -> (pop(continuation); match x with
    | ...
    | Field(e,i) -> let ogg = top(tempstack) in pop(tempstack);
      (match ogg with
       | Object il -> let r1 = applyheap(!currentheap, il) in
         let field = applyenv(r1,i) in push(field, tempdstack)
       | _ ->
         failwith("notanobject")))
    | _ -> let arg = top(tempstack) in pop(tempstack);
      push(evaltodval(arg), tempdstack))
    | _ -> failwith("No more cases for semden")
  )
val itsemnden : unit -> unit = <fun>
```

31

L'interprete iterativo 3

```
let itsemobj() =
  let continuation = top(cstack) in
  let tempstack = top(tempvalstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
  (match top(continuation) with
   | Ogg1(Class(name, fpars, extends, (b1,b2,b3) )) -> pop(continuation);
    (match extends with
     | ("Object",_) -> push(Ogg3(Class(name, fpars, extends, (b1,b2,b3) )), continuation);
      push(labelcom(b3), top(cstack));
      push(Rdecl(b2), top(cstack));
      push(labeldec(b1), top(cstack))
     | (super, superpars) ->
      let lobj = applyenv(rho, "this") in
      let superargs = findsuperargs(fpars, dlist(fpars, rho), superpars) in
      push(Ogg2(Class(name, fpars, extends, (b1,b2,b3) )), continuation);
      (match applyenv(rho, super) with
       | Classval(Class(snome, superfpars, sextends, sb), r) ->
        newframes(Ogg1(Class(snome, superfpars, sextends, sb)),
         bindlist(r, superfpars @ ["this"], superargs @ [lobj]), sigma)
       | _ -> failwith("not a superclass name")))
  )
```

32

L'interprete iterativo 4

```
let itsemobj() =
  let continuation = top(cstack) in
  let tempstack = top(tempvalstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
  (match top(continuation) with
   | ....
   | Ogg2(Class(name, fpars, extends, (b1,b2,b3) )) ->
     pop(continuation); let v = top(tempstack) in pop(tempstack);
     let newenv = eredita(rho, v, !currentheap) in
     popenv(); pushenv(newenv);
     push(Ogg3(Class(name, fpars, extends, (b1,b2,b3) ),continuation);
     push(labelcom(b3), top(cstack));
     push(Rdecl(b2), top(cstack));
     push(labeldec(b1),top(cstack))
   | Ogg3(Class(name, fpars, extends, (b1,b2,b3) )) ->
     pop(continuation); let r = (match applyenv(rho,name) with
     | Classval(_, r1) -> r1
     | _ -> failwith("not a class name")) in
     let lobj = (match applyenv(rho, "this") with | Dobject n -> n) in
     let newenv = localenv(rho, fpars, r) in
     currentheap := allocateheap (!currentheap, lobj, newenv);
     push(Object lobj, tempstack)
     | _ -> failwith("impossible in semobj"))
  val itsemobj : unit -> unit = <fun>
```

33

L'interprete iterativo 5

```
let loop () =
  while not(empty(cstack)) do
    while not(empty(top(cstack))) do
      let currconstr = top(top(cstack)) in
      (match currconstr with
       | Expr1(e) -> itsem()
       | Expr2(e) -> itsem()
       | Exprd1(e) -> itsemden()
       | Exprd2(e) -> itsemden()
       | Coml(c1) -> itsemcl()
       | Rdecl(l) -> itsemrdecl()
       | Decl(l) -> itsemdecl()
       | Ogg1(e) -> itsemobj()
       | Ogg2(e) -> itsemobj()
       | Ogg3(e) -> itsemobj()
       | _ -> failwith("non legal construct in loop"))
    done;
    (match top(labelstack) with
     | Expr1(_) -> let valore = top(top(tempvalstack)) in
       pop(top(tempvalstack)); pop(tempvalstack); push(valore,top(tempvalstack));
       popenv(); popstore(); pop(tempdvalstack)
     | Exprd1(_) -> let valore = top(top(tempdvalstack)) in
       pop(top(tempdvalstack)); pop(tempdvalstack); push(valore,top(tempdvalstack));
       popenv(); popstore(); pop(tempvalstack)
     | Decl(_) -> pop(tempvalstack); pop(tempdvalstack)
     | Rdecl(_) -> pop(tempvalstack); pop(tempdvalstack)
     | Coml(_) -> let st = topstore() in popenv(); popstore(); popstore(); pushstore(st);
       pop(tempvalstack); pop(tempdvalstack)
     | Ogg1(_) -> let valore = top(top(tempvalstack)) in
       pop(top(tempvalstack)); pop(tempvalstack); push(valore,top(tempvalstack));
       popenv(); popstore(); pop(tempdvalstack)
     | _ -> failwith("non legal label in loop"));
    pop(cstack); pop(labelstack)
  done
val loop : unit -> unit = <fun>
```

34

L'interprete iterativo 6

```
let sem (e,(r: dval env), (s: mval store), (h: heap)) = initState(); currentheap := h;
push(emptystack(tframesize(e),Noval),tempvalstack);
pushstore(emptystore(Undefined));
newframes(Expr1(e), r, s);
loop();
let st = topstore() in popstore();
let valore= top(top(tempvalstack)) in
pop(tempvalstack);
(valore, st, !currentheap)
val sem : exp * dval env * mval store * heap -> eval * mval store * heap = <fun>
let semden (e,(r: dval env), (s: mval store), (h: heap)) = initState(); currentheap := h;
push(emptystack(tframesize(e),Unbound),tempdvalstack);
pushstore(emptystore(Undefined));
newframes(Exprd1(e), r, s);
loop();
let st = topstore() in popstore();
let valore= top(top(tempdvalstack)) in
pop(tempdvalstack);
(valore, st, !currentheap)
val semden : exp * dval env * mval store * heap -> dval * mval store * heap = <fun>
let semcl (cl,(r: dval env), (s: mval store), (h: heap)) = initState(); currentheap := h;
pushstore(emptystore(Undefined));
newframes(labelcom(cl), r, s);
loop();
let st = topstore() in popstore();
(st, !currentheap)
val semcl : com list * dval env * mval store * heap -> mval store * heap = <fun>
```

35

L'interprete iterativo 7

```
let semdv(dl, r, s, h) = initState(); currentheap := h;
newframes(labeldec(dl), r, s);
loop();
let st = topstore() in popstore();
let rt = topenv() in popenv();
(rt, st, !currentheap)
val semdv : (ide * exp) list * dval env * mval store * heap ->
dval env * mval store * heap = <fun>
let semc(c:(com), (r:dval env), (s: mval store), h) = initState();
pushstore(emptystore(Undefined));
currentheap := h;
newframes(labelcom([c]), r, s);
loop();
let st = topstore() in popstore();
(st, !currentheap)
val semc : com * dval env * mval store * heap -> mval store * heap = <fun>
```

- analoghe le altre!

36

L'interprete iterativo 8

```
let semclasslist (cl, ( r: dval env )) =
  let rcl = ref(cl) in
  let rrr = ref(r) in
  let functional rr =
    while not(!rcl = []) do
      let thisclass = List.hd !rcl in
      rcl := List.tl !rcl;
      match thisclass with
      | Class(nome,_,_,_) -> rrr := bind(!rrr, nome, makeclass(thisclass,
rr))
    done;
    !rrr in
  let rec rfix = function i -> functional rfix i
  in rfix
val semclasslist : cdecl list * dval env -> dval env = <fun>
```

- il funzionale è definito in modo iterativo, ma c'è sempre un calcolo di punto fisso