Cyclic reference counting for combinator machines

D.R. Brownbridge

Abstract

This new algorithm deals correctly and automatically with the kind of cyclic (i.e. self-referencing) structures which arise in a combinator graph reduction machine. By extending the standard reference count algorithm, cycles can be handled safely at little extra cost. Cyclic reference counting uses one extra bit per pointer and per object and one extra reference count per object. Extra computation amounts to inspection of the objects in a cyclic structure from time to time before the structure becomes free. In the absence of cycles, no computation overhead is incurred. When executing some typical recursive programs, the number of objects inspected is approximately equal to the number of new objects used during execution.

This paper is to appear in Proc. IFIP Int. Conference on Functional Programming Languages and Computer Architecture (16th-18th September 1985), Springer-Verlag.

Series Editor: M. J. Elphick

© 1985 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.
Bibliographical details

BROWNBRIDGE, David Robert.
Cyclic reference counting for combinator machines. [By]
D.R. Brownbridge.
Newcastle upon Tyne: University of Newcastle upon Tyne,
Computing Laboratory, 1985.

(University of Newcastle upon Tyne, Computing Laboratory,
Technical Report Series, no. 207.)

Added entries
UNIVERSITY OF NEWCASTLE UPON TYNE.

Abstract
This new algorithm deals correctly and automatically with the kind of
cyclic (i.e. self-referencing) structures which arise in a combinator
graph reduction machine. By extending the standard reference count
algorithm, cycles can be handled safely at little extra cost. Cyclic
reference counting uses one extra bit per pointer and per object and
one extra reference count per object. Extra computation amounts to
inspection of the objects in a cyclic structure from time to time
before the structure becomes free. In the absence of cycles, no
computation overhead is incurred. When executing some typical re-
cursive programs, the number of objects inspected is approximately
equal to the number of new objects used during execution.

About the author
Dr. D.R. Brownbridge was a Research Student and then a Research Associate
in the Computing Laboratory until 1984. He is now at High Level Hardware
Ltd., P.O. Box 170, Windmill Road, Headington, Oxford, England, OX3 7BN.

Suggested keywords
COMBINATOR MACHINES
DISTRIBUTED COMPUTING SYSTEMS
GARBAGE COLLECTION
STORAGE MANAGEMENT

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404
U.D.C. 519.687
CYCLIC REFERENCE COUNTING FOR COMBINATOR MACHINES

D. R. Brownbridge

Computing Laboratory,
University of Newcastle upon Tyne,
Clarendon Road,
Newcastle upon Tyne,
England.

ABSTRACT

This new algorithm deals correctly and automatically with the kind of cyclic (i.e., self-referencing) structures which arise in a combinator graph reduction machine. By extending the standard reference count algorithm, cycles can be handled safely at little extra cost. Cyclic reference counting uses one extra bit per pointer and per object and one extra reference count per object. Extra computation amounts to inspection of the objects in a cyclic structure from time to time before the structure becomes free. In the absence of cycles, no computation overhead is incurred. When executing some typical recursive programs, the number of objects inspected is approximately equal to the number of new objects used during execution.

Introduction

The motivation of this work is the need to store structured objects in a distributed computer system. In such a system, many computers operate concurrently; no one computer can learn the state of the whole system and no one activity can require synchronised action by all the computers[1]. Such systems are seen by many as the best way to exploit advances in computer technology and may in turn provide good support for functional programming languages[2,3,4,5,6,7].

In this context the most commonly used storage management algorithm, (mark-scan garbage collection[8]) is wholly inappropriate. It can require both synchronisation of all the computers in the system and a co-ordinated search of their entire storage. On the other hand, reference counting[8,9] has very attractive properties of localised and incremental processing. The major problem is of course that reference

counting is not fully general; it cannot deal with self-referencing cycles of objects.

When the storage manager is to be incorporated as a fundamental feature of the system it must be able to manage all structures correctly. Even in systems where cycles are considered undesirable and only arise in error[4,10], the storage manager has to deal properly with them if storage space is not to be lost forever. Many modified versions of mark-scan have appeared, including 'copying' and 'on-the-fly' garbage collectors [11,12,13,14,15]. These go a long way toward removing the disadvantages of mark-scan whilst retaining efficiency and effectiveness. This paper attempts to remove the disadvantages of reference counting in the belief that the result may be a better compromise than modified mark-scan algorithms[16,17,18,19,20].

Cyclic reference counting was evolved as the 'minimal' addition to standard reference counting which can deal with cycles. Three 'micro-code' pointer operations NEW, COPY and DELETE are used to implement graph manipulating combinators as described by Turner[21]. The nature of combinators ensures that all cycles result from the executions of exactly one instruction (the Y combinator). At the point where Y is executed a single 'cyclic' pointer is created and marked as being 'weak'. Subsequent pointer manipulations proceed as usual taking care to ensure that copies of weak pointers are also marked as being weak. In most cases, pointer deletion can be handled in the standard way. By observing the relative numbers of strong and weak pointers to an object, the deletion of cycles can be handled properly. This is achieved using a new method described below, forming the main innovation of this paper.

The appendix describes a form of cyclic reference counting which would be appropriate to support languages such as Pascal or LISP where arbitrary assignment to pointers is permitted. This extension will be the subject of a further paper.

Implementing Combinators

A graph reduction machine is an alternative form of computer in which the program is represented by a directed graph. Program execution amounts to replacing (reducing) selected sub-graphs in situ, so that eventually the program is replaced by its result[22,23,24,25].

Combinator graph reduction machines (in short 'combinator machines') are a particular class of reduction machines which use combinators [26,27,21] to perform name-value binding and any associated copying or sharing. Combinators play the same role as 'displays' or the 'static chain' found in many high level language implementations[28]. The advantage is that each combinator is simple to implement, performing some small incremental operation and that functional programs can be compiled into combinator form easily. For our purpose, combinators are also attractive because they provide a complete, well-defined interface between the program and the storage manager. All manipulation of
the program graph is performed by combinators.

Subsequent sections of this paper will describe the standard and cyclic reference counting algorithms in terms of three canonical pointer operations:

Let R, S, T denote arbitrary objects currently in use (i.e. reachable from the Root) and let <R, S> denote a pointer which exists from R to S.

1) NEW(R)
   Create a pointer from object R to some free object.
   (The result is to bring a free object into use.)

2) COPY(R, <S, T>)
   Create a pointer from R to T (by copying pointer <S, T>).

3) DELETE(<R, S>)
   Destroy a pointer from R to S.

   In a combinator machine these operations can be 'micro-instructions' used to implement the combinator instruction set. As such they are not made available to programs. The micro-programs to implement each combinator will consist of a few micro-instructions each.

   object program
      ----------------- combinators: S, K, Y, etc.
      microcode
      ----------------- new, copy, delete, etc.
      storage manager

   The only complication introduced by cyclic (as opposed to standard) reference counting is an extra parameter to COPY:

   2) COPY(R, <S, T>, c)
      Create a pointer from R to T - cyclic reference count version
      If 'c' is true then a cycle is being created so the new pointer is marked as 'weak';
      otherwise no cycle is being created so the new pointer is marked 'strong'.

   To implement combinators using cyclic reference counting the following rule is sufficient to determine the value of 'c':

   (a) When executing the Y (recursion) combinator:
      COPY(R, <S, T>, true)

   (b) When executing any other instruction:
      COPY(R, <S, T>, isweak(<S, T>))
Where \texttt{isweak} simply has value \texttt{true} if the pointer from \texttt{S} to \texttt{T} is weak and \texttt{false} otherwise. In this way 'strength' of pointers is preserved when copying.

As a result of this rule, recursive function calls are bound using weak pointers and all other pointers are strong.

For example the recursive function 'f'

\[ f \ x = \ldots \ f \ \ldots \ f \ \ldots \ f \ \ldots \]

will be compiled into

\[ \text{Y} \]

\[ \text{F} \]

where 'F' is the compiled body (right hand side) of 'f'. Executing Y creates a self-reference to 'F' using a weak pointer[21,26]:

\[ \text{F} \]

Further reductions within 'F' will bind that self-reference to the recursive calls in the function body, creating a weak pointer for each:

\[ \text{F} \]

Note that only one node in this graph has weak pointers to it, hence only that node need have separate reference counts for strong and weak pointers. The remainder use only one reference count.

Finally, when 'f' is no longer needed the entire storage it occupied will be automatically reclaimed by cyclic reference counting.

\section*{The Standard Reference Counting Algorithm}

The usual reference counting algorithm [8,9] associates a single value (\texttt{Refc(X)}) with each stored object (\texttt{X}). Each operation on a pointer is accompanied by manipulation of reference counts. Without loss of generality, we assume a single object named \texttt{Root} which is regarded as
being always in use. Any objects reachable by traversing pointers from Root are said to be also in use; the remaining objects in the store are free. The aim of storage management is to keep track of which objects are in use and which are free.

Again without loss of generality, we assume it is sufficient to ensure that all free objects have a reference count of zero and any object in use (except possibly Root) has a positive reference count. The algorithm can now be summarised by the following operations on reachable objects R, S and T.

(1) NEW(R)
Create a pointer from object R to some free object.
(The result is to bring a free object into use.)

Select object U such that Refc(U)=0;
Set Refc(U) to 1. Create pointer <R,U>

(2) COPY(R,<S,T>)
Create a pointer from R to T.
Add one to Refc(S). Create pointer <R,T>

(3) DELETE(<R,S>)
Destroy a pointer from R to S.
Subtract one from Refc(S); Remove pointer <R,S>
if Refc(S)=0 then
for T in Sons(S) do DELETE(<S,T>) od

That is, if there are no more references to S, it has become free, so recursively delete the pointers which it contains.

These operations are sufficient for a 'low level' interface and can be used to encode the more usual graph manipulating operations as well as combinators. Let x, y represent distinct pointers <x1,x2>, <y1,y2> respectively:

pointer assignment:  x := y

DELETE(<x1,x2>); COPY(x1,<y1,y2>)

CONS from LISP: res := CONS(x,y)

NEW(res); COPY(res,<x1,x2>); COPY(res,<y1,y2>)

Cyclic structures are simply those where an object is reachable from itself. Starting from any given object in a cyclic structure, that object can be reached again by traversing one or more pointers. The standard reference counting algorithm is not guaranteed to deal correctly with cyclic structures. Because cycles involve (possibly indirect) self-reference, a positive reference count is not a certain indication of being in use.
Strong and Weak Pointers

Cyclic reference counting operates by assigning each pointer to one of two classes: 'strong' and 'weak'. Strong pointers do not form cycles and can therefore be managed safely with the standard algorithm. Intuitively, the weak pointers are the ones that cause cycles - the ones which cause the standard algorithm to fail.

Cyclic reference counting can be made to work because:

(a) Pointers can automatically be classified as strong or weak by observing that only Y creates cycles and that it suffices to preserve strength/weakness when copying pointers.

(b) An algorithm has been found for using strong and weak pointers to manage cycles correctly. It requires separate reference counts for strong and for weak pointers (denoted SRefc and WRefc respectively).

The rules for classifying pointers as strong or weak are as follows:

Rule 1:
The strong pointers together with the objects in use form a connected, acyclic, directed graph in which every object is reachable from Root.

As a result of Rule 1, every object in use (except possibly Root) has a strong pointer pointing to it. Consequently the reference count of strong pointers to every object in use is greater than zero. That is:

\[ SRefc(R) > 0 \] for all \( R \) in use

Rule 2:
Pointers which are not strong are weak.

Hence the net reference count of every object is the sum of its strong and weak counts:

\[ \text{Refc}(R) = SRefc(R) + WRefc(R) \]

Consider some examples showing structures divided into strong and weak pointers:

(1) Root:  
```
    a:  
```

(2) Root:  
```
    a:  
    b:  
```
Weak pointers are represented by dotted lines. Check that each structure is formed according to Rules 1 and 2. Reference counts can easily be inferred. For example in structure (3), Root has a weak reference count of 2 and a strong reference count of 0 and the two other objects have counts of 0 weak and 1 strong each. Structures (4) and (5) are isomorphic, demonstrating that there may be more than one valid way to partition the pointers into strong and weak for any given structure. Any combinator graph representing a functional program will satisfy the rules given above although some other graphs generated by NEW, COPY and DELETE do not.

**Cyclic Reference counting**

The new versions of NEW, COPY and DELETE which manage the strong and weak pointers can now be described. Again we assume that R,S are objects currently in use.

**1. NEW(R)**

Create a pointer from object R to some free object. (The result is to bring a free object into use.)

Select U such that SRefc(U)=0 and WRefc(U)=0;
Set SRefc(U) to 1. Create pointer <R,U>

Free objects contain no pointers (they are DELETED automatically) so S cannot lie on a cycle.
(2) COPY(R, <S, T>, c)

Create a pointer from R to T ('c' is an extra, boolean parameter).

if 'c' then assume the new pointer will create a cycle so create
< R, T > marked as weak and add 1 to WRefc(T)
else assume the new pointer will not cause a cycle so create
< R, T > marked as strong and add 1 to SRefc(T)

Correct operation depends entirely on the correct value of 'c' being
supplied according to Rules 1 and 2. This is easily achieved when
NEW, COPY and DELETE are used to implement combiners. As shown
above: 'c' is set to true when executing Y and otherwise takes value
isweak(< S, T >).

The appendix describes a more complex cyclic reference count
algorithm which does not require parameter 'c'. This is important
because in many applications it is as expensive to decide the correct
value for 'c' as it would be to do simple mark-scan garbage collection.
In the particular case of combinator machines we are fortunate that 'c'
can be determined automatically.

(3) DELETE(<R, S>)

Destroy a pointer from R to S.

Four cases arise depending on the relative values of SRefc(S) and
WRefc(S) and the kind of pointer being deleted.
(i) If the pointer <R, S> is weak:

Subtract 1 from WRefc(S) Remove pointer <R, S>

No further action is required; by Rule 1 every object in use has
SRefc > 1 so deleting a weak pointer never frees anything.

(ii) If the pointer <R, S> is strong and WRefc(S)=0

Subtract 1 from SRefc(S) and proceed as normal:
if SRefc(S)=0 then
  for T in Sons(S) do DELETE(<S, T>) od

By rule 1 strong pointers are acyclic; so when there are no weak
pointers the standard algorithm is applied.

(iii) If <R, S> is strong WRefc(S)>0 and SRefc(S)>1

Subtract 1 from SRefc(S) Remove pointer <R, S>
No further action is required.

(iv) If <R, S> is strong and WRefc(S)>0 and SRefc(S)=1

When deleting the last strong pointer and there are weak
pointers.
This is exactly the situation in which standard reference counting will fail. In cases (i), (ii) and (iii) we can be sure that DELETION leaves the structure correctly formed in the sense that:

(a) Free objects have zero reference counts
(b) The reference counts correspond to the actual pointers
(c) The pointers are correctly classified into strong and weak according to rules 1 and 2.

In this fourth case of DELETE, it cannot be decided immediately whether S has become free or not. There are two possibilities, demonstrated by examples (2) and (4) above. In example (2), DELETE(<Root, a>) does free 'a'; in example (4), the same operation does not cause 'a' to become free even though the reference counts at 'a' are the same in both cases.

In case (iv) of DELETE, a recursive search is made visiting the sub-objects pointed to by S. This attempts to find an 'external' pointer (such as <Root, b> in example 4) which will mean that S is still reachable from Root. At the same time it is possible to adjust the strength of pointers to ensure that after the search (a) (b) and (c) will hold. In example (4), this will ensure there is a strong pointer to a by making <a, b> weak and <b, a> strong. Thus after DELETE(<Root, a>) example (4) is left isomorphic to example (2).

The final case of DELETE can now be specified:

DELETE(<R, S>)
  Destroy a pointer from R to S.
(iv) When <R, S> is strong and WRefc(S)>0 and SRefc(S)=1
(a) Set SRefc(S) to 0 Remove pointer <R, S>
(b) Convert the weak pointers to S into strong pointers *
(c) Search to determine whether S is free by recursively visiting sub-objects of S and attempting to undo any strong cycles created in (b).
  for T in Sons(S) do SUICIDE(S, <S, T>) od
(d) if SRefc(S)=0 then
  for T in Sons(S) do DELETE(<S, T>) od

The SUICIDE step is rather subtle because it combines two functions:

(1) Determine whether S is free
(2) Adjust the strength and weakness of pointers to ensure that Rules 1-3 hold after removing <R, S>. It is defined as follows:

* See 'An Essential Implementation Trick' (below).
SUICIDE(Start,<R,S>)
    if <R,S> is a strong pointer then
        if S=Start then make <R,S> into a weak pointer
        else
            if SRefc(S)>1 then make <R,S> into a weak pointer
            else
                for T in Sons(S) do SUICIDE(Start,<S,T>) od
        At this point the reader should verify that the algorithm given
        works properly for the examples discussed above.

An Essential Implementation Trick

In the above description the alert reader will have noticed the operation
'make all the weak pointers to S into strong pointers'
A naive implementation:
'veisit every object, if it points to S change weak pointers to
strong'
is out of the question. The whole point of reference counting is to
avoid this kind of global search.

Instead there is a implementation trick which exactly suits our
purpose. Each pointer and each object has a bit associated with it.
When a pointer and a pointed-to object have the same bit-value the
pointer is strong; when a pointer and a pointed-to object have different
bit-values the pointer is weak.

All operations utilising the 'strength' of pointers can now be
defined in terms of single-bit changes:
'make all the weak pointers to S into strong pointers'
is implemented as
S.bit := NOT(S.bit)
(where 'S.bit' is the strength bit associated with S) and similarly
'make pointer P into a weak pointer'
is similarly implemented as
P.bit := NOT(P.bit)
(with appropriate changes to reference counts).
Discussion

Cyclic reference counting has been used in a combinator machine implemented in software. The system executes programs written in a dialect of SASL[21]. As a measure of the extra work involved, a count was made of how many storage elements were visited by SUICIDE.

\[
\text{suc} 3 \text{ WHERE suc x = x+1 ?}
\]

4
\[
\text{<10 reductions>}
\]
\[
\text{<38 new 38 freed 0 visited by 'suicide'>}
\]

\[
\text{fact 6 \text{ WHERE fact n = (n=1) -> 1; n*fact(n-1) ?}}
\]

720
\[
\text{<185 reductions>}
\]
\[
\text{<188 new 188 freed 228 visited by 'suicide'>}
\]

\[
\text{fib 6 \text{ WHERE fib n = (n=0) -> 1; (n=1) -> 1; fib(n-1)+fib(n-2) ?}}
\]

13
\[
\text{<966 reductions>}
\]
\[
\text{<724 new 724 freed 318 visited by 'suicide'>}
\]

Each 'visit' is a memory reference that would not be made with standard reference counting. Programs with no recursion have acyclic graphs and no cost is incurred. Recursive programs show that the number of elements visited is similar to the number of 'new' elements used. This is a good result showing that the amount of extra work is, in these cases, proportional to the space used.

Closer observation, using single-step execution, shows that there are two distinct components of overhead:

1. A fixed number of 'visits' per function call evaluated
2. A visit to the elements in use just before the function definition is finally erased.

Item (a) occurs when a copy of (part of) the function is peeled off to have a new actual parameter placed into it. This is a clear, recurrent pattern of execution and suitable for optimisation in the future, further reducing the overhead.

In addition, a design has been prepared for the use of cyclic reference counting in the UNIX file store (which already uses standard reference counting). This adds greater flexibility to UNIX allowing directories to be shared by making 'links' to them. Indeed, NEW, COPY and DELETE are very similar to the UNIX systems calls creat, link and unlink.

An outline proof of correctness of cyclic reference counting has been prepared[29], based on the algorithm in the Appendix. The present algorithm is a variant of that work which has not itself been proved. Further work is also needed to prove non-interference of concurrent reference counting activities to allow the graph to be shared and updated by many processors. It seems likely that a small set of indivisible operations will be sufficient for this.
The new algorithm is closely related to other algorithms by Bobrow and Hughes[16,19]. In the former scheme, objects are divided into zones, with intra-zonal pointers being treated differently from inter-zonal pointers. Intra-zonal pointers may form cycles and are not reference counted; inter-zonal pointers may not form cycles and are reference counted. Objects do not become free until the reference count of their whole zone becomes zero. The user of the algorithm is relied upon to create the zones and ensure that inter-zonal pointers are acyclic. When such pointers are manipulated, the user must be able to find the correct per-zone reference count in order to update it.

Hughes' scheme can be regarded as an automated version of the one of Bobrow just outlined. Instead of making the user define the zones, they are dynamically computed; instead of making the user find the per-zone reference count, each object contains a pointer to it. Zones are defined by the maximal cycles in the structure and are detected using an algorithm due to Tarjan[30]. When pointers are manipulated it is occasionally necessary to re-compute the zones.

The new algorithm that I have introduced above can be viewed as a further development of these two. Not only are cycles 'recognised' automatically, but also there is only an incremental search to determine whether an object is free.

There are of course many other schemes for object management but nearly all either rely on some form of global search (or global synchronisation), or are not suitable for combinator machines. One early system pioneering the notion of controlled access to heap structures is the AED system[31]. Much subsequent work on heap storage management was in LISP systems. In an interpreter for 'pure' LISP, cyclic pointers can be detected because they occur only in the code for recursive functions[18]. Because LISP recursive functions' bodies are constant and always copied rather than being shared it suffices to simply not count these cyclic references.

A neat scheme, related to our notion of strong and weak pointers but using no reference count, is described by Spector[20]. One mark bit per pointer is used to distinguish the initial pointer from subsequent copies of it; objects become free when the marked pointer is deleted. Obviously this relies on that pointer being the last to go. Christopher has described a way of adding a heap to a conventional language (FORTRAN) although it is rather expensive, requiring up to four scans of the store[17].

The key advantage of the cyclic reference counting algorithm is its strong locality. Even in a very large structure, possibly spanning many computers, it is only necessary (at most) to examine the objects reachable from some object to determine whether it is free. Such locality might also be especially useful in Virtual Memory systems, where alternative garbage collection methods require every page in the Virtual Memory to be examined, no matter how few objects are actually freed.
Appendix

Cyclic Reference Counting with Pointer Assignment

Cyclic reference counting can be adapted to more general situations where pointer values can be assigned at will - not just by combinators. This is equivalent to making NEW, COPY and DELETE available in the instruction set, not just in the micro-code. In this case it is no longer obvious whether any given COPY creates a cycle. Some new rule must be introduced concerning the parameter 'c' of COPY.

With general pointer assignment, NEW and DELETE are as before but COPY is modified to give COPY'. COPY is kept hidden in micro-code and COPY' is made visible in the instruction set. The rule relating COPY' to COPY is:

\[ \text{COPY}'(R,S) = \text{COPY}(R,S,\text{true}) \]

This assumes the worst case - that every COPY' creates a cycle - because no better information is available. The implementation of DELETE can now be modified to reclaim structures correctly although with more work than previously.

In case (iv) of DELETE(<R,S>), 'SUICIDE' is replaced by what is essentially a localised mark-scan garbage collection amongst the sub-objects of S. During this an extra bit per object is used which can take values red or green; also, a temporary reference count RRefc(R) is kept at each sub-object of S recording the number of pointers to it from red objects. It is assumed that initially all objects are green (e.g. they are given that colour by NEW). The 'mark' colours all the descendants of S red; on termination this ensures every object (R) for which

\[ \text{Refc}(R) > \text{RRefc}(R) \]

must be pointed to by a green object. Such green objects are by induction, reachable from Root and thus in use.

The 'scan' searches from S for these objects colouring them and their descendants green. At the same time the strength of pointers is adjusted where necessary.

After scanning, if S is red then it is free and can be dealt with in the usual way by recursively deleting pointers.

This general version of cyclic reference counting is described more fully elsewhere[29] accompanied by a proof. Work is continuing to compare the performance of the algorithm to others and to prove its correctness formally.
References


