Cyclic Reference Counting with Lazy Mark-Scan

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

Key Words: Compilers, Garbage Collection, Functional Programming

Introduction

The technique usually employed for memory management in modern programming languages is one of the variants of the mark-scan or copying algorithm, since it is difficult to deal with self-referential structures using a reference count method. 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. Its time complexity is proportional to the size of the graph in use. Mark-scan and copying algorithms generally traverse all the reachable data structures during garbage collection, which makes them unsuitable for real-time or applications that make use of large-virtual-memory.

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 field `RC` 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, a mixture of mark-scan and reference counting has been used. See [1] for a detailed analysis of these algorithms.

Reference [5] presents a simple reference-counting garbage-collection algorithm for cyclic data structures, which works as a natural extension of the standard reference counting algorithm. The cost of this algorithm may be extremely low. 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. In functional languages, most structures have a reference count of one [7], and the cost of the use of this algorithm would usually be exactly the same as the standard reference-count algorithm. Unfortunately, this is not the case for object-oriented languages, which make extensive use of sharing and of cyclic data structures, making the overhead of this algorithm far too high.

We present an algorithm, called *cyclic reference counting with lazy mark-scan*, that removes the drawback of running mark-scan every time a pointer to a cell with multiple references is deleted. This new algorithm places 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. Our performance figures show that lazy mark-scan is far more efficient than local mark-scan.

The algorithm presented here is the kernel of the shared-memory architectures for parallel cyclic reference counting described in [2, 3].

The Local Mark-Scan Algorithm

The algorithm presented in [5] 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.

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, or blue. The initial colour of each node is green; the other two 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`.

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 pointers to it.*

Procedure `recolor` mantains 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. 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.

{ Cell `R` is reachable from root.
Obtain a cell `U` from free-list and create pointer `<R,U>`}

```
New(R) = select U from free-list; make pointer <R,U>
```

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

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.

If `RC(S)>1` then 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.

```
{ Delete pointer <R,S>}
Delete(<R,S>) = remove <R,S>;
                { standard reference counting}
                if RC(S) = 1  then for T in Sons(S) do
                                       Delete(<S,T>);
                                       link S to free_list
                else decrement RC(S);
                { local mark-scan}
                      mark_red(S); scan(S); collect_blue(S)
```

A cell `T` belongs to set `Sons(S)` iff there is a pointer `<S,T>`. `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.Paint red the subgraph  S.}
mark_red(S) =  if colour(S) is green 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
```

Reference [5] contains examples of applications of the algorithm above, together with an informal proof of its correctness.

The Lazy Algorithm

The new lazy algorithm uses a queue `Q` to avoid performing a local mark-scan every time a pointer to a cell with multiple references is deleted. A reference to these cells is placed on `Q`, and the cells are painted *black*. The new invariant *P'* is maintained by all procedures (assuming it is true initially).

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

If a new cell is required and the free-list is empty, the cells on `Q` are mark-scanned. These operations are performed as follows:

```
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"
```

Operation `Copy` is unchanged.

`Delete` is now far simpler than before, since the local mark-scan to multiple referenced cells is performed lazily. The colour of cells is tested black 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 on the front of `Q` and tests its colour. If black, then a local mark-scan is performed as in the original algorithm. 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` will now also allow black cells.

```
mark_red(S) = if colour(S) is green or black then
                        recolor(S,red)
                        for T in Sons(S) do mark_red(T)
```

`scan` and `collect_blue` are the same as in the original algorithm.

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.

Performance: Local versus Lazy

A formal analysis of the behaviour of these algorithms is not simple. The performance depends on the number of shared structures, number of cycles, size of `Q`, strategy used to manage `Q`, and so on. In the best possible case, the lazy algorithm would perform as many calls to mark-scan as the number of cycles needed to be recovered to run a given program. The choice of a poor control strategy can make, in the worst case, the lazy algorithm degenerate to the local one.

We present below some practical data obtained from evaluation of the program

$$fat\ 4 + fat\ 4 + fat\ 4 + fat\ 4$$

where *fat* is the *factorial* function:

```
fat n = if n=0 then 1 else n * fat (n-1)
```

This program was executed in a Turner combinator machine [6]. The code generated made intensive use of sharing. The strategy used was *mark-scan on the oldest* [4]: if `Q` is full, mark-scan is executed on the cell pointed from the front of the queue, i.e. on its oldest element. The new pointer is placed in the back of the queue. The table below summarises the results obtained:

algorithm	heap-size	mark-red	scan	scan-green	col-blue	total
standard	179	****	****	****	****	****
local	79	3188	516	2920	396	7020
lazy #1	79	1841	509	1660	313	4323
lazy #2	79	1493	449	1296	277	3475
lazy #3	79	1081	445	916	273	2715
lazy #4	79	857	141	660	249	1907
lazy #5	79	849	133	628	241	1851
lazy #10	79	537	189	368	233	1327
lazy #20	79	117	117	0	225	459

Standard reference counting needs 179 cells to run the benchmark program, while the cyclic algorithm needs only 79 cells. As we can observe, a queue `Q` of size four (#4), equivalent to about 5% of the heap size, reduces the total number of functional calls to mark-scan to less than 28% of the number of calls to the local mark-scan algorithm. The inexistence of calls to `scan_green` in the last line of the table indicates that the algorithm has been used only to collect cycles, i.e. no unnecessary calls to mark-scan took place. In this case, the data presented shows that the lazy algorithm is far superior to the local one.

Variable-size Queues

In the lazy mark-scan algorithm, queue `Q` is implemented as a separate data-structure outside the heap. It makes no use of the spare cells in the heap. The cells in the free-list can be used to implement `Q`. Instead of placing a cell `S` in `Q` we get a cell `U` from the free-list, append `U` to `Q`, and store in `U` a pointer to `S`.

```
Delete(<R,S>) = remove <R,S>
                if RC(S) = 1 then for T in Sons(S) do
                                      Delete(<S,T>);
                                      link S to free_list;
                else decrement RC(S);
                     if colour(S) not black then
                          colour(S) := black;
                          (U = New(last_of_Q)) := S;
                          last_of_Q := U
```

If the free-list is empty `New` will de-queue cells from queue `Q`.

Performance: Fixed-size versus Variable-size

The use of the spare cells in the heap for implementing `Q` brings a substantial increase in the performance to the lazy algorithm, as can be seen in the table below, for the same benchmark:

algorithm	heap-size	mark-red	scan	scan-green	col-blue	total
standard	179	* * **	* * **	* * **	* * **	* * **
local	79	3188	516	2920	396	7020
lazy vs	79	672	209	482	237	1600
lazy #1	79	1841	509	1660	313	4323
lazy vs	80	500	190	336	234	1260
lazy vs	81	504	190	339	234	1267
lazy vs	179	78	78	0	150	306
lazy vs	279	0	0	0	0	0

Line *lazy* #1 corresponds to the lazy algorithm with queue size 1, and *lazy vs* are data for the algorithm with variable-size queue. As we can observe, a heap of minimum size to run the benchmark program reduced the number of function calls to 25% of the local mark-scan algorithm, which is equivalent to a queue of fixed size 7. Using 80 cells in total, the algorithm with variable-size queue needs only 34% of the number of function calls performed by the fixed-size algorithm. For a more detailed comparison between these algorithms see [4].

Conclusions

The efficiency of the algorithm presented in this paper is much higher than the original one for cyclic reference counting with local mark-scan. More shared cells will now be claimed directly, without any need for mark-scan. 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. The queue will be left basically with pointers to cycles and pointers to green cells in the free-list or recycled. In this case again, our algorithm performs far better than the original one. In the best case, only one local mark-scan will be performed per cycle, instead of as many as the number of external references to a cycle, as before.

Acknowledgements

I express gratitude to Prof. David Gries for his comments on a previous version of this paper, and to Márcio A.Vasques for providing the experimental data presented.

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

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

[4] R.D.Lins and M.A.Vasques. A comparative study of algorithms for cyclic reference counting. Technical Report 75, UKC Computing Lab. Report, The University of Kent at Canterbury, August 1991.

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

[6] D.A. Turner. A new implementation technique for applicative languages. *Software — Practice and Experience*, 9, 1979.

[7] W.R.Stoye, T.J.W.Clarke & A.C.Norman. Some practical methods for rapid combinator reduction. In *Proc. of ACM Symposium on Lisp and Functional Programming*, pages 159–166, Austin, August 1984. ACM.