Generational Cyclic Reference Counting

Rafael D. Lins
Departamento de Informática - U.F.PE. - Recife - Brazil
Computing Laboratory - The University - Canterbury - England.

Key Words: Compilers, Garbage Collection, Functional Programming

Introduction

Automatic garbage collection is an area of rising importance. It first appeared in LISP systems and in early list based theorem provers and has been incorporated into many areas of computer science. Fully modular programming relies on garbage collection to avoid introducing unnecessary inter-module dependencies. Explicit storage allocation is not only a burden to programmers, but is also a frequent source of subtle errors due to late or early recycling of objects. It also makes program debugging very difficult, as errors tend to occur at different times during execution.

The techniques usually employed for memory management in modern programming languages is one of the variants of the *mark-scan*, *copying* or *reference counting algorithms* (see [1, 10] for a survey of algorithms for garbage collection). A mark-scan garbage collection algorithm works in two phases. When a machine runs out of space, computation is suspended and garbage collection is performed. First, the algorithm traverses all the data structures in use, marking each cell visited. Then the scan process places all unmarked cells onto a *free-list*. The time taken by the mark-scan algorithm is proportional to the size of the heap (the work space where cells are allocated).

The copying algorithm is a modified version of the mark-scan algorithm in which the heap is divided into two halves. This algorithm copies cells from one half to the other during collection traversing all data structures in use. Its time complexity is proportional to the size of the graph in use. Practical observation shows that young cells tend to die young and old cells tend to remain alive until the very end of computation [3]. In order to avoid much of the repeated copying of old objects *generational collection* segregates objects into multiple areas by age [3]. Areas of young objects are copied more frequently than the ones with older objects. The mark-scan algorithm can also be made generational [2].

A completely different technique for memory management is offered by reference counting. In reference counting, each data structure or cell has an additional field, `RC`, which contains the number of references to it. During computation, alterations to a data structure imply changes to the connectivity of the graph and, consequently, re-adjustment of the `RC` field of the cells involved. Reference counting has the major advantage of being performed in small steps interleaved with computation. The disadvantage of the simple algorithm for reference counting is the inability to reclaim cyclic structures. To solve this problem, reference [8] presents a simple reference-counting garbage collection algorithm for cyclic data structures, which works as a natural extension of the standard reference counting algorithm. Deletion of a pointer to a shared structure increases the complexity of the local mark-scan to $O(n)$, where n is the size of the shared subgraph. Unfortunately, the overhead of this algorithm is too high for applications that make extensive use of sharing and of cyclic data structures. Making mark-scan lazy [6] removes the drawback of running mark-scan every time a pointer to a cell with multiple references is deleted, by placing a reference to these cells onto a queue. The deletion of the last pointer to a shared cell will recycle it immediately, regardless of whether there is a reference to it on the queue. This means that more shared cells will now be claimed directly without the need of the mark-scan phase. Only if the free-list is empty or the queue is full is the local mark-scan required. Experimental evidence shows that the lazy algorithm is more efficient than the local mark-scan [7].

Although local, mark-scan can be expensive and should be avoided by every means. If unavoidable it should be as effective as possible. In this paper, we introduce the concept of the age of a cell to cyclic reference counting. Lifetime figures vary from language to language and program to program, but usually between 80 to 98 percent of all newly-allocated objects die within a few million instructions, or before another megabyte has been allocated. The majority of objects die even younger, within tens of kilobytes of allocation [3, 10, 9]. Age information brings the advantage of selecting the youngest cell in the queue, increasing the likelihood of running mark-scan on garbage cells. We also use the age information as a way of detecting the existence of cycles during the mark phase. This information allows the algorithm to perform the scan phase more efficiently.

The Lazy Mark-Scan Algorithm

The algorithm presented in [8] performs a local mark-scan whenever a pointer to a shared structure is deleted. It works in three phases. In the first phase, the graph below the deleted pointer is traversed, counts due to internal references are decremented and nodes are marked as possible garbage. In phase two, the subgraph is rescanned for cells with positive reference count. These are cells to which there are external references. They are re-marked as ordinary cells and their counts are reset. All other nodes are marked as garbage. Finally, in phase three all marked cells are returned to the free list. The algorithm above was optimised in reference [6] allowing mark-scan to take place lazily. The deletion of a pointer to a shared cell pushes a reference to this cell on a queue `Q` and mark-scan is postponed. This delay has the effect of recycling some of the shared cells directly, without performing mark-scan.

We use the notation `<R,S>` to denote a pointer from node `R` to node `S`. Each node `S` has a colour `colour(S)`, which is green, red, blue, or black. The initial colour of each node is green; the other three colours are used only during execution of the algorithm that deletes a pointer. The colour of a pointer `<R,S>` is the colour of node `R`. A cell `T` belongs to set `Sons(S)` iff there is a pointer `<S,T>`.

The following invariant P is maintained by all procedures (assuming it is true initially). That P must be maintained is not mentioned in the descriptions given below; it is implicitly understood.

P: for all nodes `S`*,* `RC(S)` *is the number of green or black pointers to it.*

Procedure `recolor` maintains P as it changes the colour of a node.

```
{ Change the colour of node S to C}
recolor(S,C)=
        for T in Sons(S) do
           if colour(S)=green and C≠green then decrement RC(T);
           if colour(S)≠green and C=green then increment RC(T);
        colour(S):=C
```

The following two procedures are used only when all nodes are green or black. Free cells are linked in a structure called a *free-list*. When needed, a node is obtained from free-list using the following algorithm. Note that field `RC` remains the same for a node moved from the free-list, since the number of pointers to it remains the same. If a new cell is required and the free-list is empty, the cells on `Q` are mark-scanned.

```
{ Cell R is reachable from root.
 Obtain a cell U from free-list and create pointer <R,U>}
New(R) = if free_list not empty then select U from free_list;
                                     make pointer <R,U>
         else if Q not empty then scan_queue; New (R)
              else write_out "No cells available"
```

`scan_queue` is the routine responsible for calling the local mark-scan on the cells on `Q` as explained later.

`Copy` increases the connectivity of the graph.

```
{<S,T> exists. R is reachable from root. Create pointer <R,T>.
  Paint T green}
Copy(R, <S,T>) = increment RC(T); make pointer <R,T>;
                 colour(T) := green;
```

We now present the procedure that deletes a pointer to a node `S`. The complexity arises in that deleting a pointer to `S` may allow `S` to be placed on the free-list if all remaining pointers to it are cyclic in nature. The deletion of the last pointer to a cell automatically recycles it. Removing a pointer to a shared cell `S` forces testing of the colour of `S` to avoid multiple references on queue `Q`. If not black, the cell is painted black and appended to `Q`.

```
Delete(<R,S>) = remove <R,S>
                    { standard reference counting}
                    if RC(S) = 1 then
                              colour(S) := green;
                              for T in Sons(S) do Delete(<S,T>);
                              link S to free_list
                     else decrement RC(S);
                     { lazy reference counting}
                         if colour(S) not black then
                              colour(S) := black;
                              Q := Q ++ [S] { append S  to Q}
```

Now let us explain how `Q` is used. The algorithm pops the cell `S` on the front of `Q` and tests its colour. If black, then a local mark-scan is performed. The subgraph S is coloured red so that `RC(S)` is the number of pointers from outside subgraph S into S (see invariant P). Then, S is scanned in a fashion that makes blue the subgraph of graph S that indeed has no pointers into it and makes green the rest of it. Finally, the blue subgraph, which must be rooted at `S`, is placed on the free-list. Otherwise, the cell was in the path of a previous call to delete and has been recycled already, so `scan-queue` is re-invoked.

```
scan_queue = S := head(Q);
             Q := tail(Q);
             if colour(S) is black then
                 {local mark-scan}
                 mark_red(S); scan(S); collect_blue(S);
             else if Q not empty then scan-queue
```

`mark_red(S)` paints red `S` and all the cells in the subgraph S. It also decrements the reference counts of the cells visited, so the final reference counts are associated only with pointers from outside the subgraph.

```
{ All cells are green or black. Paint the subgraph  S red.}
mark_red(S) =  if colour(S) is green or black then
                                       recolor(S,red);
                                       for T in Sons(S) do
                                                   mark_red(T)
```

`scan(S)` searches the red subgraph S for green pointers into S (a cell will have an external reference if its reference count is greater than zero). If during `scan` an external reference is found auxiliary function `scan_green` paints green the sub-graph below the external reference. Cells with no external references are painted *blue*.

```
{ Graph S is red.
 Paint blue the subgraph of S with no green pointers to it.
 Paint green the subgraph of S with green pointers to it.}
scan(S) = if colour(S) is red then if RC(S)>0  then scan_green(S)
                                   else recolor(S,blue);
                                        for T in Sons(S) do scan(T)
```

`scan_green(S)` paints green the subgraph S and increases the reference count of the cells visited, to take into account the internal pointers within the subgraph (which had been set to zero by `mark_red`).

```
{ Make green the red-blue subgraph below a green pointer}
scan_green(U) = recolor(U,green);
                for T in Sons(U) do
                   if colour(T) is not green then scan_green(T)
```

`collect_blue(S)` recovers all the blue (garbage) cells in the subgraph given by `S` and links them to the *free-list*.

```
{ Place (possibly empty) blue subgraph  S onto free-list.}
collect_blue(S) = if colour(S) is blue then
                          recolor(S,green);
                          for T in Sons(S) do collect_blue(T);
                                              remove <S,T>;
                          link S to free_list
```

The algorithm presented above is lazy in the sense that the mark-scan phase is performed on demand, i.e. only when the free-list is empty or when the queue `Q` is full. Different strategies can be easily incorporated to it. For instance, local mark-scans can be performed every time `Q` exceeds a certain size or after a certain number of cells are claimed from the free-list.

The Generational Algorithm

For the purpose of recording the age of cells a new counter is introduced: the *age* counter (AG). There is also a global *time* counter. The time counter is initialised with zero and is incremented every time a cell is claimed from the free-list by New.

```
New(R) = if free_list not empty then select U from free_list;
                                      AG(U) := time_counter
                                      make pointer <R,U>
                                      increment time_counter
          else if Q not empty then scan_queue; New(R)
               else write_out "No cells available"
```

If AG(R)$<$ AG(U) this means that cell R is older than cell U.

We present two ways of profiting from age information:

- To observe the age of cells and, based on the fact that young cells die young, whenever needed, run the mark-scan routines from the youngest cell on Q.
- To use age information to check for the existence of cycles. If one is sure that there are no cycles mark-scan can be performed more efficiently.

The first way presented to benefit from age information needs only to modify scan_queue, as follows:

```
scan_queue = S := youngest_black_cell_in_Q;
             { local mark-scan}
             mark_red(S); scan(S); collect_blue(S);
```

Finding the youngest black cell in Q implies scanning the whole Q, depending on the size of Q this overhead is not significant. During this process green cells can be expelled from Q.

In order to be able to spell out the possibility of cycles during mark_red we check for the condition that a *all parent cells are older than their sons*. If this condition is true we know at the end of mark_red that we are dealing with an acyclic graph. This information allows us to send cells directly into the free-list or restore their original status without the intermediate state of having these cells painted blue. For this purpose a new global variable is introduced: no_cycles. Thus we have,

```
scan_queue = S := youngest_black_cell_in_Q;
             no_cycles := true;
             { local mark-scan}
             mark_red(S); scan(S);
             { there may be cycles in the graph below S.}
             if not no_cycles then collect_blue(S);
```

mark_red is modified to check if each son is younger than its parent.

```
{ All cells are green or black. Paint red the subgraph S.
  Check for the possibility of cycles in S.}
mark_red(S) =  if colour(S) is green or black then
                    recolor(S,red);
                    for T in Sons(S) do
                      if AG(T)<AG(S) then no_cycles := false
                      mark_red(T)
```

scan makes use of the no_cycles information.

```
{ Graph S is red.
 If there are no cycles then
   Send to free-list the subgraph of S with no green pointers to it.
   Paint green the subgraph of S with green pointers to it.
 else
   Paint blue the subgraph of S with no green pointers to it.
   Paint green the subgraph of S with green pointers to it.}
scan(S) = if colour(S) is red then if RC(S)>0  then scan_green(S)
                               else if no_cycles then
                                        recolor(S,green)
                                        for T in Sons(S) do
                                               scan(T);
                                               remove <S,T>;
                                        link S to free_list
                                    else recolor(S,blue);
                                    for T in Sons(S) do scan(T)
```

Proof of Correctness

The generational algorithm can be seen as a conservative extension to the algorithm of cyclic reference counting with lazy mark-scan [6] and to prove its correctness is trivial. Age information does not interfere with other information in cells.

The first optimisation described, the choice of the youngest cell in `Q` to run `scan_queue`, brings no real change to the algorithm dynamics. Any cell could have been selected, all the generational algorithm does is to select it based on the age of cells.

Now, all we have to prove is:

1. If for the graph below a cell `S` there is any cyclic subgraph `no_cycles` will be false after `mark_red`.
2. The generational version of `scan` is correct (observes property P above).

To prove the first item we should observe that only `Copy` can make a link from a younger cell to an older one. In order to close a cycle at least one cell has to point to a cell "higher up" in the graph, by construction an older cell. Thus, if there is at least a cyclic subgraph as part of a graph under `mark_red` the variable `no_cycles` will be made false. Note that the fact that `no_cycles` being false does not imply the existence of a cyclic subgraph, but states only the possibility of its existence. Copying a pointer to an older cell "from a different branch" of the graph may also flag `no_cycles` as false.

Now we draw our attention to `scan`. The only possibility of a blue cell becoming green again is when it is on a cycle with an external reference "further down" the graph. In this case the blue cell is in the transitive closure of an externally referenced cell and will be reached by `scan_green`. If `no_cycles` is true a blue cell would never become green and all `collect_blue` would do is to send it to the free-list. That is exactly what is performed by the generational version of `scan`.

This proves the cyclic reference counting algorithm with generational reference counting correct.

Conclusions

The inclusion of generational information to reference counting brings in a new strategy of avoiding unnecessary calls to the mark-scan. With minimal overhead one can also check for cycles during marking. This allows a more efficient scan phase, saving one pass through the subgraph under analysis. The algorithm presented can be easy and advantageously incorporated to the shared memory architectures described in [4, 5].

Acknowledgements

Research reported herein has been sponsored jointly by the British Council, CAPES (Brazil) grant CBE/4875/91, and C.N.Pq. (Brazil) grants No 40.9110/88.4. and 46.0782/89.4.

References

[1] J.Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[2] A.Demers, M.Weiser, B.Hayes, D.Bobrow, and S.Shenker Combining generational and conservative garbage collection: Framework and implementations. In *Conf.Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 261-269, January 1990.

[3] H.Lieberman and C.Hewitt. A real-time garbage collection algorithm based on the lifetimes of objects. *CACM*, 26(6):419-429, June 1983.

[4] R.D.Lins. A shared memory architecture for parallel cyclic reference counting. *Microprocessing and Microprogramming*, 32:53–58, North-Holland, August 1991.

[5] R.D.Lins. A multi-processor shared memory architecture for parallel cyclic reference counting. *Microprocessing and Microprogramming*, 35:563–568, North-Holland, August 1992.

[6] R.D.Lins. Cyclic Reference Counting with Lazy Mark-Scan. to appear in *Information Processing Letters*.

[7] R.D.Lins and M.A.Vasques. A comparative study of algorithms for cyclic reference counting. in *Proc. XII Congress of the Brazilian Computing Society*, pp 17-29, Rio de Janeiro, September 1992. (also Technical Report 75, UKC Computing Lab. Report, The University of Kent at Canterbury, August 1991.)

[8] A.D.Martinez, R.Wachenchauzer and R.D.Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34:31–35, 1990.

[9] D.M.Ungar. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. in ACM SIGSOFT/SIGPLAN *Software Engineering Symposium on Practical Software Development Environments*, pages 157-167, April 1984. (also ACM SIGPLAN Notices 19(5):157-167, May/87).

[10] P.R.Wilson. Uniprocessor Garbage Collection Techniques. in *Proc. of the 1992 Inter. Workshop on Memory Management* (St.Malo, France, September 1992), LNCS 637: 1–42, Springer Verlag.