Gestione dinamica della memoria a heap (nel linguaggio orientato ad oggetti)

Contenuti

- heap di oggetti, allocazione di oggetti con lista libera, restituzione di oggetti
- gestione della heap da parte del sistema: garbage collection
- digressione su altri meccanismi per gestire la heap
  - la gestione esplicita della restituzione è pericolosa
  - contatori di riferimenti
  - problemi con heap disomogenea

Heap e gestione dinamica della memoria

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

- nella implementazione corrente, gli oggetti allocati sulla heap sono realmente permanenti, poiché non esiste alcun modo di “disallocarli”
- nella tradizionale gestione della memoria a heap
  - come quella che abbiamo visto per le liste

  la heap è gestita non come un “banale” array sequenziale ma attraverso una lista libera
  - le allocazioni sono fatte prendendo il puntatore dalla lista libera
  - esiste una operazione per restituire un elemento alla lista libera
- adattiamo l’implementazione della heap vista per le liste al caso in cui gli elementi da allocare sono oggetti

La nuova heap

```
let heapsize = 6
```

- la heap

```
let objects = (Array.create heapsize ((Array.create 1 "dummy"),(Array.create 1
   Unbound), Denv(-1), (Array.create 1 Undefined)) : heap)
```

- l'array parallelo utilizzato per gestire la lista libera

```
let nexts = Array.create heapsize (-1: pointer)
```

- il puntatore alla testa della lista libera

```
let next = ref((0: pointer))
```

- la heap iniziale (solo lista libera!)

```
let emptyheap() = let index = ref(0) in
    while !index < heapsize do
      Array.set nexts !index (!index + 1); Array.set marks !index false;
      index := !index + 1    done;
    Array.set nexts (heapsize - 1) (-1); next := 0;
    svuota (markstack); objects
```

Le operazioni sulla heap

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

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

let disallocate (i:pointer) =
    let pre = !next in
    next := i;
    Array.set nexts i pre
```

La disallocazione

- nel linguaggio didattico (come in Java ed OCAML) la disallocazione non è prevista come operazione a disposizione del programmatore
- è una operazione eventualmente invocata dal sistema (implementazione!) quando la lista libera diventa vuota e non permette di allocare un nuovo oggetto
  - gli oggetti che non sono più utilizzati vengono disallocati
  - lo spazio nella heap da loro occupato è utilizzato per allocare nuovi oggetti
  - gli oggetti continuano ad essere logicamente permanenti, perché vengono eventualmente distrutti solo quando non servono più
  - gli oggetti che "non servono più" vengono determinati con una complessa procedura *(marcatura)* il cui effetto è quello di "marcare" tutti gli oggetti che servono ancora

Dopo la marcatura

- la marcatura setta a true il valore di un terzo array parallelo a quello di oggetti

```
let marks = Array.create heapsize false
```

- una volta effettuata la marcatura, tutti gli oggetti non marcati vengono disallocati, restituendoli alla lista libera (garbage collection!)

```
let collect = function () ->
        let i = ref(0) in
        while !i < heapsize do
          (if Array.get marks !i then (Array.set marks !i false)
           else disallocate(!i));
          i := !i +1
        done
```

Verso la marcatura

- l'obiettivo è quello di marcare tutti gli *oggetti attivi*
  - oggetti raggiungibili a partire dalle strutture che realizzano la pila dei records di attivazione
    - eventualmente passando attraverso altri oggetti attivi
- è necessario visitare le strutture a grafo costituite da puntatori, radicate in strutture esterne alla heap stessa (ambiente, memoria, temporanei)
- per visitare tale struttura è necessario disporre di una "pila per la marcatura" `markstack`
- dopo aver introdotto tale struttura, vedremo la procedura `markobject` che gestisce la visita della struttura di puntatori
- vedremo infine la procedura `startingpoints` che determina (e inserisce in `markstack`) tutti i puntatori contenuti in strutture esterne alla heap (punti di partenza del garbage collector)

Le strutture per la marcatura

```
let markstacksize = 100

let markstack = emptystack(markstacksize, (0:pointer))

let pushmarkstack (i: pointer) =
  if lungh(markstack) = markstacksize
   then failwith("markstack length has to be increased")
  else push(i, markstack)
```

Marcare un oggetto

```
let markobject (i: pointer) =
   if Array.get marks i then ()
   else
    (Array.set marks i true;
    let ob = Array.get objects i in
    let den = getden(ob) in
    let st = getst(ob) in
    let index = ref(0) in
    while !index < Array.length den do
      (match Array.get den !index with
          | Dobject j -> pushmarkstack(j)
          | _ -> ());
      index := !index + 1
    done;
    index := 0;
    while !index < Array.length st do
      (match Array.get st !index with
          | Mobject j -> pushmarkstack(j)
          | _ -> ());
      index := !index + 1
    done)
```

- identifico (e inserisco in `markstack`) i puntatori contenuti tra i `dval` e gli `mval` di ambiente e memoria locali dell'oggetto
- se l'oggetto era già marcato (ciclo!) non faccio niente

I punti di partenza del garbage collector

- quali strutture dello stato possono contenere puntatori alla heap?
  - solo quelle rosse

```
cstack: labeledconstruct stack stack
tempvalstack: eval stack stack
tempdvalstack: dval stack stack
labelstack: labeledconstruct stack
namestack: ide array stack
dvalstack: dval array stack
slinkstack: dval env stack
storestack: mval array stack
```

- devo cercare tutti i puntatori lì contenuti (con i prefissi-tipo `Object`, `Dobject` e `Mobject`)

I punti di partenza 1

```
let startingpoints() =
   let index1 = ref(0) in
   let index2 = ref(0) in
   (* dvalstack *)
   while !index1 <= lungh(dvalstack) do
     let adval = access(dvalstack, !index1) in
     index2 := 0;
     while !index2 < Array.length adval do
       (match Array.get adval !index2 with
            | Dobject j -> pushmarkstack(j)
            | _ -> ());
      index2 := !index2 + 1
     done;
     index1 := !index1 + 1
   done;
   index1 := 0;
   (* storestack *)
   while !index1 <= lungh(storestack) do
     let adval = access(storestack, !index1) in
     index2 := 0;
     while !index2 < Array.length adval do
       (match Array.get adval !index2 with
            | Mobject j -> pushmarkstack(j)
            | _ -> ());
      index2 := !index2 + 1
     done;
     index1 := !index1 + 1
   done;
```

I punti di partenza 2

```
let startingpoints() =
   let index1 = ref(0) in
   let index2 = ref(0) in
   .....
   (* tempvalstack *)
   index1 := 0;
   while !index1 <= lungh(tempvalstack) do
     let tempstack = access(tempvalstack, !index1) in
     index2 := 0;
     while !index2 <= lungh(tempstack) do
         (match access(tempstack,!index2) with
            | Object j -> pushmarkstack(j)
            | _ -> ());
      index2 := !index2 + 1
     done;
     index1 := !index1 + 1
   done;
   (* tempdvalstack *)
   index1 := 0;
   while !index1 <= lungh(tempdvalstack) do
     let tempstack = access(tempdvalstack, !index1) in
     index2 := 0;
      while !index2 <= lungh(tempstack) do
         (match access(tempstack,!index2) with
            | Dobject j -> pushmarkstack(j)
            | _ -> ());
      index2 := !index2 + 1
      done;
      index1 := !index1 + 1
    done
```

Allocazione di oggetti con (eventuale) recupero

```
let mark() =
   startingpoints();
   while not(empty(markstack)) do
     let current = top(markstack) in
     pop(markstack);
     markobject(current)
   done
```

```
let newpoint() = if not(!next = -1) then !next
   else
     (mark(); collect();
     if !next = -1 then failwith("the heap size is not sufficient")
     else !next)
```

Condizioni per poter realizzare un garbage collector

- per ogni struttura dello stato (pila dei records di attivazione) devo sapere dove possono esserci puntatori alla heap
  - per poter realizzare `startingpoints`
- per ogni struttura nella heap devo sapere dove possono esserci puntatori ad altri elementi della heap
  - per poter realizzare `markobject`

Digressione su altri costrutti ed altre tecniche

- la realizzazione di una gestione automatica della heap via garbage collection è stata prima di Java limitata ai linguaggi funzionali e logici
  - semplice struttura della pila dei records di attivazione
  - uniformità delle strutture allocate sulla heap (s-espressioni, termini)
- linguaggi come PASCAL, C, C++ hanno scelto di affidare al programmatore la restituzione di strutture e oggetti alla memoria libera, fornendo costrutti del tipo `free` o `dispose`
  - le strutture non sono più davvero permanenti
  - il programmatore dovrebbe eliminare una struttura solo quando essa è logicamente non più attiva (priva di cammini d'accesso)
    - è difficile tenere traccia della dinamica dei cammini d'accesso creati con i puntatori
    - un primo rischio è quello di creare *garbage*
      - se il programmatore si dimentica di restituire una struttura quando muore l'ultimo camino d'accesso
      - non essendoci altri cammini d'accesso il garbage non potrà più essere restituito
    - un secondo rischio (ben più grave!) è quello di creare *dangling references*

Dangling references

- quando il programmatore restituisce alla lista libera una struttura che ha ancora dei cammini d'accesso
  - l'esecuzione può finire in uno stato di errore, perché si cerca di seguire un "puntatore a nulla"
  - la cella della heap restituita potrebbe essere stata riutilizzata per allocare altre strutture, che verrebbero manipolate in modo scorretto
  - in quei casi in cui una parte del contenuto della struttura (vedi liste ed s-espressioni in LISP) è utilizzata per rappresentare la lista libera, un accesso ad un dangling reference potrebbe portare a distruggere senza rimedio gran parte della lista libera
- sono tutti errori molto difficili da localizzare anche perché non necessariamente ripetibili
  - l'effetto dipende dalle dimensioni della heap e persino da possibili esecuzioni pregresse
- ecco perché Java è tanto migliore di C++!

Altre gestioni da parte del sistema

- altri algoritmi di garbage collector
  - l'algoritmo di marcatura naif che abbiamo visto può essere migliorato in molti modi e vengono utilizzati molti altri algoritmi
  - segnaliamo soltanto due problemi "storici" dell'algoritmo naif
    - la marcatura richiede una pila tanto più grande quanto maggiore è il numero di strutture marcate
      - e quanto meno utile è il garbage collector
      - affrontato con l'algoritmo classico di Schorr & Waite che utilizza la struttura a grafo stessa "per ricordare quello che resta da visitare"
    - la marcatura parte quando si esaurisce la lista libera e si cerca di allocare una nuova struttura
      - a quel punto, la computazione si sospende per dare spazio alla marcatura
      - la cosa è quasi sempre visibile e può essere fastidiosa soprattutto in una applicazione interattiva
      - affrontato con i garbage collectors incrementali
- quest'ultimo problema
  - non concentrare in una unica fase temporale il costo della gestione automatica

si può risolvere con una tecnica alternativa al garbage collector

- i contatori di riferimento

I contatori di riferimenti

- ogni struttura allocata nella heap ha associato un contatore
  - che conta il numero di cammini d'accesso alla struttura
- tutte le operazioni che manipolano puntatori vengono appesantite
  - perché devono gestire (incrementare, decrementare) i contatori
- una struttura viene restituita alla lista libera quando il suo contatore diventa 0
  - il costo della gestione è distribuito nel tempo
  - maggiore occupazione di memoria (un intero per ogni struttura)
  - non funziona con strutture dati circolari

Heap disomogenea

- se gli elementi da allocare nella heap sono disomogenei
  - la lista libera è una lista di "blocchi" non tutti della stessa dimensione
    - all'inizio è addirittura formata da un unico blocco che contiene tutto
  - il blocco liberato da un elemento restituito non sempre va bene per allocare un nuovo elemento
- esiste un problema di politiche di allocazione e di organizzazione dell'informazione nella lista libera
  - politica "first-fit"
    - si prende il primo blocco libero sufficiente a contenere il nuovo elemento
  - politica "best-fit"
    - si prende il miglior blocco, cioè quello che produce il minore "sfrido"
    - facile da implementare, se i blocchi nella lista libera sono ordinati per dimensione
- in ogni caso, si può presentare il problema della *frammentazione*
  - la memoria ancora disponibile è divisa in blocchi così piccoli da essere inutilizzabili
  - si può tentare di risolvere il problema con il *compattamento*

Il compattamento

- ha come obiettivo la generazioni di blocchi più grandi
  - possibilmente uno solo!

  a partire da una heap frammentata
  - il problema esiste solo se la heap è disomogenea
- nella versione banale (compattamento parziale)
  - si fondono blocchi adiacenti
  - facile da fare, se la lista libera è ordinata per indirizzi
- nella versione complessa (compattamento totale)
  - si spostano ad una estremità tutte le strutture attive e si ricava un unico blocco di tutto quello che resta
  - più complicato ancora della marcatura, perché vanno modificati tutti i puntatori