Journal of Universal Computer Science, vol. 13, no. 6 (2007), 830-838
submitted: 19/1/07, accepted: 22/3/07, appeared: 28/6/07 © J.UCS

Cyclic Reference Counting with Permanent Objects

Rafael Dueire Lins
(Departamento de Eletrônica e Sistemas, CTG
Universidade Federal de Pernambuco, 50.740-530 – Recife – PE – Brazil
rdl@ufpe.br)

Francisco Heron de Carvalho Junior
(Departamento de Computação, Universidade Federal do Ceará,
50.740-530 – Fortaleza - CE – Brazil
heron@lia.ufc.br)

Zanoni Dueire Lins
(Departamento de Eng. Elétrica e Sistemas e Sistemas de Potência, CTG,
Universidade Federal de Pernambuco, 50.740-530 – Recife – PE – Brazil
zdl@ufpe.br)

Abstract: Reference Counting is the memory management technique of most widespread use today. Very often applications handle objects that are either permanent or get tenured. This paper uses this information to make cyclic reference counting more efficient.

Keywords: Memory management, garbage collection, reference counting, cyclic graphs, permanent objects, tenured objects.
Categories: D.4.2

1 Introduction

Reference counting as described in [Collins 60] and [Jones and Lins 96] is a simple memory management technique in which each data structure keeps the number of external references (or pointers) to it. It was developed by [Collins 60] to avoid user process suspension provoked by the mark-scan algorithm in LISP. Reference counting performs memory management in small steps interleaved with computation. [Mc Beth 63] noticed that reference counting was unable to reclaim cyclic structures, because the counter of cells on a cycle never drops to zero, causing a space-leak, as may be observed in Figure 1.

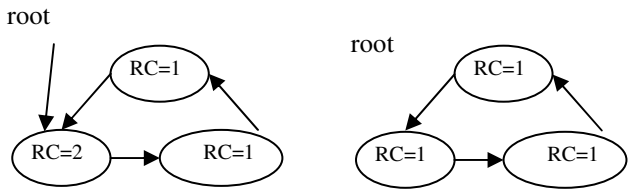

Figure 1: Isolating a cycle from root causes a space-leak

In real applications cyclic structures appear very often. For instance, recursion is frequently represented by a cyclic graph and web pages have hyperlinks to other web pages that frequently point back to themselves [Lins 06]. These are two examples that may give an account of the importance of being able to handle cycles in reference counting. Several researchers looked for solutions for this problem.

The first general solution for cyclic reference counting was presented in reference [Martinez et al. 90], where a local mark-scan is performed whenever a pointer to a shared data structure is deleted. Lins largely improved the performance of the algorithm in two different ways. The first optimization as described in [Lins 92b] widely acknowledged as the first efficient solution to cyclic reference counting, postpones the mark-scan, as much as possible. This algorithm is implemented is both IBM Java machines developed at IBM T.J.Watson and IBM-Israel in Cooperation with the Technion, both of them reporting excellent performance as described in [Bacon et al. 01a,b]. The second optimization [Lins 93] relies on a creation-time stamp to help in cycle detection.

A decade later than the general solution to cyclic reference counting was presented, Lins introduced the *Jump_stack*, a data structure which largely increases the efficiency of the previous algorithms presented in [Lins 02]. This data structure stores a reference to the "critical points" in the graph while performing the local marking (after the deletion of a pointer to a shared cell). These nodes are revisited directly, saving a whole scanning phase in.

One of the strategies used in optimizing compilers and applications is to recognize whenever data is permanent or is "so old" that may be tenured. Handling such objects in a distinctive fashion avoids making copies of them and all the computational effort involved in its management. This paper introduces permanent objects to cyclic reference counting, increasing the efficiency of the previous algorithms.

2 Efficient Cyclic Reference Counting

The algorithm with permanent objects is designed on top of the efficient cyclic reference counting algorithm presented in [Lins 02]. Thus, it is explained in this section. The general idea of the algorithm is to perform a local mark-scan whenever a pointer to a shared structure is deleted. The algorithm works in two steps. In the first step, the sub-graph below the deleted pointer is scanned, rearranging counts due to internal references, marking nodes as possible garbage and also storing potential links to root in a data structure called the "*Jump-stack*". In step two, the cells pointed at by the links stored in the Jump-stack are visited directly. If the cell has reference count greater than one, the whole sub-graph below that point is in use and its cells should have their counts updated. In the final part of the second step, the algorithm collects garbage cells.

Now, implementation details of the algorithm are presented. As usual, free cells are linked together in a structure called *free-list*. A cell $\boldsymbol{B}$ is *connected* to a cell $\boldsymbol{A}$ ($\boldsymbol{A \rightarrow B}$), if and only if there is a pointer $\boldsymbol{<A, B>}$. A cell $\boldsymbol{B}$ is *transitively connected* to a cell $\boldsymbol{A}$ ($\boldsymbol{A \overset{*}{\rightarrow} B}$), if and only if there is a chain of pointers from $\boldsymbol{A}$ to $\boldsymbol{B}$. The initial point of the graph to which all cells is use are transitively connected is called *root*. In

addition to the information of number of references to a cell, an extra field is used to store the color of cells. Two colors are used: green and red. Green is the stable color of cells. All cells are in the free-list and are green to start with.

There are three operations on the graph:

New(R) gets a cell U from the free-list and links it to the graph:

```
New (R) = select U from free-list
          make_pointer <R, U>
```

Copy(R, <S,T>) gets a cell R and a pointer <S, T> to create a pointer <R, T>, incrementing the counter of the target cell:

```
Copy(R, <S,T>) = make_pointer <R, T>
                 Increment RC(T)
```

Pointer removal is performed by Delete:

```
Delete (R,S) = Remove <R,S>
               If (RC(S) == 1) then
                  for T in Sons(S) do
                      Delete(S, T);
                  Link_to_free_list(S);
               else Decrement_RC(S);
                    Mark_red(S);
                    Scan(S);
```

A cell T belongs to the bag Sons(S) iff there is a pointer <S,T>. One can observe that the only difference to standard reference counting in the algorithm above rests in the last two lines of Delete, which will be explained below. Mark_red is a routine that "analyzes" the effect of the deleted pointer in the sub-graph below it. The sub-graph visited has the counts of cells decremented. Whenever a cell visited remains with count greater than one two possibilities may hold:

1. The cell is an entry point of root into the sub-graph below it.
2. The value is a transient one and may become zero at a later stage of Mark_red, indicating that it is not an entry point from root.

To perform this analysis whenever a cell is met by Mark_red with count greater than one after decrementing, it is placed in the Jump_stack. The code for Mark_red follows:

```
Mark_red(S) =  If (Color(S) == green) then
                  Color(S) = red;
                  for T in Sons(S) do
                      Decrement_RC(T);
                      if (RC(T)>0 &&
                          T not in Jump_stack)
                       then Jump_stack = T;
                      if (Color(T) == green)
                       then  Mark_red(T);
```

Scan(S) verifies whether the Jump_stack is empty. If so, the algorithm sends cells hanging from S to the free-list. If the jump-stack is not empty there are nodes in the graph to be analysed. If their reference count is greater than one, there are external pointers linking the cell under observation to root and counts should be restored from that point on, by calling the ancillary function Scan_green(T).

```
Scan(S) =  If RC(S)>0 then Scan_green
```

```
else
  While (Jump_stack ≠ empty) do
    T = top_of_Jump_stack;
    Pop_Jump_stack;
    If (Color(T) == red && RC(T)>0)
    then
         Scan_green(T);
  Collect(S);
```

Procedure Scan_green restores counts and paints green cells in a sub-graph in use, as follows,

```
Scan_green(S) = Color(S) = green
                  for T in Sons(S) do
                      increment_RC(T);
                      if color(T) is not green
                      then
                          Scan_green(T);
```

Collect(S) is the procedure in charge of returning garbage cells to the free-list, painting them green and setting their reference count to one, as follows:

```
Collect(S) =  If (Color(S) == red) then
                for T in Sons(S) do
                   Remove(<S, T>);
                   RC(S) = 1;
                   Color(S) = green;
                   free_list = S;
                   if (Color(T) == red) then
                      Collect(T);
```

3 Cyclic Reference Counting with Permanent Objects

As already mentioned in the introduction of this paper, permanent objects appear very often in real implementations of systems and languages. Treating such objects differently from temporary ones is a way to increase the efficiency of the cyclic reference counting algorithm presented above. Below, the algorithm presented in the last section is modified to handle permanent objects efficiently. Operations are explained in terms of the same atomic actions presented above.

New(R) gets a cell U from the free-list and links it to the graph. The color of the new object depends on its nature. Temporary objects are set as "green" while permanent objects are set as "white". Permanent objects have their reference count set to "overflow".

```
New (R) = select U from free-list
          make_pointer <R, U>
          if  R is permanent then (color(R):= white); RC(R):=∞;
                                  else (color(R):= green)
```

Although it may at first sight that permanent objects increase the complexity of the allocation routine New, in reality this is not the case. Permanent objects appear during graph creation, instead of during graph manipulation, without any need to perform the testing during run-time. Another low-cost alternative is to allow two different combinators for cell creation one for permanent objects and another for temporary ones:

```
New Perm (R) = select U from free-list
```

```
                make_pointer <R, U>
                color(R):= white; RC(R):=∞;

New_Temp (R) = select U from free-list
                make_pointer <R, U>
                color(R):= green
```

Copy(R, <S,T>) gets a cell R and a pointer <S, T> to create a pointer <R, T>. If the target T cell is temporary its count gets incremented:

```
Copy(R, <S,T>) = make_pointer <R, T>
                  if  color(T) is not  white
                  then Increment RC(T)
```

Again, the increase in the complexity of this operation is apparent. The color test needs not to be performed in the case of having counts with overflow. Thus it is possible to make use of the definition of Copy as before, provided that one has in account that the value of counts in permanent object is not consistent, i.e. does not stand for the number of pointers to it.

```
Copy(R, <S,T>) = make_pointer <R, T>
                  Increment RC(T)
```

Pointer removal is performed by Delete, which may remain unaltered it one assumes that the decrement of overflow remains the same.

```
Delete (R,S) = Remove <R,S>
               If (RC(S) == 1) then
                     for T in Sons(S) do
                          Delete(S, T);
                     Link_to_free_list(S);
                else Decrement_RC(S);
                     Mark_red(S);
                     Scan(S);
```

The real changes in the algorithm with permanent objects appear during the mark-scan. Permanent objects never have their counts altered. Whenever a permanent object is part of a cycle under the local mark-scan it stops further analysis. Although Mark_red remains with the same definition as before, one must observe that the analysis does not propagate through *white* (permanent) cells.

```
Mark_red(S) =  If (Color(S) == green) then
                  Color(S) = red;
                  for T in Sons(S) do
                       Decrement_RC(T);
                       if (RC(T)>0 &&
                          T not in Jump_stack)
                        then Jump_stack = T;
                       if (Color(T) == green)
                        then  Mark_red(T);
```

As before, Scan(S) verifies whether the Jump_stack is empty. If so, the algorithm sends cells hanging from S to the free-list. If the jump-stack is not empty there are nodes in the graph to be analyzed. If their reference count is greater than one, there are external pointers linking the cell under observation to root and counts should be restored from that point on, by calling the ancillary function Scan_green(T).

```
Scan(S) =  If RC(S)>0 then Scan_green
```

```
else
  While (Jump_stack ≠ empty) do
    T = top_of_Jump_stack;
    Pop_Jump_stack;
    If (Color(T) == red && RC(T)>0)
    then
         Scan_green(T);
  Collect(S);
```

Procedure Scan_green visits only red cells, restores their counts and paints green cells in a sub-graph in use. Thus it is slightly modified to:

```
Scan_green(S) = If Color(S) = red then
                  Color(S): = green
                  for T in Sons(S) do
                       increment_RC(T);
                       if color(T) is  red
                       then
                            Scan_green(T);
```

Collect(S) is the procedure in charge of returning garbage cells to the free-list, setting their reference count to one, as follows:

```
Collect(S) =  If (Color(S) == red) then
                for T in Sons(S) do
                  if (Color(T) == red) then
                       Collect(T);
                 RC(S):= 1;
                 free_list := S;
```

4 Tenuring objects

The generational hypothesis states that "young objects die young and old objects tend to remain in use until the end of computation". Taking this into account, very often systems and languages tend to give a permanent status to objects that "live" over a certain time or operational barrier. This change of status is called "tenure".

Several different tenuring policies may be adopted. One of them is change into white the color of a given object whenever it were green and have a reference count greater than a certain threshold value "*t*". This strategy changes the code for Copy into:

```
Copy(R, <S,T>) = make_pointer <R, T>
                  If RC(T) ≥ t and  color(T)== green
                       then color(T);= white
                       Increment RC(T)
```

Notice that tenuring polices may yield to space-leaks, as tenured objects may become garbage. The code for Delete remains unchanged. One should observe that it may claim white cells provided one is removing the last reference to it. Tenured cells only avoid the propagation of the local mark-scan through them. That means that the space leak only involves cyclic structures of tenured cells.

4.1 Avoiding Space-leaks

A possibility of having tenured objects and avoiding permanent cell loss that cause space leaks is to introduce a new color, "*grey*" to tenured objects and place them in the Jump-stack for later analysis. This will cause the redefinition of Copy as:

```
Copy(R, <S,T>) = make_pointer <R, T>
                    If RC(T) ≥ t and  color(T)== green
                        then color(T);= grey
                              Jump_stack:= T
                    Increment RC(T)
```

One should notice that grey objects hold their actual reference count value. Tenured objects are analyzed only at last, i.e. whenever the free-list is empty. At this moment, the Jump_stack is empty. Thus, the code for New is now written as:

```
New (R) = if free-list not empty then
                select U from free-list
                make_pointer <R, U>
                 if  R is permanent then (color(R):= white); RC(R):=∞;
                                else (color(R):= green)
          else
             If Jump_stack not empty then
                For T in Jump_stack  do
                        NMark_red(T)
                Scan(T)
                New(R)
           else
                write_out "No cells available; execution aborted";
```

NMark_red is a new procedure that takes into account the possibility of cells being grey.

```
NMark_red(S) =  If (Color(S) == green or grey) then
                    Color(S) = red;
                    for T in Sons(S) do
                        Decrement_RC(T);
                        if (RC(T)>0 &&
                            T not in Jump_stack)
                         then Jump_stack = T;
                        if (Color(T) == green or grey)
                         then  Mark_red(T);
```

The code for Delete remains unchanged. One should observe that it may claim grey cells directly, provided one is removing the last reference to it. Grey cells only avoid the propagation of the local mark-scan.

The tenuring policy must be carefully adopted as it either may cause space-leaks or a high operational overhead.

5 Proof of the Correctness

Providing formal proofs of the correctness of algorithms is not a simple task. This section elucidates the on the correction of the algorithms presented in this paper. The starting point is assuming the correctness of the algorithm for efficient cyclic reference counting as described in [Lins 02].

5.1 Cyclic RC with Permanent Objects

Permanent objects may be seen as objects that are permanently linked to root. Thus, the only role they play is to stop the propagation of the local mark-scan. In doing so the algorithm saves the need of placing the object onto the Jump_stack during Mark_red and later, during Scan removing it from the Jump_stack and, as its reference count will never drop to allow the object to be collected, having to call Scan_green on it. The changes introduced into the code of the routines implement the operations described above making explicit the possibility of handling *white* objects.

5.2 Tenured Objects

The policy presented above to decide when an object may be considered permanent was focused on the number of references to it. There is a hidden assumption that the higher the number of references the longer lived will be the object. Under such hypothesis, all the algorithm does is to tenure a temporary object making it a permanent one. As one has already argued for the correctness of the algorithm with permanent objects, this change in status does not alter the overall behavior of the algorithm, provided one may be able to accept the possibility of a space-leak. This is the case when the deletion of the last pointer isolates an island from root with no elements to any later analysis.

5.3 Tenured Objects without Space-leaks

The whole idea of the algorithm for tenured objects without space-leaks is to keep references (in the Jump_stack) for deciding later about the validity of a tenured object. The analysis of possible candidates for recycling is performed in extreme circumstances only, i.e. whenever the free-list is empty.

6 Conclusions

Permanent objects appear very often in real applications, thus addressing them in an efficient way is a matter of concern to whoever implement systems, languages, etc. This paper shows how to introduce permanent objects to cyclic reference counting in a simple way, with almost no overhead to atomic operations, but avoiding the unnecessary propagation of the local mark-scan. Besides that, this paper shows two different ways of working with tenured objects. The first alternative may cause space-leaks of cyclic data structures encompassing tenured objects. The second alternative makes possible to reclaim all garbage at a high operational cost. Both alternatives point in the direction of having a conservative tenuring policy to avoid overheads.

The presented algorithm will certainly have a large impact in the decreasing amount of communication exchanged between processors either in shared-memory architectures as described in [Lins 91] and [Lins 92] or in distributed environments presented in [Lins 06] and [Lins and Jones 93], thus bringing more efficiency in tightly and loosely coupled systems.

Acknowledgements

This work was sponsored by CNPq – Brazilian Government.

References

[Bacon et al. 01a] D.F.Bacon and V.T.Rajan. "Concurrent Cycle Collection in Reference Counted Systems", Proceedings of European Conference on Object-Oriented Programming, June, 2001, Springer Verlag, LNCS vol 2072.

[Bacon et al. 01b] D.F.Bacon, C.R.Attanasio, H.B.Lee, R.T.Rajan and S.Smith. "Java without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector", Proc.of the SIGPLAN Conf. on Programming Language Design and Implementation, June, 2001 (Not. 36,5).

[Collins 60] G.E. Collins, "A method for overlapping and erasure of lists", Comm. of the ACM, 3(12):655—657, Dec.1960.

[Jones and Lins 96] R.E. Jones and R.D. Lins, "Garbage Collection Algorithms for Dynamic Memory Management", John Wiley & Sons, 1996.

[Lins 91] R.D.Lins,"A shared memory architecture for parallel cyclic reference counting, Microprocessing and microprogramming", 34:31—35, Sep. 1991.

[Lins 92a] R.D. Lins, "A multi-processor shared memory architecture for parallel cyclic reference counting, Microprocessing and microprogramming", 35:563—568, Sep. 1992.

[Lins 92b] R.D.Lins, "Cyclic Reference counting with lazy mark-scan", Information Processing Letters, vol 44 (1992) 215—220, Dec. 1992.

[Lins 93] R.D.Lins, "Generational cyclic reference counting", Information Processing Letters, vol 46 (1993) 19—20, 1993.

[Lins 02] R.D.Lins. "An Efficient Algorithm for Cyclic Reference Counting", Information Processing Letters, vol 83 (3), 145-150, North Holland, August 2002.

[Lins 06] R.D.Lins, "New algorithms and applications of cyclic reference counting", Invited Keynote Paper, Proceedings of ICGT 2006 – International Conference on Graph Transformation and Applications, LNCS, Springer Verlag, September 2006.

[Lins and Jones 93] R.D. Lins and R.E.Jones, "Cyclic weighted reference counting", in K. Boyanov (ed.), Proc. of Intern. Workshop on Parallel and Distributed Processing, NH, 1993.

[Mc Beth 63] J.H. McBeth, "On the reference counter method", Comm. of the ACM, 6(9):575, Sep. 1963.

[Martinez et al. 90] A.D. Martinez, R. Wachenchauzer and R. D. Lins, "Cyclic reference counting with local mark-scan", Information Processing Letters vol. 34 (1990)31-35, North Holland, 1990.