CAPITOLO 2

HEAP MANAGEMENT & GARBAGE COLLECTION

INTRODUZIONE

L'allocazione dinamica degli oggetti e il loro recupero automatico quando non piu' accessibili (**garbage collection**) e' quanto avviene dietro le scene di operazioni **new** (*malloc*) e **dispose** (*delete*) nella gestione di puntatori e oggetti nei linguaggi di programmazione. I problemi sono di due tipi:

I. allocazione/ deallocazione di oggetti tutti della stessa dimensione (*LISP*)
II. allocazione/ deallocazione di oggetti di dimensione variabile (*Pascal, C, C++, Java, C# ecc*)

Noi ci occupiamo del caso **II** in uso nei linguaggi piu' comuni.
La deallocazione segue due filosofie contrapposte:

usr) sotto il completo controllo e responsabIlita' dell'utente (*Pascal, C, C++*) mediante primitive tipo "*dispose*", "*delete*", "*free*", ecc.

aut) automatica o "*garbage collected*": completamente invisibile all'utente (Java, C#, Oberon ecc.).
In questo caso il recupero avviene in due fasi:

m) Mark: identificazione dello spazio recuperabile
s) Sweep: inserimento dello spazio recuperato nelle zone disponibili all'utente.

Fase di marcatura "Mark"

m1) Si possono usare contatori detti *reference counters*, che indicano il numero di volte in cui ciascun blocco e'referenziato. Se zero allora il blocco
m2) Si implementa una lista di celle accessibili e seguendo i link di accesso si marcano i blocchi accessibili usando "*type (class) descriptors*".

Fase di recupero "Sweep"

s1) *Recupero*: si organizzano i blocchi liberi in liste di "spazio libero"
s2) *Deframmentazione*: si compattano le celle libere in vari modi e livelli:

1) fondendo insieme i blocchi contigui ricorsivamente;
2) compattando integralmente lo spazio libero mediante rilocazione completa: tutti i blocchi usati vengono rilocati in modo continuo all'inizio (fine) dello Heap e tutti i blocchi liberi vengono fusi in un unico spazio non strutturato detto Tail (array di celle contigue) alla fine (inizio) dello Heap. Ci si riconduce quindi a una situazione simile a quella iniziale.

ALGORITMI DI GESTIONE DELLO SPAZIO

1) QUICK FIT (QF) di Weinstock Wilf et al. [1]

Idea di base

a. I programmi gestiscono un numero relativamente basso di tipi "***POINTER TO T***",

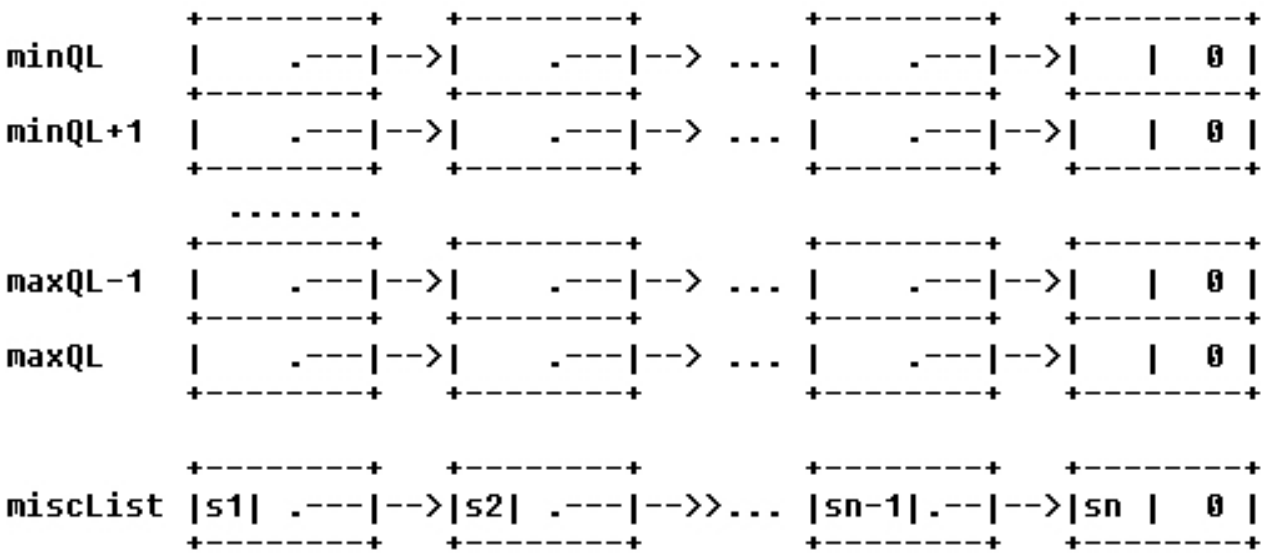
b. I tipi di di dato allocati piu' frequentemente riguardano quantita' di dimensione fissa di memoria relativa a piccoli record di pochi campi (dimensioni relativamente piccole).

Per questo Quick Fit (QF) divide lo HEAP in due parti:

1) memoria gia' allocata (o usata in precedenza) e recuperata detta **Working Storage (WS)**,

2) memoria mai allocata prima detta **TAIL**.

Ogni blocco (o cella) della WS appartiene ad una opportuna lista di spazio libero. QF mantiene piu' liste di celle, ogni lista e' omogenea nella dimensione delle celle ad eccezione dell'ultima. Si ha una struttura dati formata da un insieme di liste "*quicklist*" ciascuna composta dalle celle libere di dimensione minQL, minQL+1, ..., maxQL-1, maxQL.
Queste dimensioni sono scelte per coprire le richieste di dimensione piu' frequente. La misclist comprende blocchi di dimensione variabile e diversa da quelle delle quicklist. All'inizio le quicklist e la misclist sono tutte vuote ed esiste un unico grande TAIL raggruppante tutto lo spazio libero ottenuto dal sistema (OS). Ad un punto intermedio dell'esecuzione la situazione e' la seguente:



Algoritmo di base

Piu' che un algoritmo QF rappresenta una famiglia di algoritmi che funzionano come segue:

1) Se la richiesta riguarda un blocco di dimensione tra minQL e maxQL e la QL rispettiva non e' vuota rimuovere il primo blocco della lista e restituirne l' indirizzo al richiedente.

2) Altrimenti se la richiesta e' soddisfatta dalla struttura TAIL, ritagliare il blocco da TAIL mediante un semplice incremento di puntatori, e restituirne l' indirizzo al richiedente.

3) Se 1 e 2 falliscono si usa lo schema seguente:

- **a.** usare la misclist in una scansione *first-fit* fino a trovare una cella o blocco di dimensione sufficiente
- **b.** implementare la misclist tramite un *albero bilanciato AVL* per Ridurre il tempo di ricerca

Altre migliorie consistono in:

- **c.** spezzare i blocchi grandi in due parti allocando lo spazio richiesto e accodando il resto nella lista opportuna
- **d.** periodicamente compattare tutti blocchi contigui in uno solo.

La deallocazione e' semplice:

1) Se la dimensione del blocco da deallocare e' prevista da una quicklist inserirlo nella lista opportuna (all'inizio).
2) Altrimenti inserirlo all'inizio della misclist.

Per l'analisi delle prestazioni e confronti con altri metodi si rimanda a **[2,3]**. L'analisi puo' essere fatta sia comparativamente con altri metodi,sia in termini assoluti, valutando il tempi di allocazione e deallocazione medi.
Una sperimentazione fatta dimostra che l' 80% delle allocazioni proviene dalle quicklist e il 96% dalle quicklist o dal TAIL. E' stato anche provato che una variante dei Buddy System e' leggermente piu' veloce e che una strategia di best fit utilizza la memoria in modo piu' efficiente.
Quick fit resta comunque estremamente competitivo ed uno dei migliori sia in tempo di esecuzione sia in utilizzo della memoria.

Nota:

Secondo D. Ungar e F. Jackson (OOPSLA'88) il metodo non sembra pratico per architetture moderne. Benche' il capitolo seguente mostri il contrario almeno con linguaggi type safe.

BIBLIOGRAFIA

[1] W. A. Wulf, R. K. Johnson, C. B. Weinstock, S. O. Hobs, C. M. Geschke, "THE DESIGN OF AN OPTIMIZING COMPILER", American Elsevier, 1975.

[2] Knuth, "SORTING AND SEARCHING", The Art of Computer Programming, Vol. III, Addison-Wesley, 1973.

[3] C. B. Weinstock, "DYNAMIC STORAGE ALLOCATION", Ph.D Thesis, Carnegie Mellon University, 1976.

APPENDICE

Dettagli e specifiche

Supponendo che il size richiesto sia nel reg r0 e che l'indirizzo del blocco venga restituito in r1, il codice generato e' il seguente (Ada VAX):

```
        mov   QL[r0],r1       ; root della quicklist
        beq   L1              ; salta se vuota
        movl  (r1),QL[r0]     ; restituisci il 1o blocco
        br    done
L1:     cmpl  r0,tailrest     ; c'e' spazio?
        bgtr  L2              ; no, usa schema complesso
        movl  tailbagin,r1    ; inizio tail
        addl3 r0,r1,tailbegin ; incrementa
        subl2 r0,tailrest     ; decr spazio restante
        br    done
L2:     pushl r0              ; push requested size
        calls $1,hardalloc    ; call compex strategy
```

Codice in Ada dell'algoritmo originale completo:

```
--
-- package specification
--

     with SYSTEM
     generic
       minQL: integer:=o;        -- min quick list item size
       maxQL: integer :=32;      -- max quick list item size
       OSchunk: integer :=4096;  -- how much we get from the OS
     package QuickFit is
       function Alloc(n: natural) return ststem.address;
       procedure DeAlloc(s: system.address;n: natural);
     end QuickFit;
--
-- package body of QuickFit
--

with UNCHECKED_CONVERSION;
package body QuickFit is

--------------------  Types For Storage on a Free List
type free_cell;                    -- a free block(see below)
type frePtr is access free_cell;  -- pointer to a free block
type free_cell  is
record
       next: freePtr;     -- pointer to the next free block on
                          -- a list
       size: natural;     -- size of the current free block
end record;

--------------------   Constants Related to Allocation Size

Chunk:  constant integer := natural'size/system.storage_unit;
MinChunk:  constant integer := _cell'size/system.storage_unit;

--------------------   The Free Space Pointer

QL: array(minQL..maxQL) of freePtr;  -- One list for each size
MiscList: freePtr;          -- A single list for all sizes
TailBegin: system.address;  -- Beginning of space left in the tail
TailRemaining: natural :=0; -- Amount of space left in the tail

--------------------   Unchecked Conversion Between Types
function ConvertToFreeptr is new
UNCHECKED_CONVERSION(natural,freePtr);
function ConvertToFreeptr is new
UNCHECKED_CONVERSION(system.address,freePtr);
function            ConvertToAddress            is            new
```

```
UNCHECKED_CONVERSION(freePtr,system.address);
function ConvertToAddress is new
UNCHECKED_CONVERSION(natural,system.address);
function ConvertToNatural is new
UNCHECKED_CONVERSION(system.address,natural);

--------------------  Support Routines

function "+"(s: system.address; i: natural) return system.address
is
--   addition between addresses and naturals
begin
 return ConvertToAddress(ConvertToNatural(s)+i);
end "+";

function  ComputeSize(size natural) return natural is

--  Round a size request up to
-- (a) an integral numb. of chunks" and,
-- (b) at least MinChunk (so that there is always
--  space for the free space information)

begin
  if size < minChunk
  then return minChunk;
  else return (size+(Chunk-1))/Chunk;
  end if;
end ComputeSize;

--------------------  Allocaion Strategy Routines

function  QuickAlloc(size: natural) return system.address is
--  perform an allocation from one of the quick lists
cell: freePtr;
begin
 cell := QL(size);
 QL(size):= cell.next;
 return ConvertToAddress(cell);
end QuickAlloc;

function TailAlloc(size: natural) return system.address is
-- perform an allocation from the tail
locn: system.address;
begin
 locn := TailBegin;
 TailBegin := TailBegin + size;
 Tailremaining := Tailremaining - size;
 return locn;
end TailAlloc;
```

```
function MiscAlloc(size: natural) return system.address is
-- perform an allocation from the micellaneous list
begin
 -- Uses whatever method pleases you.
 -- Return the null address if allocation is not possible.
 -- The next line is only for syntactic correction
 return ConverToAddress(null);
end MiscAlloc;

function ExtendTail is
-- obtain new storage from the operating system
begin
 if TailRemaining > 0 then then DeAlloc(TailBegin,TailRemaining)
 end if;
 --
 -- Tailbegin := << space obtained from the operating system.>>
 -- <<raise storage_error if the OS fails>>
 -- TailRemaining := OSChunk;
 --
end ExtendTail;

function HardAlloc(size: natural) return system.address is
begin
 if size > OSchunk then raise storage_error;
 end if;
 locn := MiscAlloc(size);
 if licn /= ConvertToAddress(null) then return locn;
 end if;
 extendTail;
 return TailAlloc(size);
end hardAlloc;

--------------------------- Entry points to the QuickFit Allocator

function Alloc(n: natural) return system.address is
-- allocate a chunk of size (at least) n storage units
size: natural;
begin
 size := ComputeSize(n);
 if size in minQL..maxQL then
    if QL(size) /= null then return QuicjAlloc(size);
    elsif size <= TailRemaining then return TailAlloc(size);
    else return HardAlloc(size);
    end if;
 elsif size <= TailRemaining then return TailAlloc(size);
 else return HardAlloc(size);
 end if;
end Alloc;
```

```
procedure DeAlloc(s: system.address; n: natural) is
-- deallocate the chunk at s of length n
size: natural;
cell: freePtr := ConvertToFreePtr(s);
begin
   size := ComputeSize(n);
   cell.size := size;
   if not(size in minQL..maxQL)
   then  cell.next := MiscList;
         Misc.List := cell;
   else  cell.next := QL(size);
         QL(size) := cell;
   end if;

end DeAlloc;

end QuickFit;
```

UN CASO COMPLETO : LO HEAP DI OBERON

Oberon e' un linguaggio *type-safe* derivato da Modula-2 che supporta uno stile di programmazione object-oriented. Oberon e' anche il nome dell'intero sistema (sistema operativo e window system) in cui opera il linguaggio. Il kernel di Oberon implementa anche un allocatore di memoria e un garbage collector. Nel sistema Oberon **[1]** B. Heeb e C. Pfister hanno implementato una variante di QuickFit. La memoria viene allocata in blocchi di dimensione multipla di un valore minimo costante B, la dimensione di ogni blocco richiesto viene arrotondata al minimo multiplo di B superiore o uguale al size richiesto. Qui le quicklist vengono chiamate freelist e sono "ancorate"ad una struttura di array globale A.

```
A: ARRAY[1..N] OF ADDRESS (* freelist *)
```

dove:

```
size(A[i]=i*B) per i=1,..N-1;
```

e A[N] e' ancora la misclist dei blocchi di dimensione variabile.
L'allocazione di un blocco di dimensione s consiste nel rimuovere il primo elemento della lista A[k] dove:

```
(s<=k*B)and(forall i<k: ((s> B*i) or (A[i]=NIL)))
```

Nel caso in cui k=s/B si restituisce l'indirizzo del blocco trovato, ltrimenti k>s/B e il blocco viene diviso in due parti di dimensioni rispettive s e r=k*B-s. Il primo viene allocato restituendone l' indirizzo e il size, il secondo viene inserito in A[r/B]. Nell'implementazione originale di Oberon sulla station Ceres, i blocchi erano di dimensione ristretta a una potenza di due, metodo che tende ad aumentare la frammentazione della memoria.

L'array A[i] i<N viene usato per blocchi di piccola dimensione (non piu' di 128 byte ciascuno) mentre A[N] contiene blocchi di dimensione variabile secondo la formula (N+j)*B, j=1,2, ... usando una strategia di tipo best-fit per determinare il blocco piu' vicino alla richiesta. La routine di allocazione e' riportata di seguito in pseudo-codice:

```
PROCEDURE Allocate(VAR a: ADDRESS; size LONGINT);
   VAR i: INTEGER;l,r:ADDRESS;
       BEGIN
        i:= MIN(size/B,N))  (* indice lista  forzato <=N *)
       WHILE (i<N)&(A[i]=NIL) DO i:=i+1 OD; (*find min non empty
list*)
       l:=ADR(A[i]); a:=l^;(*addr. and value of pnt to 1th free
block*)
          WHILE (a<>NIL)&(a.size<size)
          DO (*if i=N then first fit*)
                 l:=ADR(a.next); a:=l^
          OD;(*end while*)
          IF a#NIL THEN
              l^:=a.next; (*  remove block from freelist*);
             IF a.size>size THEN (* block must be split *)
                i:=MIN((a.size-size)/B,N);
                r:=a+size;
               r.size:=a.size-size; (* adj. size of residual blk*)
               r.next:=A[i];(* insert residual blk in free list *)
              A[i]:=r       (* insert residual blk in free list *)
              FI
          FI
 END Allocate;
```

GARBAGE COLLECTION

Si svolge nelle due fasi classiche Mark-and-Sweep. In Oberon molte variabili sono locali e quindi allocate sullo stack e questo fatto riduce lo spreco di memoria prodotto rispetto a linguaggi tipo Lisp e Smalltalk. In questi ultimi linguaggi l'algoritmo mark-and-sweep non sembra particolarmente adatto[6], mentre per Oberon e lingaggi simili si e' mostrato piu' che adeguato. Nella prima fasi si marcano tutti gli oggetti referenziati, nella seconda si attraversano tutti i blocchi dello heap recuperando quelli non marcati. La fase Mark e', in un primo raffinamento, la seguente:

```
  PROCEDURE Mark(q: ADDRESS);
           VAR off:ADDRESS;
           BEGIN
                IF (q<>NIL)&Unmarked(q) THEN
                   SetMark(q);off:=FirstPointerOffset(q);
                   WHILE off>=0
                   DO
                       Mark(MEM[q+off]);
                       off:= NextPointerOffset(q,off)
                   OD;
                 FI
   END Mark;
```

Si usano due procedure ausiliarie:

- ***FirstPointerOffset*** che individua il campo di record contenente il primo puntatore;
- ***NextPointerOffset*** iterativamente fornisce il campo successivo di tipo puntatore. Un offset negativo e' usato come "sentinella" per terminare la ricerca.

Procedure ausiliarie

PROCEDURA	AZIONE SVOLTA
FirstPointerOffset	individua il record field contenente il primo puntatore
NextPointerOffset	itera individuando i record fielddi tipo puntatore successivi, un valore<0 funge da sentinella (terminatore)
Unmarked	funzione di controllo marcatura
SetMark	procedura di marcatura

L'implementazione di FirstPointerOffset e di NextPointerOffset richiede un metodo efficiente per determinare gli offset di ogni pointer all'interno di un record. Questo offset e' lo stesso per tutte le variabili dello stesso tipo. E' quindi ragionevole fornire un "type descriptor" che implementa una tavola con tutti gli offset dei puntatori interni ad un record.

Il "**Type descriptor**" e' una struttura dati costruita dal compilatore e passata al RTS (si ha un unico descrittore per tipo record o classe) cosi' formato:

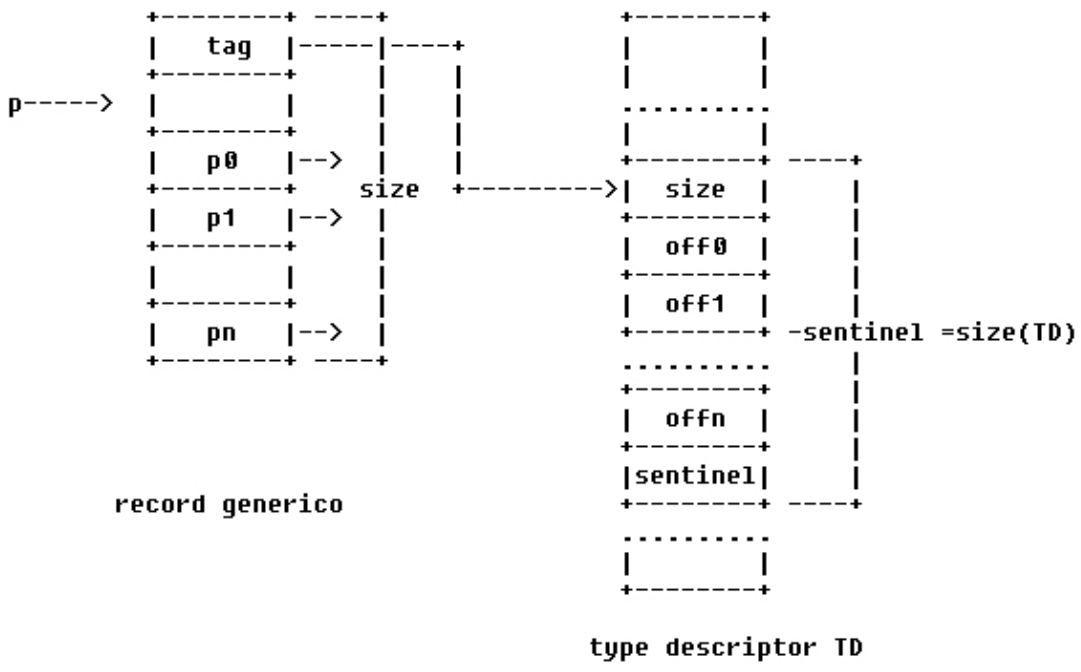
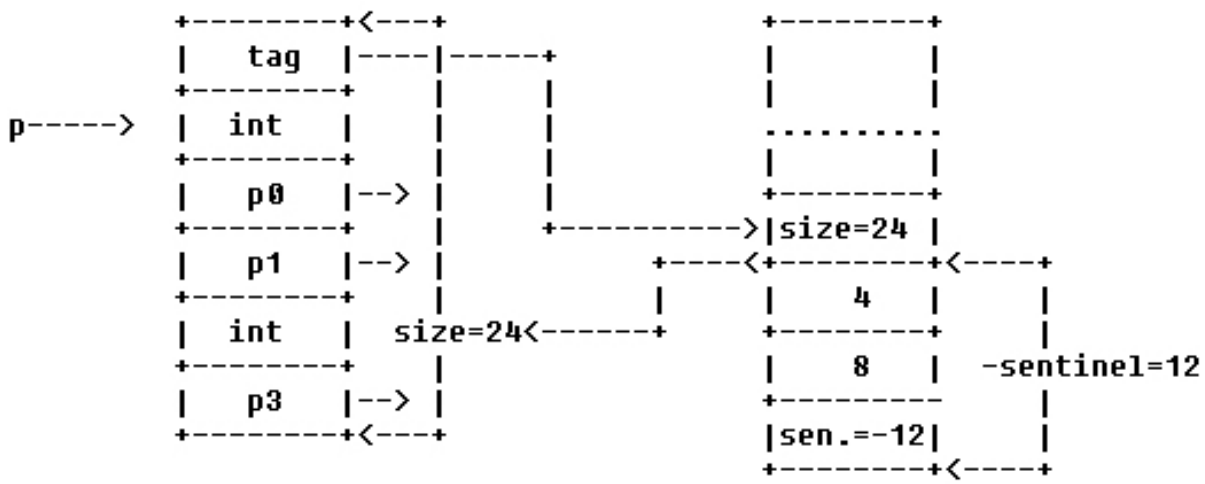


Fig. 1 Type descriptor per fase Mark (visita e marking)

Il descrittore contiene i campi size e ptable: ptable e' un una tavola di pointer-offset che descrive dove una variabile di tipo T ha un puntatore relativamente alla propria base.

Un esempio pratico di type descriptor e' il seguente:

```
TYPE
     A =    RECORD
                s:INTEGER;
                p0,p1: POINTER TO B;
                t: REAL;
                p3:  POINTER TOC;
            END;
```



La versione seguente usa low level features sostituendo le procedure FirstPointerOffset NextPointerOffset con incrementi del type tag con il size di un pointer:

```
PROCEDURE Mark(q: ADDRESS);
 VAR off:ADDRESS;
 BEGIN
      IF (q#NIL)&Unmarked(q) THEN
         SetMark(q);INC(Tag(q));
         WHILE MEM[Tag(q)]>=0
         DO
             Mark(MEM[q+MEM[Tag(q)]]);
             INC(Tag(q));
         OD; (*end while*)
         RestoreTag(q)
       FI
END Mark;
```

dove il type tag di un record cambia durante la fase di marking in modo da puntare sempre al campo offset da processare e dopo quello appena elaborato **(Fig. 2)**.

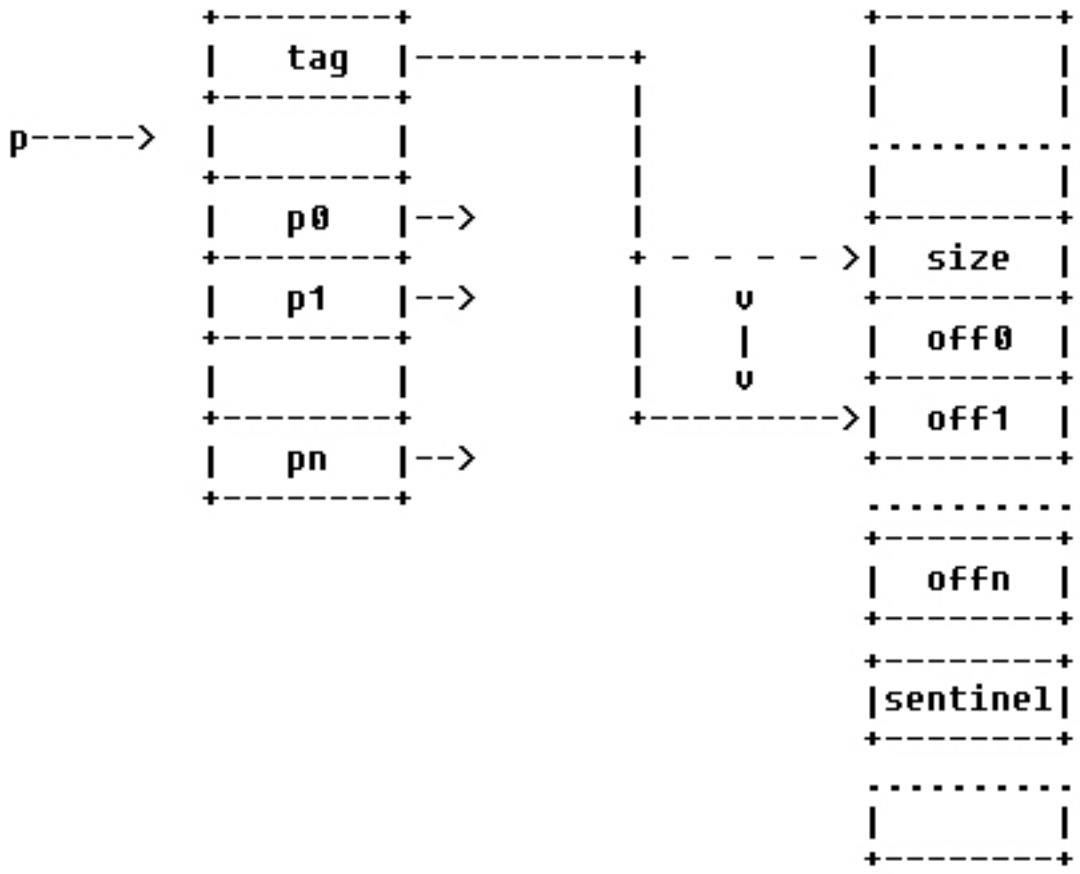


Fig. 2 Type Tag durante la fase Mark

Il loop sugli offset termina quando si incontra un offset negativo.
Questo valore e' inzializzabile in modo che la RestoreTag si riduca a

```
Tag(q):= Tag(q)+mem[Tag(q)]
```

La routine Mark si trasforma, muovendo la guardia (q#NIL)&Unmarked(q) fuori dalla procedura mark nella versione seguente:

```
PROCEDURE Mark(q: ADDRESS);
  VAR r: ADDRESS;
  BEGIN  (* (q#NIL)&JustMarked(q) *)
       DO (* endless loop*)
          Increment(Tag(q));
          IF mem[Tag(q)] >=0 THEN
             r:=mem[q+mem[Tag(q)]];
             IF (r#NIL)&Unmarked(r) THEN
                SetMark(r);
                Mark(r)
             FI
          ELSE RestoreTag(q);
             RETURN (* exit from endless loop*)
          FI
       OD (* closes endless loop*)
END Mark;
```

JustMarked significa che l'oggetto e' marcato, ma nessuno dei discendenti lo è. La ricorsione e' ora eliminabile con un stack esplicito. La chiamata ricorsiva e' sostituita da Push(q);q:=r; e Return da Pop(q).

```
PROCEDURE Mark(q: ADDRESS);
VAR r: ADDRESS;
BEGIN  (* (q#NIL)&JustMarked(q) *)
   Stack:=empty;
   DO
     Increment(Tag(q));
     IF mem[Tag(q)] >=0 THEN
       r:=mem[q+mem[Tag(q)]];
       IF (r#NIL)&Unmarked(r) THEN
         SetMark(r);
         Push(q);
         q:=r
       FI
     ELSE RestoreTag(q);
     IF Stack=empty THEN
       EXIT
     ELSE
       Pop(q)
     FI
   OD
END Mark;
```

Il difetto di questa procedura e' l'uso di uno stack che richiede ulteriore memoria per lessere eseguito. L'algoritmo di Deutsch Schorr Waite [7], lo stack viene distribuito nelle locazioni dei puntatori che via via vengono attraversati. Qui usiamo una variabile p come stack pointer, cioe' come puntatore all'oggetto contenente il predecessore, a sua volta contenuto in uno dei campi puntatore: esattamente in quello attualmente scandito. Il vecchio puntatore e' memorizzato sia nella variabile ausiliaria r sia sullo stack. Questo introduce le seguenti ulteriori modifiche:

```
Stack=Empty  → p=NIL
Push(q)  → mem[q+mem[Tag(q)]] := p; p := q;
Pop(q)   → a := p+mem[Tag(p)]; r:= mem[a];mem[a]:=q;q:=p;p:=r;
```

```
PROCEDURE Mark(q: ADDRESS);
  VAR p,r,a: ADDRESS;
  BEGIN  (* (q#NIL)&JustMarked(q) *)
       p :=NIL;
       DO  (*endless loop *)
          Increment(Tag(q));
          IF mem[Tag(q)] >=0 THEN
             r:=mem[q+mem[Tag(q)]];
             IF (r#NIL)&Unmarked(r) THEN
                SetMark(r);
                mem[q+mem[Tag(q)]] := p;
                p :=q; q := r;
             FI
          ELSE RestoreTag(q);
             IF p=NIL THEN
                EXIT
             ELSE
                a := p + mem[Tag(p)];
                r := mem[a];
                mem[a] := q;
                q :=p; p:=r
             FI
          FI
       OD (*closing endless loop *)
END Mark;
```

Concludiamo riportando la versione scritta nell'assembler del Motorola MC68000. L'implementazione considera anche dati addizionali, non rilevanti per il problema trattato relativi al RTTI che devono essere memorizzati nel type descriptor tra size e ptable.
I type descriptor vanno trattati esattamente come ogni altro record: quindi devono avere un type descriptor che differiscono solo per il campo size.
Essi condividono un meta meta type descriptor il cui tag punta a se' stesso cioe' al proprio type descriptor. E' comunque piu' conveniente trattare questi casi in modo speciale (ad esempio inserirli nell'area dati globali).

Il seguente codice assembler MC68000 implementa la fase mark dell'algoritmo.

```
    * 68000 mark phase for garbage collector
    * A0: pointer to father
    * A1: pointer to node
    * A2: temporary, for pointer rotation
    * A3: tag or pointer to current pointer offset
    * pointer offsets are usually accessed via A3 with offset
      of Offset(ptable)-4-1. The pointer is incremented before
      it is accessed, thus the subtraction of PtrSize. The
      subtraction of 1 comes from the set mark bit (bit # 0).
    * D0: offset
    * D1: temporary
    PtrSize   EQU         4          size of pointers and offsets
    Tag       EQU         -4         offset of type tag
    Tail      EQU         Tag+3      low byte of type tag
    Mark      EQU         0          mark bit (in TagL)
    PTab      EQU         36         ptable offset
    Offset    EQU         PTab-PtrSize-1

    Start     MOVE.L      A1,D1          NIL test
              BEQ         End            NIL
              BSET.B      #Mark,TagL(A1) test and set mark bit
              BNE         End            marked
              MOVE.L      #0,A0          father:=NIL
              MOVE.L      Tag(A1),A3     lad first tag
              BRA         Loop

    Up        ADD.L       D0,A3          adjust tag
              MOVE.L      A3,Tag(A1)     save tag
              MOVE.L      A0,D1          NIL test
              BEQ         End            father=NIL (sentinel)
              MOVE.L      Tag(A0),A3     load father.tag
              MOVE.L      Qffset(A3),D0  load offset
              MOVE.L      (A0,D0),A2     rotate pointers, step1
              MOVE.L      A1,(A0,D0)     rotate pointers, step2
              MOVE.L      A0,A1          rotate pointers, step3
              MOVE.L      A2,A0          rotate pointers, step4

    Loop      ADDQ.L      #PtrSize,A3    address of next offset
              MOVE.L      OffSet(A3),D0  load next offset
              BMI         Up         negative sentinel reached,EoL
              MOVE.L    (A1,D0),A2   load son(& rotate pts, step1)
              MOVE.L      A2,D1          NIL test
              BEQ         Loop           NIL
              BSET.B      #Mark,TagL(A2) test and set mark bit
              BNE         Loop           marked
```

```
Down    MOVE.L      A3,Tag(A1)      save tag
        MOVE.L      A0,(A1,D0)      rotate pointers, step1
        MOVE.L      A1,A0           rotate pointers, step2
        MOVE.L      A2,A1           rotate pointers, step3
        MOVE.L      Tag(A1),A3      load new tag
        BRA         Loop

End
```

Sweep Phase

In questa fase si traversa lo heap sequenzialmente blocco per blocco. Per fare questo e' neccessario conoscere la dimensione di ogni blocco. La somma tra il size di un blocco e il suo indirizzo fornisce l'indirizzo del blocco successivo. Dat type tag di ogni blocco si ottiene un riferimento al type descriptor che contiene tra l'altro la dimensione. I blocchi liberi si trattano similmente, con la differenza che ciascun blocco libero e' il proprio type descriptor. La figura seguente illustra la situazione.

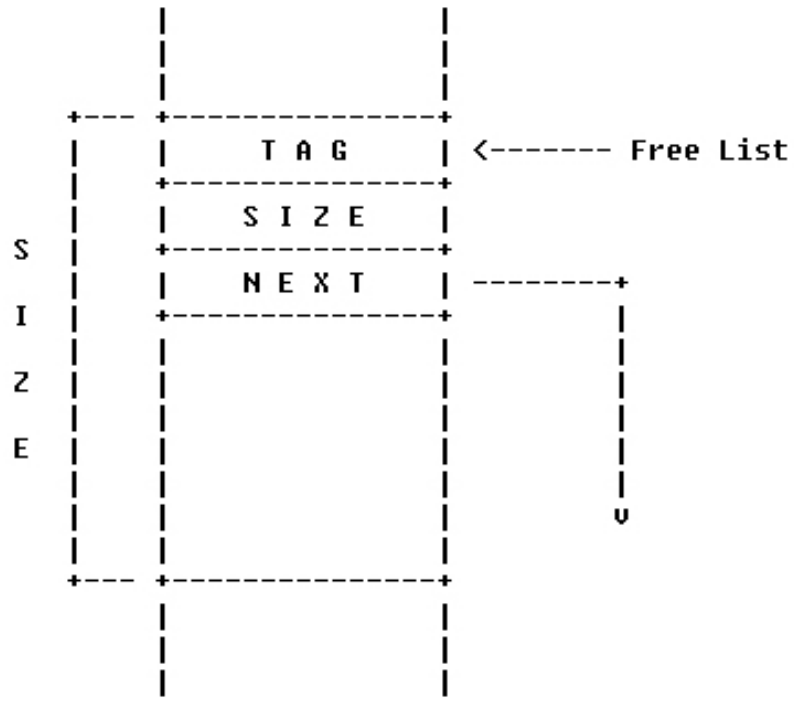


Fig. AA Struttura di un blocco libero

Tutte le free list vengono azzerate all'inizio di questa fase. L'algoritmo sweep le ricostruisce nuovamente condensando blocchi consecutivi smarcati in un unico blocco unione dei blocchi componenti contigui e inserendolo nella free list opportuna. Tutte le marcature vengono azzerate.

```
PROCEDURE Scan;
        VAR p: ADDRESS;
        BEGIN
          a[1..N]:=NIL;
          q:=HeapStart;
          DO
            WHILE (q#MemSize)&Marked(q)
            DO
              ResetMark(q);
              q:= q+mem[mem[q]]
            OD;
            IF q # MemSize
            THEN
             p:=q;
             DO
               q := q + mem[mem[q]]
             UNTIL (q=MemSize) OR Marked(q);
             Insert(p,q-p)
            FI
          UNTIL q = MemSize
 END Scan;
```

La procedura Insert(p,s) inserisce un blocco libero all'indirizzo p nella free list di size s. L'invariante in questo merge-sweep e' che tutti i blocchi liberi gia' visitati hanno size massimo, cioe' sono blocchi contigui immersi. Solo il blocco di visita piu' recente potrebbe richiedere un merging con quello successivo. Questo invariante rappresenta la differenza principale tra la deallocazione merge-sweep e la deallocazione di blocchi arbitrari.

BIBLIOGRAFIA

[1] B. Heab, C. Pfister, "AN INTEGRATED HEAP ALLOCATOR/GARBAGE COLLECTOR",Oberon Technical Notes C. Pfister Ed., ETH Technical Report, 30-39 .

[2] N. With, "THE PROGRAMMING LANGUAGE OBERON", Software practice and Experience, 18(7), 661-670.

[3] N. Wirth J. Gutknecht, "THE OBERON SYSTEM", 1989.

[4] H. Eberle, "DEVELOPMENT AND ANALYSIS OF A WORKSTATION COMPUTER", Ph.D Thesis, no. 8431, 1987 ETH Zurich.

[5] D. Knuth, "The Art of Computer Programming", Addison-Wesley, 1973.

[6] J. McCarthy,"RECURSIVE FUNCTIONS OF SYMBOLIC EXPRESSIONS AND THEIR COMPUTATION BY MACHINE", CACM, 3, 1960, 184-195.

[7] D. Ungar, F. Jackson, "TENURING POLICIES FOR GENERATION-BASED STORAGE RECLAMATION", OOPSLA'88 Proceedings, 1988, 107-118.

[8] H. Schorr, W. Waite, "AN EFFICIENT MACHINE-INDEPENDENT PROCEDURE FOR GARBAGE COLLECTION IN VARIOUS STRUCTURES", CACM, 10 (8), 1967, 501-505.

LETTURE ULTERIORI

[a] Tuning Garbage Collection with the 1.3.1 Java Virtual Machine,

http://java.sun.com/docs/hotspot/gc/index.html