Implementazione degli oggetti

Contenuti

- effetto dell'implementazione di ambiente e memoria sugli oggetti
  - oggetti come ambiente e memoria permanente
- effetto degli oggetti sull'implementazione di ambiente e memoria
  - ambienti e memoria esistono sia sulla pila che sulla heap
- alcune modifiche nell'interprete iterativo

Oggetti e implementazione dello stato

- nella semantica attuale, un oggetto è un ambiente
  - come sempre rappresentato da una funzione
  - ottenuto a partire dall'ambiente complessivo esistente dopo l'esecuzione del blocco
  - limitandosi (`localenv`) alle associazioni per `this`, campi e metodi (inclusi quelli ereditati)
- tale ambiente è utilizzato soltanto per selezionare campi e metodi (`Field`)
- l'ambiente non-locale dell'oggetto
  - è soltanto l'ambiente globale delle classi, ma potrebbe essere un ambiente qualunque se permettessimo classi annidate

  è rappresentato completamente nelle chiusure dei suoi metodi
- la memoria esportata nella posizione della pila corrispondente all'attivazione precedente contiene tutto
  - anche le locazioni locali ad attivazioni ritornate non accessibili perchè non denotate nell'ambiente
  - anche le locazioni locali dell'oggetto che restano accessibili (permanenti) in quanto denotate nell'ambiente-oggetto (che rimane nella heap)
- quando l'ambiente e la memoria vengono implementati come pile di ambienti e memorie locali, bisogna adattare anche l'implementazione dell'oggetto
  - ambiente locale
  - memoria locale

La nuova struttura degli oggetti

```
type obj = ide array * dval array * dval env * mval array
```

- come negli ambienti e memorie locali
  - `ide array`, array di identificatori, come gli elementi di `namestack`
  - `dval array`, array di valori denotati, come gli elementi di `dvalstack`
  - `dval env`, (puntatore ad) ambiente, come gli elementi di `slinkstack`
  - `mval array`, (puntatore a) memoria, come gli elementi di `storestack`
  - non serve un tag per la retention, perché l'ambiente locale così costruito è permanente

La nuova heap

- come nel caso di ambiente e memoria anche la heap non può più essere una funzione
- da

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

  a

```
type heap = obj array
```

- mantenendo per ora l'implementazione "banale" di `pointer` e `newpoint`

```
type pointer = int
let newpoint =   let count = ref(-1) in
   function () -> count := !count +1; !count
```

Operazioni sulla heap

```
let emptyheap () = initpoint();
      (Array.create 100
      ((Array.create 1 "dummy"),(Array.create 1 Unbound),
                Denv(-1), (Array.create 1 Undefined)) : heap)

let currentheap = ref(emptyheap())

let applyheap ((x: heap), (y:pointer)) = Array.get x y

let allocateheap ((x:heap), (i:pointer), (r:obj)) =
    Array.set x i r; x

let getnomi(x:obj) = match x with
  | (nomi, den, slink, st) -> nomi

let getden(x:obj) = match x with
  | (nomi, den, slink, st) -> den

let getslink(x:obj) = match x with
  | (nomi, den, slink, st) -> slink

let getst(x:obj) = match x with
  | (nomi, den, slink, st) -> st
```

Ambiente e memoria devono cambiare!

- nel linguaggio imperativo, ambiente e memoria erano semplicemente interi
  - interpretati come puntatori alle tabelle locali nelle varie pile che realizzano l'ambiente e la memoria
- in presenza di oggetti, ambienti e memorie locali possono essere permanenti essendo allocate sulla heap
- possiamo astrarre dalle differenze e definire un solo tipo di ambiente e memoria

```
type 't env = Denv of int | Penv of pointer
type 't store =  Dstore of int | Pstore of pointer
```

  - le versioni che iniziano con `D` (dinamiche) sono puntatori nelle pile
  - le versioni che iniziano con `P` (permanenti) sono puntatori nella heap
- le implementazioni delle operazioni si preoccuperanno di trattare i due casi in modo appropriato
  - mostriamo nel seguito solo un paio di operazioni ridefinite

Cercare il valore denotato da un nome

```
let applyenv ((r: dval env), (y: ide)) =
    let den = ref(Unbound) in
    let (x, caso) = (match r  with
      | Denv(x1) -> (x1, ref("stack"))
      | Penv(x1) -> (x1, ref("heap"))) in
       let n = ref(x) in
       while !n > -1 do
       let lenv =
       if !caso = "stack"
         then access(namestack,!n)
         else getnomi(applyheap(!currentheap,!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 := (if !caso = "stack"
              then Array.get (access(dvalstack,!n)) !index
              else Array.get (getden(applyheap(!currentheap,!n))) !index);
              index := nl)
             else index := !index + 1
           done;
            if not(!den = Unbound) then  n := -1
       else  let next =  (if !caso = "stack" then access(slinkstack,!n)
           else getslink(applyheap(!currentheap,!n))) in
           caso := (match next with
             |Denv(_) -> "stack"
             |Penv(_) -> "heap");
           n := (match next with
             |Denv(n1) -> n1
             |Penv(n1) -> n1)
       done;
       !den
```

Le operazioni sulla memoria

```
let applystore ((x: mval store), (d: loc)) =
   match d with
     | (s2,n2)->
       (match s2 with
          | Dstore(n1) -> let a = access(storestack, n1) in
              Array.get a n2
          | Pstore(n1) -> let a = getst(applyheap(!currentheap,n1)) in
                  Array.get a n2)
     | _ -> failwith("not a location in applystore")

let allocate ((s:mval store),  (m:mval)) =
    let (s2, n2) = newloc() in
    (match s2 with
      | Pstore(n1) -> let a = access(storestack, n1) in
             Array.set a n2 m; ((s2, n2),s)
      | Dstore(n1) -> let a = access(storestack, n1) in
                  Array.set a n2 m;  ((s2, n2),s))

let update((s:mval store),  (d: loc),  (m:mval)) =
    if applystore(s, d) = Undefined then failwith ("wrong assignment")
     else  match d with
     | (s2,n2) ->
      (match s2 with
          | Dstore(n1) -> let a = access(storestack, n1) in
              Array.set a n2 m; s
          | Pstore(n1) -> let a = getst(applyheap(!currentheap,n1)) in
                  Array.set a n2 m;  s )
```

L'ereditarietà

```
let eredita ((rho: dval env), ogg, (h: heap)) =
   let currn = (match rho with | Denv(n) -> n) in
   let (point, arrnomisotto, arrdensotto, arrstore) = (match ogg with
     | Object(n) -> let oo = applyheap(!currentheap,n) in (n, getnomi(oo), getden(oo), getst(oo))
     | _ -> failwith("not an object in eredita")) in
   let stl = Array.length arrstore in
   let newstore = Array.create stl Undefined in let index = ref(0) in
   while !index < stl do
     Array.set newstore !index (Array.get arrstore !index);
     index := !index + 1   done;
   pop(storestack); push(newstore, storestack);
   let currarrnomi = access(namestack, currn) in
   let currarrden = access(dvalstack, currn) in
   let r = access(slinkstack, currn) in
   let currl = Array.length currarrnomi in
   let oldlen = Array.length arrnomisotto in index := 0;
   while not(Array.get arrnomisotto !index = "this") do index := !index + 1 done;
   index := !index + 1;
   let newlen = (currl + oldlen - !index )  in
   let newarrnomi = Array.create newlen  "dummy" in
   let newarrden = Array.create newlen  Unbound in let newindex = ref(0) in
   while !newindex < currl do
     Array.set newarrnomi !newindex (Array.get currarrnomi !newindex);
     Array.set newarrden !newindex (Array.get currarrden !newindex);
     newindex := !newindex + 1
   done;
   while !newindex < newlen do
     Array.set newarrnomi !newindex (Array.get arrnomisotto !index);
     Array.set newarrden !newindex (Array.get arrdensotto !index);
     newindex := !newindex + 1;
     index := !index + 1
   done;
   pop(namestack);pop(dvalstack);pop(slinkstack);
   push(newarrnomi, namestack); push(newarrden,dvalstack); push(r,slinkstack)
```

Localenv

```
let localenv ((r:dval env) , (s: mval store), Dobject(ob), (li:ide list), (r1: dval env)) =
    let (rint, sint) = (match (r,s) with      | (Denv nr, Dstore ns) -> (nr, ns)
      | _ -> failwith("heap structures in localenv")) in
    let oldst = access(storestack, sint) in
    let oldnomi = access(namestack, rint) in
    let oldden = access(dvalstack, rint) in
    let storesize = Array.length oldst in
    let newst = Array.create storesize Undefined in
    let index = ref(0) in
    while not(!index = storesize) do
       Array.set newst !index (Array.get oldst !index);
       index := !index + 1
    done;
    let oldenvlength = Array.length oldnomi in
    let newenvlength = oldenvlength - (List.length li) in
    let newnomi = Array.create newenvlength "dummy" in
    let newden = Array.create newenvlength Unbound in
    let index = ref(0) in
    let newindex = ref(0) in
    while not(!index = oldenvlength) do
      let lname = Array.get oldnomi !index in
      let lden = Array.get oldden !index in
      if notoccur(lname, li) then (
      Array.set newnomi !newindex lname;
      let ldennuova = (match lden with
        | Dfunval(e,rho) -> if rho >= r then Dfunval(e,Penv(ob)) else lden
     | Dprocval(e,rho) -> if rho >= r then Dprocval(e,Penv(ob)) else lden
     | Dloc(sigma, n) -> if sigma >= s then Dloc(Pstore(ob),n) else lden
     |  _ -> lden              ) in
      Array.set newden !newindex ldennuova;
      newindex := !newindex + 1)
      else ();
      index := !index + 1
    done;
    (newnomi, newden, r1, newst)
```

Cosa cambia nell'interprete iterativo 1

- numerosi piccoli adattamenti relativi alla introduzione dei due tipi di ambiente e memoria
- qualche modifica più importante nel trattamento degli oggetti

```
| 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));
        pushlocalenv(b1,b2,rho);
        pushlocalstore(b1); ()
     | (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")))
```

Cosa cambia nell'interprete iterativo 2

```
| Ogg2(Class(name, fpars, extends, (b1,b2,b3) )) ->
     pop(continuation);
     let v = top(tempstack) in
     pop(tempstack);
     eredita(rho, v, !currentheap);
     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));
     pushlocalenv(b1,b2,rho);
     pushlocalstore(b1);  ()

| 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 newobj = localenv(rho, sigma, Dobject(lobj), fpars, r) in
    currentheap := allocateheap (!currentheap, lobj, newobj);
    push(Object lobj, tempstack)
```