Internal Report no. 88-10

A cyclic reference counting algorithm and its proof

E.J.H. Pepels
M.C.J.D. van Eekelen M.J. Plasmeijer

July 1988

This document is ported from another format. Known discrepancies are highlighted in red. If someone can solve them, this would be highly appreciated. Please send me the improved version: mailto:marko@cs.ru.nl
Marko van Eekelen, April 2014.

Department of Theoretical Computer Sciences and Computational Models
Faculty of Science
University of Nijmegen
The Netherlands

A cyclic reference counting algorithm and its proof

E.J.H. Pepels
M.C.J.D. van Eekelen M.J. Plasmeijer

Computing Science Department, University of Nijmegen, Nijmegen, The Netherlands,
and partially supported by the Dutch Parallel Reduction Machine Project, sponsored by the Dutch ministry of Science and Education

electronic mail address: marko@cs.ru.nl

ABSTRACT
Brownbridge has presented a cyclic reference counting algorithm in [Brownbridge85]. Unfortunately, this algorithm does not work correctly in all cases. Based on his ideas, we present a new algorithm, and give its correctness proof.
Like in the original algorithm, pointers have a strength, strong or weak, to administer cycles. The graph is kept connected with strong pointers, which do not form cycles; weak pointers are used to close cycles. Via procedures for pointer copying and pointer deletion this form is maintained.
It appeared that in the original algorithm after a pointer deletion in certain cases more parts of the graph have to be scanned and adjusted. A straightforward solution to this problem does not terminate. We have added a very subtle marking scheme, to guarantee on one hand the termination of the algorithm, and on the other the maintenance of the form of the graph.
The complete algorithm is very tricky, and therefore we prove it to be correct. This takes some effort. The proof is mainly based on some properties of acyclic graphs.
The algorithm is reasonably efficient for typical functional graph rewrite systems, certainly if some optimizations are done. With these optimizations, there is no time overhead compared to classical reference counting algorithms, if there are no cycles in the graph. Unfortunately the algorithm turns out to be very inefficient for some pathological cases.

TABLE OF CONTENTS

1 Introduction

We can distinguish three classes of garbage collection algorithms: mark-scan, copying and reference counting algorithms. The mark-scan and copying algorithms are most frequently used; reference counting garbage collection causes in general more overhead, since with each pointer manipulation a reference count update also has to be done.

The idea of the classic reference counting algorithm is to keep a count in each node indicating how much pointers there are to that node. When a pointer to a node is copied, the Reference Count in the node is incremented by one. When a pointer to a node is deleted, the RC in the node is decremented by one. If after deletion of a pointer the RC of a node has dropped to zero, the node in question is not reachable any more from any other node, and therefore it is garbage. Now all pointers to its sons must be deleted, and the node can be added to the free space.

With the introduction of graph rewrite systems, especially distributed ones, reference counting garbage collection gains new interest. On one hand the number of references to a node can be useful information for a graph rewrite system. If it is known, for instance, that the RC of an array is one, updates of it can be done in situ. On the other hand, just because all information needed to collect garbage is locally present, this class of algorithms is particularly suited for distributed computer systems. The information for garbage collection is automatically passed among the constituting processors via the pointer manipulations, so almost no extra overhead for garbage collection is needed.

There is one disadvantage of the classic RC algorithm: it can not recover self referencing (cyclic) structures, that have become garbage: in such a structure to each node there is at least one reference, so that the RC of all nodes is at least one, although the entire structure is garbage.

1.1. Brownbridge's algorithm

Brownbridge has presented an algorithm that can collect cyclic structures for combinator machines [Brownbridge85]. The basic ideas of his algorithm can roughly be sketched as follows:
All pointers in the graph have a strength: strong or weak. The graph is kept in such a form that the following two invariants hold:

- all nodes are reachable via a chain of strong pointers
- there are no strong cycles

In each node two reference counters are kept instead of one: one counter for the strong, and one counter for the weak references to a node.
New graphs are built according to the rules mentioned. In functional graph rewrite systems (e.g. Turner's combinator machine [Turner79], or Clean [Brus et al 87]), it is always known whether a newly built structure is cyclic or not. Thus it is very easy to maintain the rules when new graphs are constructed: all pointers are strong, except those which close a cycle; these pointers are weak. Changes to the graph are made by explicitly calling the procedures for pointer copying and pointer deletion.

The copying and deletion of pointers is done as follows:
A copy of an existing pointer is always weak. In this way no strong cycle can be formed, and the node to which the pointer is pointing stays reachable via a chain of strong pointers. (Brownbridge assumes wrongly that a copy of a pointer has the same strength as its original, but in this way strong cycles can be formed. Examples of this error are given in [Salkild85].) Note that the assumption that weak pointers close cycles does not hold any more. Weak pointers can occur everywhere. But the two invariants are still valid.
As long as there stay strong pointers to a node we can delete arbitrary pointers (weak and strong) to it. When the last strong pointer to a node is deleted, and there are no other pointers to it, the node is garbage and can be deleted.

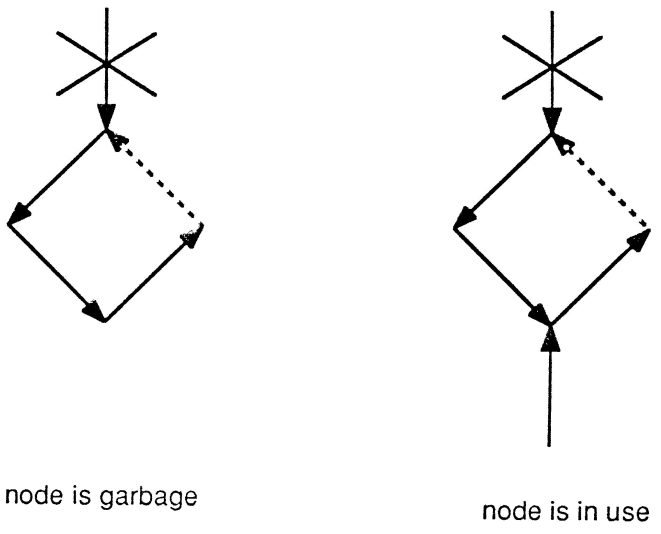

figure 1.1

When the last strong pointer to a node to which there are still weak pointers, is deleted, we have exactly the case in which the classic version of the RC algorithm fails. A pointer to a (possible) cyclic structure is deleted, and we do not know whether there are still other references to the structure. So some special action has to be undertaken to detect whether the node is garbage or not, and if not, to adjust the graph such that the invariants hold again. Possible situations that can occur before this special action are given in figure 1.1. Weak pointers are dashed. Deleted pointers are marked with a cross.
The following special action has to be undertaken: all pointers to the node are made strong. Thus, if the node is non-garbage, it is reachable via strong pointers again. Since in this way strong cycles can be formed, the structure reachable from the node must be traversed to remove these strong cycles. During this traversion (along strong pointers, because they form the strong cycles) two things happen: external references are searched for, and the possibly generated strong cycles are broken up by making some pointers weak. These two goals can be reached in one way: if an external entry is detected, the possible strong cycle is broken up by making the pointer to this entry weak.
If no external reference is detected, the node is garbage and can be deleted. (see figure 1.2).

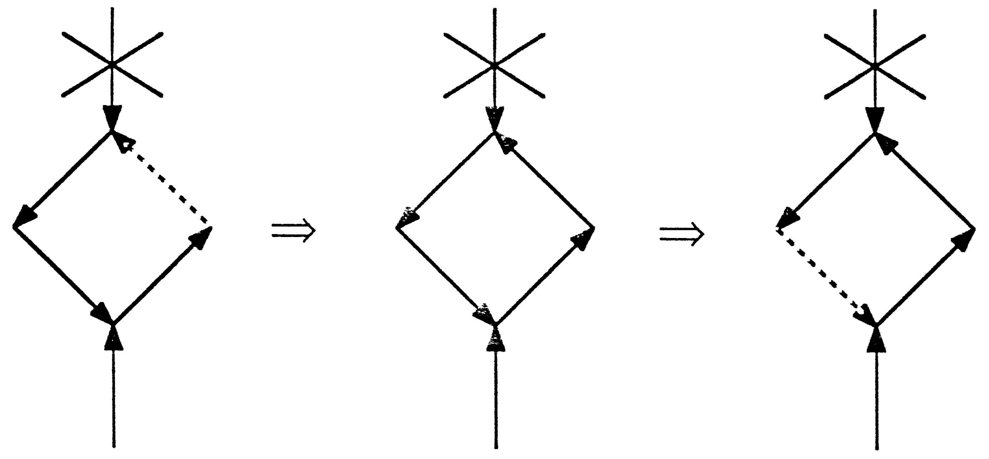
figure 1.2

In the algorithm all weak pointers to a node are made strong. This is efficiently implemented by a trick: in each pointer and in each node a bit is reserved. When the bit values of a pointer and the node it is pointing to are the same, the pointer is strong; otherwise the pointer is weak. In this way, it is not only possible to change the strength of a pointer of which the node is the starting point, but also the strength of all pointers to that node can be changed: when the bit in the node is swapped, all pointers to that node change strength.

1.2. *Flaws and improvements*

Although these basic ideas are very good, the concrete algorithm is not entirely correct. This is caused by the fact that, while the rules are not violated, external references can be weak. In the original algorithm of Brownbridge, weak pointers to nodes, which are encountered during the graph traversion, are ignored. In this way cyclic structures are incorrectly considered to be garbage. This is showed in figure 1.3.

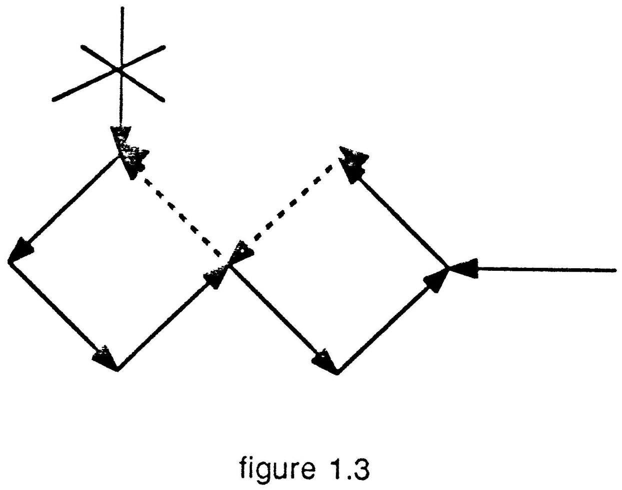
figure 1.3

The node to which the last strong pointer is deleted, is still reachable via a chain of pointers, but some of the pointers are weak. When the structure is traversed, this weak pointer is ignored, so it seems that there are no external references to it, and the node is considered to be garbage.

Salkild [Salkild85] has found this error, and proposed an improvement: if a node is encountered, to which there are weak pointers and only one strong pointer (the pointer along which the traversal reaches the node), the (bit in the) node is swapped, and a same search is started from this node. Unfortunately, with this improvement the algorithm gives in certain cases rise to an infinite number of such searches [Salkild85]:

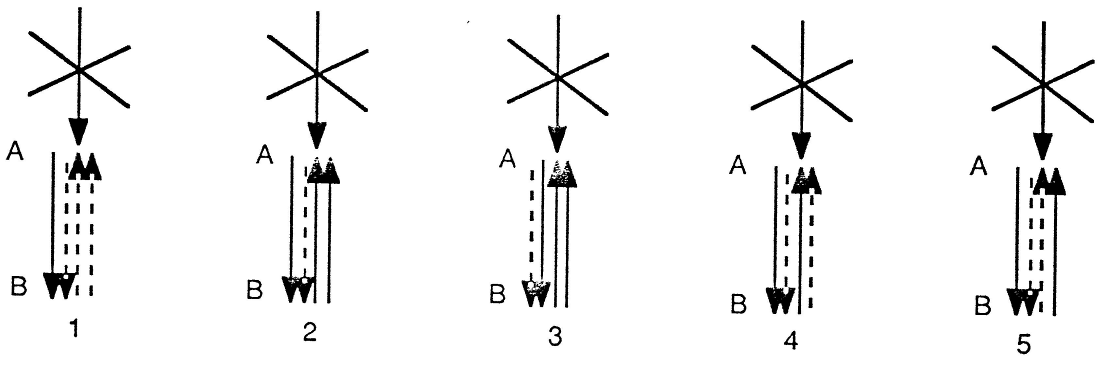

figure 1.4

The strong pointer to node A is deleted. Since the SRC of A is now zero, while the WRC of A is 2, node A is swapped and its subgraph is traversed (situation 1,2). Node B is encountered; its SRC is one, and its WRC is one too. So node B is swapped (situation 3), and its subgraph is traversed. Node A will be encountered twice. The first time its SRC is two, and the pointer is made weak (situation 4). The second time its SRC is one, as is its WRC. So node A is swapped and its subgraph is traversed (situation 5). In this traversal node B is encountered again with SRC one and WRC one. The reader can verify that from now on situation 4 and 5 will indefinitely succeed each other.

We felt that it is possible to change the algorithm such that it terminates in all cases. Therefore a marking scheme was developed. However, it was very difficult to guarantee on one hand the termination of the algorithm, and on the other hand the maintenance of the invariants. It appeared that two kinds of marks are needed to make the algorithm work correctly: one to prevent an infinite number of searches for external references and strong cycles, and one to guarantee the termination of such a search. These two kinds of marks can be implemented by one counter in a node.
After these repairs, the algorithm had become so tricky, that we could only believe that it really worked, if a proof was given. Although this turned out to be very hard, we finally managed to do it.

The rest of the paper is organized as follows: In chapter 2 the correct algorithm is presented, and a thorough intuitive description of it is given. In chapter 3 we present the proof. In section 3.1 we give some formal definitions about graphs. In section 3.2 we will prove that after pointer copying the two invariants (strong pointers do not form cycles, and all nodes are reachable via a chain of strong pointers) hold. In section 3.3 we prove the same for pointer deletion. In chapter 4 we discuss some optimizations; furthermore we give some topics for further research and we draw some conclusions.

2 The algorithm

2.1. *The algorithm in pseudo-Modula2*

Below the adapted algorithm and the datastructures used are presented in pseudo-Modula2.
An intuitive description of the algorithm is given in the next section.

(* In each pointer there is a bit which indicates together with the bit in the node its strength *)

```
TYPE NODEPTR =
  RECORD
    Ptr:            POINTER TO NODE;
    Strength:       BIT
END NODEPTR.
```

(* In each node there is a bit which indicates together with the bit in the pointer the strength of the pointer. Further there are two reference counters; which counter is used for the WRC and which for the SRC depends on the value of the bit Strength in the node. The right value of the RC's is determined by the access procedures WRC and SRC. One counter is reserved for the marking that is done during the algorithm. The number of pointers per node is variable. *)

```
TYPE NODE =
  RECORD
  Strength:       BIT;
  RC1:            CARDINAL;
  RC2:            CARDINAL;
  Mark:           CARDINAL;
  NumberOfSons:   CARDINAL;
  Sons:           ARRAY [1..NumberOfSons] OF NODEPTR;
  Info:           (* further information *)
END NODE.
```

(* a global variable for the marking scheme *)

```
VAR depth: CARDINAL
```

(* the procedure for pointer copying *)

```
CopyPtr (ptr): NODEPTR =
VAR newptr: NODEPTR;
  newptr := ptr;
  IF IsStrong (newptr)
  THEN    MakeWeak (newptr)
  ELSE    INC (WRC (ptr^))
  END;
RETURN newptr.
```

(* We assume that the procedures MakeWeak and MakeStrong adjust the WRC and the SRC in the right way. *)

```
DeletePtr (ptr) =
 VAR startnode: NODE;
 startnode := ptr^;
 IF IsWeak (ptr)
 THEN    DEC (WRC (startnode))
 ELSE    DEC (SRC (startnode));
    IF SRC (startnode) = 0 AND WRC (startnode) = 0
    THEN DeleteNode (startnode)
    ELSIF SRC (startnode) = 0 AND WRC (startnode) > 0
    THEN depth := 1;
         AdjustPresumableCycle (startnode);
         IF SRC (startnode) = 0
         THEN DeleteNode (startnode)
         ELSE (* SKIP *)
         END
    ELSE (* SKIP *)
    END
 END.

AdjustPresumableCycle (topnode) =
 INC (depth);
 Swap (topnode);
 MarkAsTopnode (topnode);
 AdjustGraphBetween (topnode, topnode);
 UnmarkAsTopnode (topnode)
 DEC (depth).

Swap (node) =
 node.Strong := NOT node.Strong.

AdjustGraphBetween (node, topnode) =
 FOR ptr IN Sons (node)
 DO IF IsStrong (ptr)
    THEN nextnode := ptr^;
         IF nextnode = topnode
         THEN MakeWeak (ptr)
         ELSIF NotMarkedAsTopnode (nextnode)
         THEN IF SRC (nextnode) ≥ 2
              THEN MakeWeak (ptr)
              ELSIF WRC (nextnode) ≥ 1
              THEN AdjustPresumableCycle (nextnode);
                   IF SRC (nextnode) = 0
                   THEN MakeStrong (ptr);
                        MarkAsTopnode (nextnode);
                        AdjustGraphBetween (nextnode, topnode);
                        UnmarkAsTopnode (nextnode)
                   ELSE (* SKIP *)
                   END
              ELSE  AdjustGraphBetween (nextnode, topnode)
              END
        ELSIF NotMarkedAsVisited (nextnode)
        THEN  MarkAsVisited (nextnode);
              AdjustGraphBetween (nextnode, topnode);
              UnmarkAsVisited (nextnode)
        ELSE  (* SKIP *)
        END
    ELSE (* SKIP *)
    END (* IF *)
 END (* DO *).

(* some marking procedures *)

MarkAsTopnode (node) =
    node.Mark := 1.

UnmarkAsTopnode (node) =
    node.Mark := 0.

NotMarkedAsTopnode (node) =
    node.Mark = 0.

MarkAsVisited (node) =
    node.Mark := depth.

UnmarkAsVisited (node) =
    node.Mark := 1.

NotMarkedAsVisited (node) =
    node.Mark ≠ depth.

(* the procedure for deletion of nodes *)

DeleteNode (node) =
    IF NotMarkedAsGarbage (node)
    THEN MarkAsGarbage (node);
        FOR ptr IN Sons (node)
        DO DeletePtr (ptr)
        END
    ELSE (* SKIP *)
    END;
    AddNodeToFreelist (node).

MarkAsGarbage (node) =
    node.Mark := 1.

NotMarkedAsGarbage (node) =
    node.Mark ≠ 1.
```

2.2. *Intuitive description of the algorithm*

We repeat that the procedures CopyPtr, DeletePtr and DeleteNode, must work in such a way that outside of them the invariants are guaranteed:

- there are no strong cycles.
- all non-garbage nodes are reachable via a chain of strong pointers. (We assume that there is one node in the graph, that is non-garbage, but to which there are no pointers. All other non-garbage nodes are reachable from this node. We will call this node the program root.)

If a pointer is copied (CopyPtr, lines 16-24), the copy of the existing pointer can be made weak safely, since the node, to which the pointer is pointing, stays reachable via a chain of strong pointers, and no strong cycle can possibly be made by adding a weak pointer.

After a pointer has been deleted (DeletePtr, lines 25-42), care must be taken that all non-garbage nodes are still reachable via a chain of strong pointers. This is no problem when a weak pointer is deleted: the WRC of the node to which the pointer is pointing has to be decremented by one (line 29). If a strong pointer is deleted, and it is the last pointer to a node, then clearly this node is garbage, and can be deleted (lines 31-32). If a strong pointer is deleted, and there are other strong pointers to the node in question, only its SRC has to be decremented by one (lines 29 and 39).

But if the last strong pointer to a node is deleted, and there are still weak pointers to the node, we are in big trouble. It is not sure whether the node is really garbage (since there are no strong pointers to it), or whether the node is still reachable via weak pointers.

Examples are given in figure 2.1.

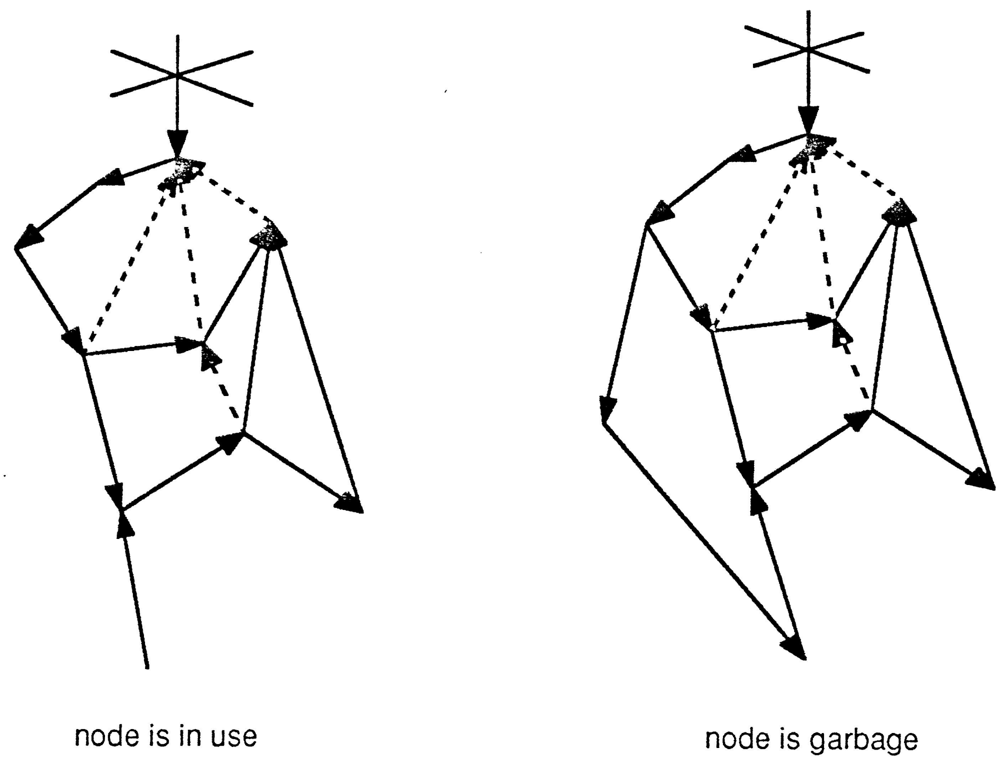

figure 2.1

Alas, DeletePtr has not a broad sight over the total graph: it has only some local information. Yet some adequate action has to be undertaken to figure out whether the

node is garbage or not, and, if not, to adjust the graph such that the invariants hold again (lines 33 to 39).
This adjustment is done by AdjustPresumableCycle. The node is swapped by APC because, if it is reachable from the program root, it must be reachable via strong pointers after DeletePtr has done its work. Making all pointers to it strong is a first step in the right direction.
Further on in the algorithm it will appear that APC calls itself recursively on other nodes. To prevent an infinite number of adjustments done by APC, starting from the same node, startnode is marked as topnode (line 46).
By swapping the node, probably a strong cycle has been generated, and it is still not clear whether the node is garbage or not. So a search through the graph is started by calling AdjustGraphBetween (line 48). The aim of a search like this is twofold:

- to detect a path from a node outside the subgraph of topnode (i.e. all nodes which are reachable from topnode) to topnode, and make it entirely strong.
- to break up the strong cycle which is possibly generated by the swap.

AGB traverses the graph via strong pointers. Weak pointers can safely be ignored since they do not form strong cycles (line 54).
If AGB arrives at topnode again (lines 56 and 57), there was surely a strong cycle, and the only possibility is to make this closing pointer of the strong cycle weak. A typical example is given in figure 2.2.

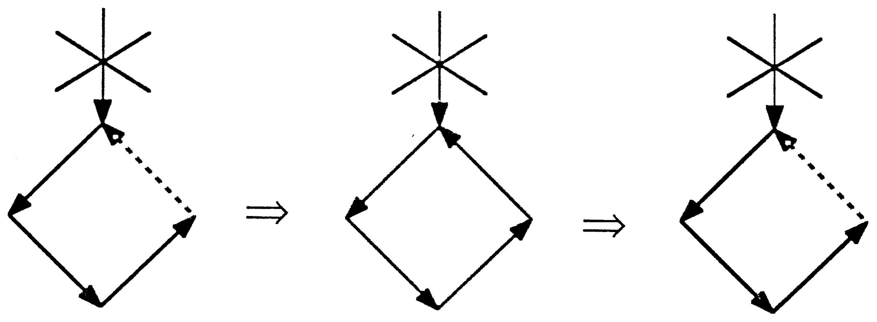

figure 2.2

If a node is encountered, that is not marked as topnode (line 58), then there are several possibilities. If the SRC of the node is at least two (line 59), the pointer which is currently treated, can be made weak, since in this way a strong cycle that might exist over this pointer is removed, and the node to which it is pointing has still a SRC of at least one. An example is given in figure 2.3.

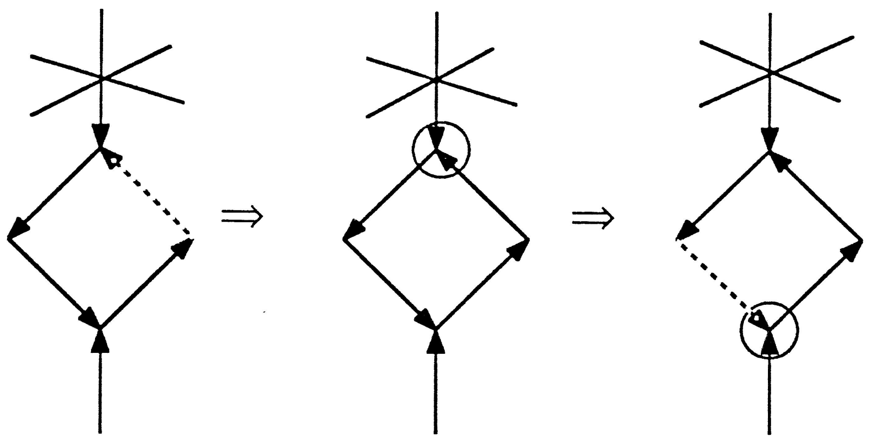

figure 2.3

If there are weak pointers to the node, and there is only one strong pointer to it (the current pointer) (line 60), then we have some trouble: it could be that one of the weak pointers is the only external reference to the subgraph (see figure 2.4).

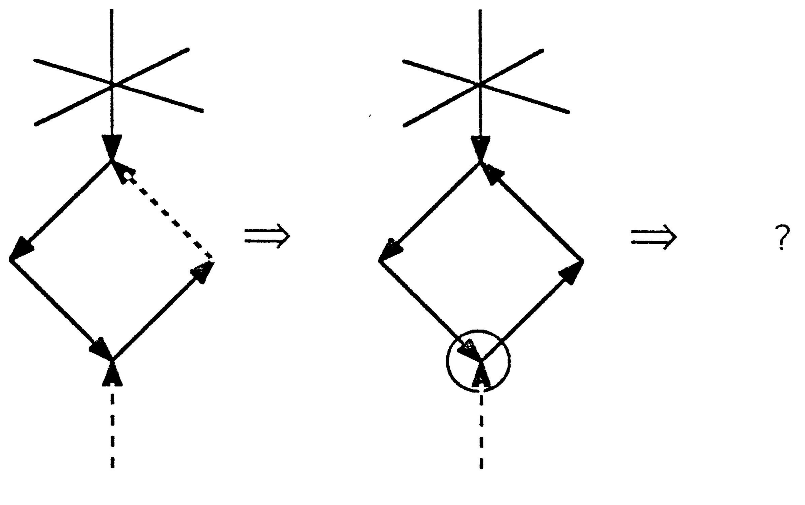

figure 2.4

Since we want the path to startnode to be strong after the algorithm is terminated, and this pointer could be part of the only path to startnode, the only right thing to do is to swap the node to make this external pointer strong. Subsequently a search with this node as topnode has to be started, for two reasons:

- a strong cycle might have been generated by the swap.
- in the subgraph of this node there can be strong or weak external references too.

This treatment of the graph (swapping and searching) is exactly the same as that of APC. So APC is called on this node (line 62).
When APC is terminated, all pointers to the node can have been made weak. Then this node will surely not be reachable via strong pointers afterwards. Apparently, all

pointers to this node are contained in a cycle, so this is not the right place to make a pointer of the 'main' cycle weak. So the entrance pointer (contained in the main cycle) is made strong again, and the node is kept marked, to indicate that from this node on already a search has been done. Now the search for external references and strong cycles is continued in the 'main' cycle (lines 63 to 69, see also figure 2.5).

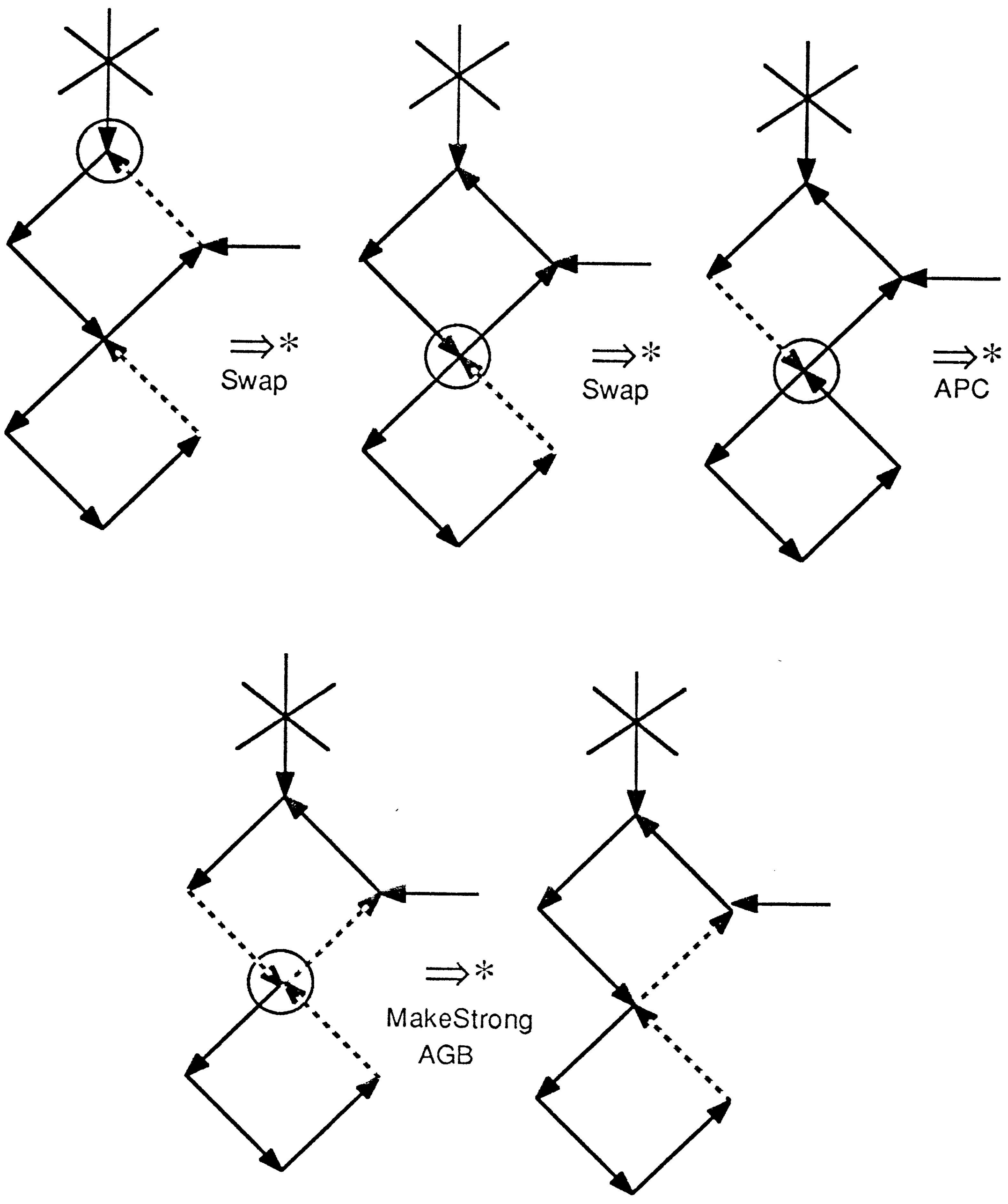

figure 2.5

If there are no weak pointers to the node and the node is unmarked and its SRC is one (line 70), we just proceed traversing the graph.
If the node is already marked as topnode (lines 73 and further), however, the pointer may not be weakened, even if its SRC is at least two. It is possible that the strong pointer which is currently treated, is a part of the path from the program root to the node. The aim of the algorithm is to make this path entirely strong, so that the pointer

may not be weakened. This is demonstrated in figure 2.6.

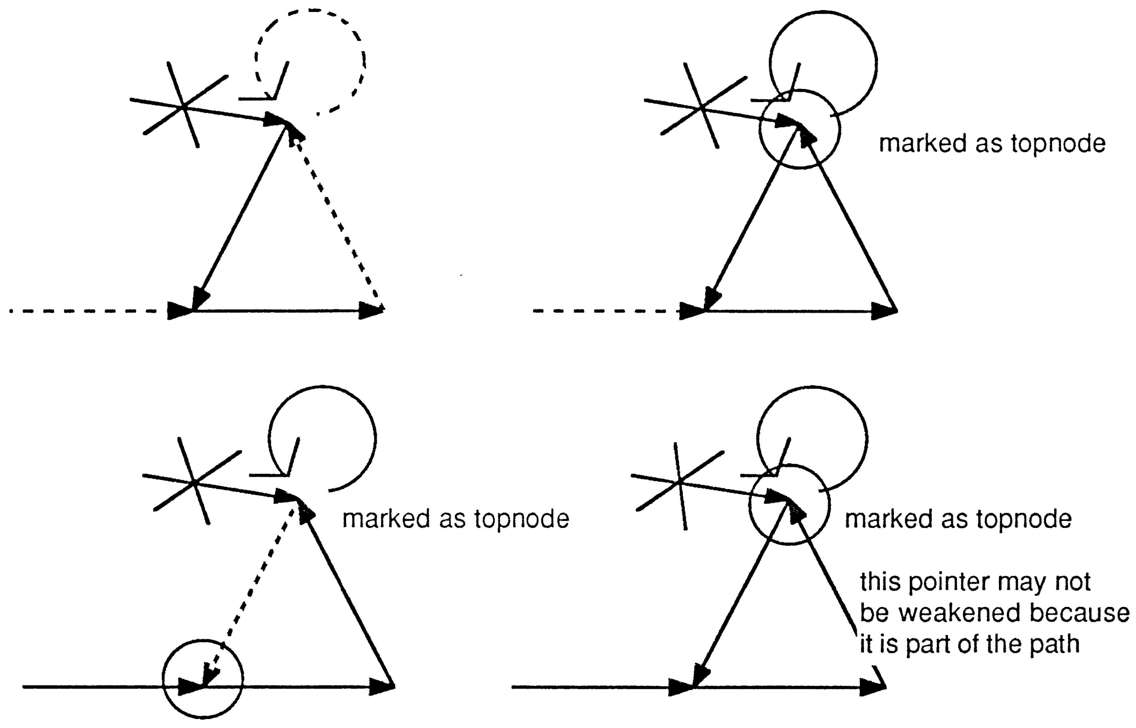
marked as topnode
marked as topnode
marked as topnode
this pointer may not be weakened because it is part of the path

figure 2.6

Since there still may be strong cycles over the topnode of the current incarnation of AGB, AGB may not stop traversing the graph at this moment. It must go on, looking for strong cycles, and weakening pointers to topnode on them (line 74).

As a conclusion we can say that pointers to a node, marked as topnode, are only weakened in the incarnation of APC on the node on which APC is called (the topnode of the incarnation of APC). To understand the algorithm it is important to keep in this mind.

Because the algorithm must go on looking for strong cycles, a more careful administration must be done to guarantee termination of the algorithm. This is necessary because pointers to nodes, marked as topnode, have been made strong, thereby possibly generating strong cycles. If AGB would traverse the graph without administering any such nodes, it is possible that one of such cycles is traversed indefinitely (see figure 2.7).

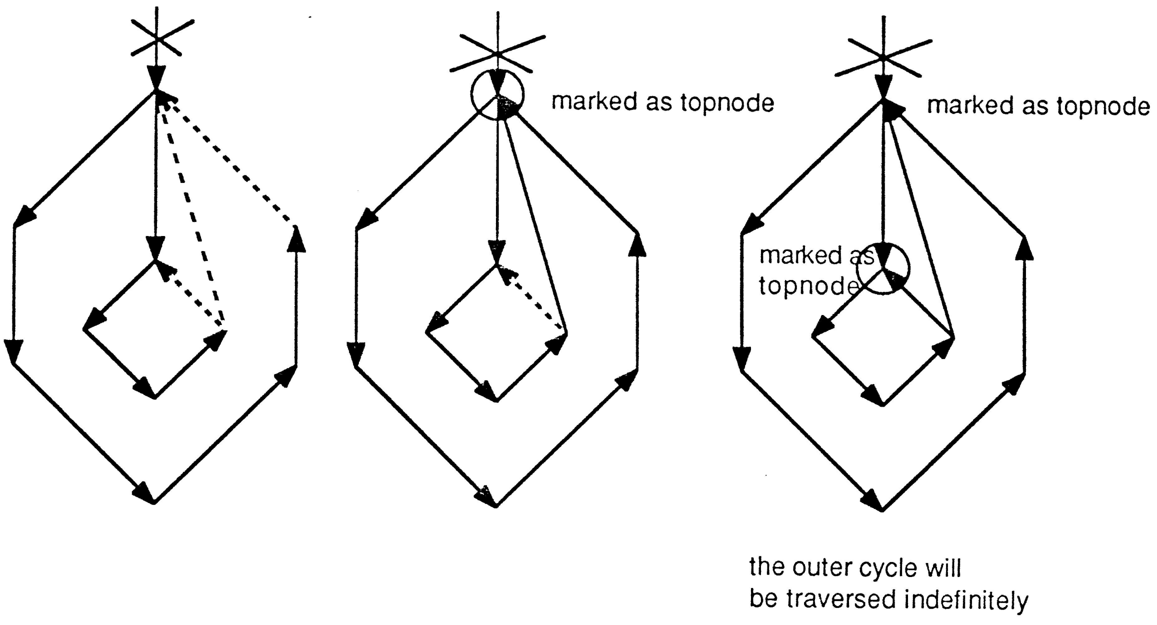

figure 2.7

Since cycles are only generated over nodes which are marked as topnode, it suffices to administer in each incarnation of APC the nodes, which are already marked as topnode, also as visited. Now when a node is encountered which is administered as already visited in this incarnation, it is clear that a strong cycle is detected, and the graph traversion done by AGB can be stopped (line 76). The reader can verify this in the example above.

At first sight it seems that the administration of all visited nodes per incarnation forces us to reserve a certain amount of memory per incarnation, which means that at least a large amount of memory must be reserved for this marking. Fortunately, it appeared that it was possible to implement the topnode marking and the visitation marking of the nodes by one single counter in each node.
This is done as follows: the counter in nodes which are not marked is zero. Marking as topnode is done by setting the counter to one. Unmarking as topnode is setting the counter to zero again. Marking as visited is done by setting the counter to the current 'depth' of the incarnation of APC. We initialize the the value of 'depth' at one, which means that the first incarnation of APC has already depth two. Therefore there is no interference with the topnode marking. Unmarking as visited is done by setting the counter to one (= marked as topnode, unmarked as visited).

Now we have the problem that a visit mark on a node, done by a certain incarnation of APC, can be removed when a deeper incarnation marks the same node as visited, and unmarks it afterwards. In this case the node becomes administered as unmarked as visited, while it is visited by at least one incarnation. The termination of the algorithm is yet guaranteed for the following reasons:
First, there will be a finite number of incarnations of APC nested in each other, since all these incarnations have their topnodes marked, and on marked nodes no new incarnation is started.
Second, one incarnation of APC will fail to terminate if it traverses a strong cycle indefinitely.
But we can simply deduce that the deepest incarnation of APC will terminate. This is

because all nodes which are possibly part of a strong cycle (the topnodes) are marked as visited in this incarnation will not become unmarked by deeper incarnations (because there aren't any).
But then we know that the deepest minus one incarnations will terminate too. This can be seen as follows. Consider the deeper calls to APC which are done in a deepest minus one incarnation of APC. The number of this calls must be finite, since they only can happen in a call to AGB on a son of the node on which the deepest minus one incarnation is called, and the number of sons of a node is finite. So the visit marks done by the deepest minus one incarnation can only be removed a finite number of times. After all these calls are terminated, no mark will be removed any more, and the deepest minus one incarnation will terminate.
In the same way we see that the deepest minus two incarnations will terminate, also the deepest minus three incarnations, and so on. So all incarnations of APC will terminate.

After APC is called on startnode is terminated, we have two possibilities:

- The SRC of startnode is at least one.
  In this case there were external references to startnode, that is to say startnode is still in use.
- The SRC of startnode is zero.
  In this case all pointers to startnode were part of cycles without external references over startnode, in other words startnode is garbage. So startnode can be deleted.

When, finally, on startnode DeleteNode is called, then first it becomes marked as already known garbage. This is necessary because there are possibly cycles in the graph. When garbage nodes would try to delete their sons without marking them, this would give rise to a cycle of garbage nodes, which try indefinitely to delete each other. The mark in a node can be used to mark a node as garbage. Although the procedure DeletePtr, which uses the mark too, is called while there are garbage nodes in the graph, these two kinds of marking do not interfere. This is because APC traverses the graph via strong pointers and to garbage nodes no or only weak pointers are pointing.
After the marking all pointers in the node are deleted, after which the node itself can be added to the freelist (or otherwise be administered as free).

3 Proof of the algorithm

The intuitive description of the algorithm seems plausible, however the algorithm is rather tricky, and a proof is the only way to show that it really works. In the following sections a proof is given.
We are going to show that after a pointer is copied or deleted, the following invariants still hold:
- there are no strong cycles.
- all non-garbage nodes are strongly reachable from the program root.

And also we are going to prove that after the algorithm has terminated the following equivalence holds:
- 'startnode' is garbage ⇔ the SRC of 'startnode' is zero.

The rest of this chapter is organized as follows:
In section 3.1 we give some definitions about graphs, such that it is clear for anyone what we are talking about. Furthermore we give some useful properties of graphs.
In section 3.2 we prove that the invariants still hold after pointer copying.
In section 3.3 we prove the same for pointer deletion. This last proof turns out to be very hard, and it is split up into two parts: in section 3.3.1 we prove that the procedures DeletePtr and DeleteNode are correct, provided that the procedure AdjustPresumableCycle is correct.
This is proved in section 3.3.2. First we prove that the marking mechanism ensures in all cases the termination of APC.
In the proof we use the property of graphs without cycles: when the RC of all nodes (except the root) is at least one, they are all reachable from the root. We prove that after APC has terminated, there are no strong cycles in the graph. Subsequently it is proved that the SRC of all nodes, except the program root and 'startnode' is (still) at least one after APC is finished. Using this properties of APC, we prove finally that if 'startnode' is non-garbage, its SRC is at least one too. With the property we know then that all nodes, including 'startnode' must be strongly reachable from the program root.

3.1. Definitions and properties of graphs

A *graph* is a pair <X, U> where

(1) X is a set $\{x_1, ...x_n\}$ of elements called nodes. Nodes can have a *mark*, which is a non-negative number.

(2) U is a multiset $(u_1, ..., u_m)$ of elements of the Cartesian product (X, X), called pointers.
Pointers have a colour; the colour of a pointer is either *strong* or *weak*.

For a pointer u = (x, y), node x is called the *source* of the pointer, and node y is called its *endpoint*. We also say that pointer u is pointing to node y.

The *strong reference count (SRC)* of a node is the number of strong pointers pointing to it. The *weak reference count (WRC)* of a node is the number of weak pointers pointing to it. The *reference count (RC)* of a node is the total number of pointers pointing to it.

A node y is a *son of* node x if there is a pointer u = (x, y). A node y is a *strong son of* node x if there is a strong pointer u = (x, y).

A *path* is a sequence of pointers u = (u_1, ...,u_i , ..., u_z), $z \geq 1$, where all ui are pointers of a graph G, such that for all pointers u_i and u_{i+1} the endpoint of ui is the source of u_{i+1}. A strong path is a path consisting of strong pointers only.

The pointers ui are said to be *contained in* the path u. The number of pointers z in the path is called the *length* of the path.
A node x is *contained in* a path u if it is the source or endpoint of a pointer which is contained in u.
Node x_1 is called the *source* of the path, and node y_z is called the endpoint of the path. A path from x to y is a path of which the source is x, and the endpoint is y.

A node y is said to be *reachable* from node x if there exists a path from x to y. A node y is said to be *strongly reachable* from node x if there exists a strong path from x to y.

A *cycle* is a path such that
 (1) there is no node to which two or more pointers of the path are pointing
 (2) the source and the endpoint of the path are the same node

A *strong cycle* is a cycle of which all pointers are strong.

An *acyclic* path is a path not containing cycles.
If there is a path P from a node x to a node y, and y is contained in a cycle C, such that no pointer, which is contained in P, is contained in C, we call y the entrynode of P in C. The pointer in the path P which is pointing to y, is called the entrypointer of P in C.

The *subgraph* G' in a graph G = <X, U> of a node t is a pair <X', U'> such that
 (1) X' = the set of nodes in G reachable from t, and t itself
 (2) U' = the set of pointers in U, which have both their source and endpoint in X'.

The *strong subgraph* G' in a graph G = <X, U> of a node t is a pair <X', U'> such that
 (1) X' = the set of nodes in G strongly reachable from t, and t itself
 (2) U' = the set of pointers in U, which have both their source and endpoint in X'.

In each graph there is a special node which we will call the *program root*.
The program root is not reachable from any node in the graph. It will be convenient to say that the program root is reachable from itself, though this is not strictly true. The RC of the program root is zero.
A node x, that is not the program root, is said to be garbage if there is no path from the program root to x. A node x is said to be non-garbage if there is no path from the program root to x, or if x is the program root.

Consider a graph. Then the following property and corollary are valid:

Property 1
The RC of all nodes, except {x_1, ...x_n}, is at least one, and the RC of {x_1, ...x_n} is zero, and there are no cycles in the graph $\Leftrightarrow$
 all nodes are reachable from at least one of the nodes {x_1, ...x_n}, and

no node x_i is reachable from a node x_j.

Corollary 1
For graphs without strong cycles the following equivalences hold:
1. the SRC of all nodes, except the program root, is at least one ⇔
   all nodes are strongly reachable from the program root
2. the SRC of all nodes, except the program root and node x, is at least one, and the SRC of node x is zero ⇔
   all nodes are strongly reachable from the program root or from x, and x is not strongly reachable from the program root and vice versa.

3.2. *Proof of pointer copying*

The proof that CopyPtr works correctly is obvious. Before CopyPtr is called, we have the invariants:
- there are no strong cycles
- all non-garbage nodes are strongly reachable from the program root

CopyPtr *adds* a weak pointer, so it is impossible that these invariants are violated.

3.3. *Proof of pointer deletion*

In this section we will prove the correctness of pointer deletion. With pointer deletion also node deletion is involved. Therefore we will prove the correctness of pointer and node deletion together in the first part of this section. Thereafter we will prove the correctness of the procedure AdjustPresumableCycle, which does the proper adjustment of the graph in case the last strong pointer to a possibly cyclic structure is removed.

3.3.1. Proof of the procedures DeletePtr and DeleteNode

Below in the procedure DeletePtr assertions are marked, which we are going to prove. Outside of DeleteNode, we start from the invariants
- there are no strong cycles
- all non-garbage nodes are strongly reachable from the program root

and prove that when DeletePtr is finished they still hold. DeleteNode is only called in DeletePtr, so it suffices to show that DeleteNode works correctly in this case.
Furthermore we prove that DeleteNode is called only on nodes which are garbage. We just mention here that we don't have explicitly to prove that all garbage nodes are deleted; nodes can only become garbage via pointer deletion, and with each pointer deletion DeletePtr is called, and if necessary DeleteNode.
But first an additional definition:

Definition
When DeletePtr is called with argument a pointer which is pointing to node x, we call x the startnode of this incarnation of DeletePtr.

Abbreviations
PR is the program root.
SN is the startnode of the current incarnation.
O is the set of all topnodes of existing incarnations of DeletePtr, except the current

incarnation (all old startnodes).

The proof of the procedures is given below. With assertions we prove the invariants. The second invariant proved differs slightly from the proper one to prove. Actually we prove that after DeletePtr is finished all non-garbage nodes are strongly reachable from the program root or from the startnodes of existing incarnations of DeletePtr. This implies automatically that afterwards all non-garbage nodes are strongly reachable from the program root, since then no incarnation of DeletePtr exists any more.

```
        {there are no strong cycles} (A1a) ∧
        {all non-garbage nodes are strongly reachable from PR, SN or O} (A1b)
    1.  DeletePtr (ptr) =
    2.  VAR startnode: NODE;
    3.  startnode := ptr^;
    4.  IF IsWeak (ptr)
    5.  THEN DEC (WRC (startnode))
                {there are no strong cycles} (A2a) ∧
                {all non-garbage nodes are strongly reachable from PR or O} (A2b)
    6.  ELSE  DEC (SRC (startnode));
    7.          IF SRC (startnode) = 0 AND WRC (startnode) = 0
                THEN  {there are no strong cycles} (A3a) ∧
                        {startnode is garbage} (A3b) ∧
                        {all non-garbage nodes are strongly reachable from PR, SN or
                O} (A3c)
    8.  DeleteNode (startnode)
                        {there are no strong cycles} (A4a) ∧
                        {all non-garbage nodes are strongly reachable from PR or O}
(A4b)
    9.  ELSIF SRC (startnode) = 0 AND WRC (startnode) > 0
    10. THEN depth := 1;
                {there are no strong cycles} (A5a) ∧
                {all non-garbage nodes, except SN, are strongly reachable
                                from PR, SN or O} (A5b) ∧
    11.         {WRC (startnode) ≥ 1} (A5c)
    12.         AdjustPresumableCycle (startnode);
                {there are no strong cycles} (A6a) ∧
                {all non-garbage nodes, except SN, are strongly reachable
                        from PR, SN or O} (A6b) ∧
                {the SRC of startnode is at least one Û startnode is non-garbage }
(A6c)
    13.         IF SRC (startnode) = 0
    14.         THEN  {there are no strong cycles} (A7a) ∧
                        {all non-garbage nodes are strongly reachable from PR, SN or
O}                                                              (A7b) ∧
                        {startnode is garbage} (A7c)
    15.                 DeleteNode (startnode)
                        {there are no strong cycles} (A8a) ∧
                        {all non-garbage nodes are strongly reachable from PR or O}
```

```
(A8b)
16.         ELSE  (* SKIP *)
                  {there are no strong cycles} (A9a) ∧
                  {all non-garbage nodes are strongly reachable from PR or O} (A9b)
17.         END
            {there are no strong cycles} (A10a) ∧
            {all non-garbage nodes are strongly reachable from PR or O} (A10b)
18. ELSE  (* SKIP *)
    {there are no strong cycles} (A11a) ∧
    {all non-garbage nodes are strongly reachable from PR or O} (A11b)
19. END
    {there are no strong cycles} (A12a) ∧
    {all non-garbage nodes are strongly reachable from PR or O} (A12b)
20. END.
    {there are no strong cycles} (A13a) ∧
    {all non-garbage nodes are strongly reachable from PR or O} (A13b)


    {there are no strong cycles} (A14a) ∧
    {all non-garbage nodes are strongly reachable from PR, SN or O} (A14b) ∧
    {node is garbage} (A14c)
21. DeleteNode (node: NODE) =
22. IF NotMarkedAsGarbage (node)
23. THEN MarkAsGarbage (node);
24.         FOR ptr IN SonsOf (node)
25.         DO    {there are no strong cycles} (A15a) ∧
                  {all non-garbage nodes are strongly reachable from PR, SN' or O' where SN' = ptr^ and O' = O È {SN}} (A15b)
26.               DeletePtr (ptr)
                  {there are no strong cycles} (A16a) ∧
                  {all non-garbage nodes are strongly reachable from PR or O' where O' = O È {SN}} (A16b)
27.         END;
            {there are no strong cycles} (A17a) ∧
            {all non-garbage nodes are strongly reachable from PR or O} (A17b)
28.         AddNodeToFreelist (node)
            {there are no strong cycles} (A18a) ∧
            {all non-garbage nodes are strongly reachable from PR or O} (A18b)
29. ELSE (* SKIP *)
30. END.
    {there are no strong cycles} (A19a) ∧
    {all non-garbage nodes are strongly reachable from PR or O} (A19b)
```

Proof of the assertions

(A1) -

(A1a) and (A1b) are the preconditions of DeletePtr in DeleteNode. Furthermore (A1a) and the invariant 'all non-garbage nodes are reachable from PR' hold when a non-nested call to DeletePtr is done, so in this case (A1a) and (A1b) hold too.

(A2) -

A weak pointer is deleted: it is impossible that there are strong cycles generated (A2a). There are no strong pointers deleted or weakened, so with (A1b) we know (A2b).
(A3) -
There is no strong cycle formed (A3a); startnode is garbage since there is no pointer to it (A3b); (A3c) follows from (A1b).
(A4) -
These are the postconditions of DeleteNode.
(A5) -
There is no strong cycle formed (A5a); only a strong pointer to startnode has been deleted, so all other nodes stay strongly reachable from PR, SN or O (A5b); from the IF-THEN-ELSE construct we know that WRC (startnode) 3 1.
(A6) -
These are the postconditions of AdjustPresumableCycle; they are proved in section 3.4.2.
(A7) -
These invariants follow from (A6) and from the fact that SRC (startnode) = 0.
(A8) -
These are the postconditions of DeleteNode.
(A9) -
(A9a) follows from (A1a); we know SRC (startnode) 3 1 ((A6) and the properties of the IF-THEN-ELSE construct); now from corollary 1 we know that startnode is reachable from the program root; so we know with (A6) that all non-garbage nodes are reachable from PR or O (A9b).
(A10) -
This follows directly from (A8) and (A9).
(A11) -
(A11a) follows from (A1a); From lines 7 and 9 it follows that in this case the SRC of startnode is at least one. Together with corollary 1 this yields (A11b).
(A12) -
This follows from (A2), (A4), (A10) and (A11).
(A13) -
This follows from (A12).
(A14) -
These are the preconditions of DeleteNode in DeletePtr.
(A15) -
(A15a) follows from (A14a); from (A14b) we know that all non-garbage nodes are reachable from PR, SN or O, so all non-garbage nodes are surely reachable from PR, SN, O or ptr^.
(A16) -
These are the postconditions of DeletePtr.
(A17) -
(A17a) follows from (A16a); all pointers with source node have been deleted, so there are no nodes reachable any more from node, so with (A16b) we know (A17b).
(A18) -
This follows from (A17).
(A19) -
This follows from (A18).

Note that the Mark in the nodes is used both for marking in APC and in DeleteNode. These two kinds of marking do not interfere since only nodes to which there are no or only weak pointers are pointing, and APC traverses the graph via strong pointers.

3.3.2. Proof of the procedure AdjustPresumableCycle

As a consequence of section 3.3.1 we have to prove that, starting from the preconditions
- there are no strong cycles
- all non-garbage nodes, except startnode, is at least one
- WRC (startnode) ≥ 1

that after APC is finished the following postconditions hold:
- there are no strong cycles.
- all non-garbage nodes, except SN, are strongly reachable from PR, SN or O
- startnode is non-garbage Û the SRC of startnode is at least one

But with the corollary in mind, it suffices to show that:
- there are no strong cycles.
- the SRC of all nodes, except startnode and the program root, is at least one.
- startnode is non-garbage Û the SRC of startnode is at least one.

Since it is not obvious that APC terminates, we must prove its termination too.
So it is sufficient to prove the following four theorems. This is done below.

Theorem 1
APC called on startnode terminates.

After a call to APC with argument startnode is terminated, the following three theorems hold:

Theorem 2
There are no strong cycles.

Theorem 3
The SRC of all nodes, except startnode and the program root, is at least one.

Theorem 4
startnode is non-garbage Û the SRC of startnode is at least one.

Definitions
We say that a call to APC on a node X is done when APC is called with argument X.
We say that a call to AGB on a node X is done when AGB is called with first argument X.
When APC is called with argument a node, we call this node the topnode of this incarnation of APC.
When AGB is called with second argument node, we call node the topnode of this incarnation of AGB.
A complete superincarnation (on topnode) of the procedure AGB is the set of all incarnations of AGB which are called within one call to AGB with both arguments topnode, and which all have topnode as second argument.
A partial superincarnation (on a node x with source topnode) of the procedure AGB is the subset of the complete superincarnation of AGB on topnode, that contains only all incarnations of AGB which are called within one call to AGB on the node s which is

the endpoint of p.

Abbreviations
We will call a node reachable if it is reachable from PR, SN or O. We will call a node strongly reachable if it is strongly reachable from PR, SN or O.

PROOF OF THE THEOREMS

Theorem 1
APC called on startnode terminates.

PROOF OF THEOREM 1
First we prove two lemmata:

Lemma 1.1
Every partial superincarnation S of one call to AGB contains a finite number of calls to APC.

Proof of lemma 1.1
All nodes on which a call to APC is done are marked as topnode, and on marked nodes APC will not be called again. These nodes stay marked as topnode until the last call to APC is done in S. Since the number of nodes in the graph is finite, the number of calls to APC in S, and so the number of calls to APC must be finite.

Lemma 1.2
Every partial superincarnation S of one call to AGB contains a finite number of incarnations of AGB.

Proof of lemma 1.2 by contradiction.
Assume that there is a partial superincarnation S of AGB with source topnode that contains infinitely many incarnations of AGB.
Note that AGB traverses the subgraph of topnode via strong pointers, so that it can easily be seen that there must be a cycle in the graph, which AGB traverses infinitely many times via strong pointers (otherwise the number of incarnations is finite, since the number of nodes in the graph is finite).
Note that this cycle cannot contain topnode, since otherwise (line 56 in the program section 2.1) no more calls to AGB will be done, and the total number of incarnations is finite.
Since AGB traverses this cycle, there must be a path from topnode to it. Now consider the entrynode E of this path in this cycle. On this node too an infinite number of calls to AGB in S is done. Note that the RC of E is at least two. Consider the mark in E. We have the following possibilities when a call to AGB is done on E:

- E is not marked as topnode and not marked as visited.
  To guarantee new calls to AGB on E its SRC must be one (lines 59 and 62 in the program). Now a call to APC is done on E; this call starts a new superincarnation of AGB on E. After this call E is marked as topnode. Since there must be an infinite number of incarnations of AGB on E, its SRC must be zero after the call to APC, otherwise no more calls to AGB will be done on E (lines 63 to 66 in the program).
- E is marked as topnode and not marked as visited.
  Then E becomes marked as visited, and a new call to AGB will be done on

E (lines 72 to 74 in the program)

- E is marked as topnode and marked as visited.
  Then no more calls to AGB on E will be done in this superincarnation.

As a conclusion we can say, that the first possibility is applicable the first time a call to AGB is done on E only, since thereafter E will stay marked as topnode until S is finished. Once E is marked as visited in S, no more calls on E will be done. Now the only possibility to guarantee more calls to AGB on E, is that calls to AGB on E are done in other superincarnations; in those calls E can become marked as visited too, and afterwards unmarked as visited, that is to say its mark is set to one again (line 75 in the program). Consequently the original visit mark is lost.
So we arrive at the following scenario: E becomes marked as topnode, subsequently a call on E to AGB in S is done; thereby E becomes marked as visited. In another superincarnation (– S, after a call to APC on E) E becomes unmarked as visited, that is, its counter is set to one again. Thereafter a new call to AGB on E is done in S, and so on.
But we know from lemma 1.1 that only a finite number of calls to APC is done in a partial superincarnation, so that the mark can only be removed a finite number of times. Once all these calls are done, E stays marked as visited, and no more calls to AGB will be done any more on it.
So the assumption that there are infinitely many incarnations of AGB in a partial superincarnation of AGB leads to a contradiction.

With these lemmata we can prove the theorem:
The number of incarnations of APC nested in each other is finite.
This can be seen as follows: APC called on startnode generates a complete superincarnation of AGB, containing a finite number of partial superincarnations. Each of these partial superincarnations contains a finite number of calls to APC (lemma 1.1).
Each incarnation of APC does a finite number of calls to AGB, so each incarnation will terminate.
This can be seen as follows: a call to APC generates a finite number a complete superincarnation of AGB, containing a finite number of partial superincarnations. All these partial superincarnations contain all a finite number of incarnations of AGB (lemma 1.2).
So, APC terminates.

END OF PROOF OF THEOREM 1

Theorem 2
After a call to APC on startnode, there are no strong cycles.

PROOF OF THEOREM 2
We are going to prove the lemma below. The theorem follows directly from it, if we realize, that before the call to APC on startnode there are no strong cycles in the subgraph of the program root, and the call to APC only affects the subgraph of startnode.

Lemma 2.1
After a call to APC on topnode, there are no strong cycles in the subgraph of topnode.

Proof of lemma 2.1
When APC starts, the invariants
- there are no strong cycles.
- all nodes which are reachable are strongly reachable.

hold.

Since we start with a graph without strong cycles, the only way to get a strong cycle is by making weak pointers strong. Now the reasoning is as follows: we show that when pointers are made strong, the possibly generated strong cycles all contain a topnode of an existing incarnation of AGB. Subsequently we show that when a call to AGB on topnode is finished, there are no strong cycles containing topnode any more.

Weak pointers are made strong by swapping nodes, or by explicitly making weak pointers strong.
When we consider the algorithm, we see that topnode is swapped, thereby possibly generating strong cycles containing topnode. After the swapping a call to AGB is done with second argument topnode.
On one place in the algorithm a weak pointer is explicitly made strong, namely when after a call to APC the SRC of nextnode is zero (line 64 in the program in section 2.1). In this case strong cycles can be generated too, but all these cycles, containing nextnode, also contain a topnode of an incarnation of AGB. This can be seen as follows. There are two possibilities:
- Such a cycle contains startnode. Then we are done, since on startnode a call to APC is done, such that it is a topnode of an incarnation of AGB.
- Such a cycle does not contain startnode. In this case there is a path from startnode to a node in the cycle containing nextnode. Consider the entrynode E of this path in this cycle. The SRC of E must be one, otherwise AGB would stop traversing the graph here, and there would be no call done on nextnode. But then a call to APC has been done on E. So E is a topnode of an incarnation of AGB. So the possible strong cycle which has been generated contains a topnode of an incarnation of AGB.

So indeed to prove that after a call to APC on topnode there are no strong cycles in the subgraph of topnode, it suffices to show that after a call to AGB all strong cycles containing topnode which were generated, have disappeared. Since AGB traverses the graph via strong pointers, it is also sufficient to show that when AGB does not call itself recursively any more, the pointer which AGB is treating is on that moment not contained in a strong cycle which contains topnode too. Below in the algorithm we use assertions to prove this. The proof is simple and straightforward.

```
1. AdjustGraphBetween (node, topnode: NODE) =
2. FOR ptr IN Sons (node)
3. DO    IF IsStrong (ptr)
      a. THEN nextnode := ptr^;
            i. IF nextnode = topnode
           ii. THEN MakeWeak (ptr)
                  1. {there is no strong cycle which contains both ptr and
                     topnode} (A1)
          iii. ELSIF NotMarkedAsTopnode (nextnode)
           iv. THEN IF SRC (nextnode) 3 2
                  1. THEN MakeWeak (ptr)
                        a. {there is no strong cycle which contains both ptr
                           and topnode} (A2)
```

```
                    2. ELSIF WRC (nextnode) 3 1
                    3. THEN AdjustPresumableCycle (nextnode);
                         a. {there is no strong cycle which contains both ptr
                            and topnode} (A3)
                         b. IF SRC (nextnode) = 0
                         c. THEN MakeStrong (ptr);
                                i. MarkAsTopnode (nextnode);
                               ii. AdjustGraphBetween (nextnode,
                                   topnode);
                              iii. {there is no strong cycle which contains
                                   both ptr and
                                                        topnode}
                                   (A4)
                               iv. UnmarkAsTopnode (nextnode)
                         d. ELSE (* SKIP *)
                                i. {there is no strong cycle which contains
                                   both ptr and
                                          a. topnode} (A5)
                         e. END
                         f. {there is no strong cycle which contains both ptr
                            and topnode} (A6)
                    4. ELSE AdjustGraphBetween (nextnode, topnode)
                         a. {there is no strong cycle which contains both ptr
                            and
                                        topnode} (A7)
                    5. END
                    6. {there is no strong cycle which contains both ptr and
                       topnode} (A8)
                v. ELSIF NotMarkedAsVisited (nextnode)
               vi. THEN MarkAsVisited (nextnode);
                    1. AdjustGraphBetween (nextnode, topnode);
                    2. {there is no strong cycle which contains both ptr and
                       topnode} (A9)
                    3. UnmarkAsVisited (nextnode)
              vii. ELSE (* SKIP *)
                    1. {there is no strong cycle which contains both ptr and
                                                         i. topnode} (A10)
             viii. END
          b. ELSE (* SKIP *)
                i. {there is no strong cycle which contains both ptr and topnode}
                   (A11)
          c. END
     4. END.
     5. {there is no strong cycle which contains both node and topnode} (A12)
```

Proof of the assertions

(A1) -

ptr itself is weak.

(A2) -

ptr itself is weak.

(A3) -

The postcondition of AGB (and so the postcondition of APC too) tells us that there is no strong cycle containing nextnode; ptr is pointing to nextnode, so there is surely no strong cycle which contains both ptr, nextnode and topnode.
(A4) -
This follows with the reasoning in the proof of (A3) from the postcondition of AGB.
(A5) -
This is follows from (A3).
(A6) -
This follows from (A4) and (A5).
(A7) -
This the postcondition of AGB.
(A8) -
This follows from (A2), (A6) and (A7).
(A9) -
This follows with the reasoning in the proof of (A3) from the postcondition of AGB.
(A10) -
Assume that there is a strong cycle which contains both ptr and topnode. Then there must be a strong cycle which contains both nextnode and topnode.
Since nextnode is marked as visited, we know that nextnode has been marked as topnode, and that it already has been visited in this superincarnation. So AGB has traversed a cycle via strong pointers, which contains nextnode. But all these cycles can not contain topnode, since AGB does not call itself recursively any more as soon as topnode is encountered. So we have a contradiction.
So there is no strong cycle which contains both ptr and topnode.
(A11) -
This follows from (A1).
(A12) -
This follows with the reasoning above from (A1), (A8), (A9), (A10) and (A11).

END OF PROOF OF THEOREM 2

Theorem 3
After a call to APC with both arguments startnode, the SRC of all nodes, except the program root and startnode, is at least one.

PROOF OF THEOREM 3
We prove the lemma below with assertions. The theorem follows directly from it.

Lemma 3.1
After a call to APC on topnode is finished, the SRC of all nodes in the subgraph of topnode, except topnode, is at least one.

Proof of lemma 3.1
```
{¡ nodes N ε subgraph of topnode\topnode: SRC (N) 3 1} (A1a) ∧
{WRC (topnode) ≥ 1} (A1b)
AdjustPresumableCycle (topnode) =
 INC (depth);
 Swap (topnode);
 {¡ nodes N | subgraph of topnode: SRC (N) ≥ 1} (A2)
 MarkAsTopnode (topnode);
 AdjustGraphBetween (topnode, topnode);
 {¡ nodes N | subgraph of topnode: SRC (N) ≥ 1} (A3)
 UnmarkAsTopnode (topnode)
 DEC (depth).
{¡ nodes N | subgraph of topnode: SRC (N) ≥ 1} (A4)

{¡ nodes N | subgraph of topnode: SRC (N) ≥ 1} (A5)
AdjustGraphBetween (node, topnode: NODE) =
 FOR ptr IN Sons (node)
 DO    IF IsStrong (ptr)
       THEN ne¡tnode := ptr^;
            IF nextnode = topnode
            THEN MakeWeak (ptr)
                 {¡ nodes N | subgraph of topnode \ topnode: SRC (N) ≥ 1} (A6)
            ELSIF NotMarkedAsTopnode (nextnode)
            THEN IF SRC (nextnode) ≥ 2
                 THEN MakeWeak (ptr)
                      {¡ nodes N| subgraph of topnode \ topnode: SRC (N) ≥
1} (A7)
                 ELSIF (WRC (nextnode) 3 1)
                 THEN AdjustPresumableCycle (nextnode);
                      {¡ nodes N | subgraph of topnode \ {topnode,
nextnode}:
                                          SRC (N) ≥ 1} (A8)
                      IF SRC (nextnode) = 0
                      THEN MakeStrong (ptr);
                           {¡ nodes N | subgraph of topnode \ topnode:
                                          SRC (N) ≥ 1} (A9)
                           MarkAsTopnode (nextnode);
                           AdjustGraphBetween (nextnode, topnode);
                           {¡ nodes N | subgraph of topnode \ topnode:
                                          SRC (N) ≥ 1} (A10)
                           UnmarkAsTopnode (nextnode)
                      ELSE (* SKIP *)
                           {¡ nodes N | subgraph of topnode \ topnode:
                                          SRC (N) ≥ 1} (A11)
                      END
                           {¡ nodes N | subgraph of topnode \ topnode:
                                          SRC (N) ≥ 1} (A12)
                 ELSE AdjustGraphBetween (nextnode, topnode)
                      {¡ nodes N | subgraph of topnode \ topnode:
                                          SRC (N) ≥ 1} (A13)
                 END
                 {¡ nodes N | subgraph of topnode \ topnode: SRC (N) ≥ 1}
(A14)
            ELSIF NotMarkedAsVisited (nextnode)
            THEN MarkAsVisited (nextnode);
                 AdjustGraphBetween (nextnode, topnode);
                 {¡ nodes N | subgraph of topnode \ topnode: SRC (N) ≥ 1}
(A15)
                 UnmarkAsVisited (nextnode)
            ELSE (* SKIP *)
                  {¡ nodes N | subgraph of topnode \ topnode: SRC (N) ≥ 1}
(A16)
            END
        ELSE  (* SKIP *)
            {¡ nodes N | subgraph of topnode \ topnode: SRC (N) ≥ 1} (A17)
        END
  END.
{¡ nodes N | subgraph of topnode \ topnode: SRC (N) ≥ 1} (A18)
```

Proof of the assertions

(A1) -
These are, with corollary 1, preconditions of APC in DeletePtr. The preconditions of APC in APC itself are weaker.

(A2) -
Before the swap the WRC of topnode is 3 1 (A1b)

(A3) -
This is the postcondition of APC.

(A4) -
This follows from (A3).

(A5) -
This is the precondition of AGB in APC.

(A6) -
A strong pointer to topnode is weakened; the RC of other nodes is not affected.

(A7) -
The SRC of nextnode was at least two.

(A8) -
This is the postcondition of AGB, and so the postcondition of APC.

(A9) -
ptr to nextnode has become strong.

(A10) -
This is the postcondition of AGB.

(A11) -
This follows from (A8).

A12) -
This follows from (A9), (A10) and (A11)

(A13) -
This is the postcondition of AGB.

(A14) -
This follows from (A7), (A12) and (A13).

(A15) -
This is the postcondition of AGB.

(A16) -
This is the postcondition of AGB.

(A17) -
This follows from (A5).

(A18) -
This follows from (A6), (A14), (A15), (A16) and (A17).

END OF PROOF OF THEOREM 3

Theorem 4
After a call to APC on startnode is finished the following equivalence holds:
startnode is non-garbage ⇔ the SRC of startnode is at least one

PROOF OF THEOREM 4
Consider the following lemmata 4.1 to 4.4.
The lemmata 4.1 and 4.4 prove the implication in theorem 4 from the left to the right and from the right to the left, respectively. In lemma 4.4 lemma 4.3 is used; in lemma 4.3 lemma 4.2 is used.

Lemma 4.1
After a call to APC on startnode is finished the following implication holds:
startnode is non-garbage ⇦ the SRC of startnode is at least one

Proof of lemma 4.1
This follows from the structure of the graph: after the call to APC the SRC of all nodes except the program root is at least one (assumption and theorem 3), and there are no strong cycles (theorem 2), so (corollary 1) there must be a strong path from the program root to startnode.

Lemma 4.2
Consider a topnode on which a call to AGB is done. The strong pointers to this topnode will not be weakened but in the complete superincarnation of AGB on topnode.

Proof of lemma 4.2
In the whole superincarnation of AGB on a node, this node stays marked as topnode. In the algorithm, nodes marked as topnode are not swapped, and pointers to marked nodes are not weakened, unless the node is topnode of the superincarnation. So pointers to nodes will not become weak, unless the node is the topnode of the superincarnation.

Lemma 4.3
Assume that on a node N a call to APC is done, and before the call the following properties hold:
P1: the SRC of N is one, and the WRC of N is at least one, if APC is called on startnode
the SRC of N is zero, and the WRC of N is at least one, if APC is called on another node
P2: there exists an acyclic path P from the program root to N; all nodes on this path are unmarked
P3: the pointer to N contained in P is weak
and after the call to APC the following property holds:
P4: the SRC of N is zero
and the length of P is, say, z
Then there is a node M on the path from the program root to N, for which the following properties hold:
Q1: the SRC of M is one, and the WRC of M is at least one
Q2: there exists an acyclic path P' from the program root to M; all nodes on this path are unmarked

Q3: the pointer to M contained in P' is weak
Q4: a call to APC has been done on M, and after this call the SRC of M was zero
Q5: the length of P' is < z

Proof of lemma 4.3
After the call to APC the SRC of N is zero (P4); this means that all pointers which were weak before the call to APC, and have become strong by the swap in APC, are weakened again by AGB. So the pointer contained in P is weakened too. This pointer has been made weak in the superincarnation called on topnode (lemma 4.2). So there must be a path from N to the source of the pointer to N in P, otherwise this pointer could not have been weakened. So there must be a cycle containing N and at least one other node of P. The entrynode of P in this cycle is the node M with the desired properties. We prove below the properties Q1 to Q5.
Q1: From the reasoning above follows that a call to AGB is done on M. When this call is done there are two possibilities:
- SRC (M) ≥ 2; but then AGB stops traversing the graph and the pointer to N would not have become weak. So it is impossible that the SRC of M is ≥ 2.
- SRC (M) = 1. Now a call to APC is done on M; within this call the pointer to N can not have been weakened (lemma 4.2). So after this call the SRC of M must be zero, such that the original call to APC can weaken the pointer to N. So the only possibility is that the SRC of M is 1 when a call to AGB is done on it.
Q2: M is contained in P, an acyclic path from the program root to N. The acyclic path from the program root to M is the path P' with the desired properties.
Q3: We know from Q1 that the SRC of M is 1. There is a strong pointer in the cycle to M, so all other pointers to M, including the one contained in P', must be weak.
Q4: See the proof of Q1.
Q5: From the reasoning above follows that M is not the same node as N, so the length of P' must be smaller then the length of P.

Lemma 4.4
After a call to APC on startnode is finished the following implication holds:
startnode is non-garbage _ the SRC of startnode is at least one

Proof of lemma 4.4 by contradiction
Assume that after the call to APC on startnode its SRC is zero.
In this case startnode has all properties P1 to P4 mentioned in lemma 4.3: its SRC is zero before the call to APC on it (P1). There is a path P from the program root to startnode, since startnode is non-garbage (by assumption); since no call to APC has been done yet, there is no node marked. This proves (P2). Since the SRC of startnode is zero before the call to APC, there are only weak pointers pointing to it, so the pointer to startnode in P is weak too (P3). After the call to APC the SRC of startnode is zero by assumption (P4).
So (lemma 4.3) there must be a node M contained in P with the properties Q1 to Q4 of lemma 4.3, and the length of the path P' from the program root to M is smaller then the length of P (Q5).
The properties Q1 to Q4 are exactly the same as the properties P1 to P4. So there must be a node M' with again the properties Q1 to Q5, and the length of the path from the program root to M' P'' again must be smaller then the length of the P' (Q5). But this leads to an infinite number of nodes, all on the path from the program root to

startnode, and the length of the path from the program root to such a node is smaller and smaller, but by definition positive!
So we have a contradiction. So the SRC of startnode must be at least one after the call to APC on it.

END OF PROOF OF THEOREM 4

4 Optimizations, further research and conclusions

4.1. Optimizations

We first discuss a few possible optimizations:
As we have seen, most of the trouble is caused by the weak pointers in the graph. If the last strong pointer to a node which WRC is still at least one, the subgraph of the node in question is (partially) traversed. Weak pointers encountered during the traversion cause even more trouble.
Optimization of the algorithm can be done by generating as few as possible of weak pointers, and by treating them in a more efficient manner.
In the presented algorithm, all new (copied) pointers are by definition weak. A simple optimization would be to administer whether there are already cycles (weak pointers) in the graph. As long this is not the case all copies of pointers can be strong. As long as there are no strong cycles, the time overhead caused by the algorithm with this optimization is the same as that of the classic reference counting algorithm. Of course this optimization can be further extended, as demonstrated in [Salkild85], but how to establish whether a copy of a strong pointer can be strong, is implementation dependent.
A second (simple) optimization can be done in DeleteNode. If first the weak pointers are deleted, and thereafter the strong pointers, many cases are avoided, in which the last strong pointer to a node is deleted to which still weak pointers are pointing.
The third optimization we mention can be done in AGB: as it is implemented now, all sons are successively inspected, and if a strong one is found, its subgraph is traversed. In this way it can happen, that a pointer which was initially weak has become strong during traversion of the subgraphs of sons, on which a call to AGB was done before. Its subgraph will be later on again traversed, although there are no strong cycles in it any more, and if there was an external reference to it, there would already be a strong path to 'topnode'. A more optimal way would be to establish beforehand which sons are strong, and to traverse only their subgraphs.

4.2. Further research

An important subject for further research is to establish whether the algorithm can be extended for use in a distributed environment. Reference counting algorithms are very suited for distributed computer systems, since with each pointer manipulation the information for garbage collection is automatically passed among the constituting processors. Unfortunately most reference counting algorithms can not handle cyclic structures.
In the algorithm presented here still all information for the garbage collector is present in the nodes, but it seems a challenge to extend it for distributed use. We expect that it will not be very easy, but if we succeed in developing an efficient distributed version of it, it would be very useful in distributed computer systems.

Further has to be investigated how the presented algorithm, with its optimizations, behaves in a realistic graph rewriting system. Of course, even if the presented optimizations are implemented, one can easily think of pathological graphs for which an other garbage collection algorithm would be much more efficient.

4.3. Conclusions

We have presented an algorithm for garbage collection with reference counters that can handle cycles. With some effort we also managed to give a proof of it.
This proof has been the principal subject of this paper. However, for an algorithm to be useful in practice, it must not only be correct, but also be reasonably efficient. In this case, the reader can easily imagine that there are pathological cases in which the algorithm is very inefficient. It might be possible that in each new incarnation of APC on all nodes of the graph another incarnation of APC is started., only to determine whether 'startnode' is garbage or not. So the complexity of the algorithm is at least exponential.
Although we didn't study the behaviour of the algorithm very thoroughly, we can say something about its efficiency: when there are no cycles at all, the algorithm is almost as efficient as the classic RC algorithm: there is no time overhead, and the space overhead is one reference counter and one counter for the marks per node. If there are only a few cycles, which are not too badly nested, then not very much traversion of the graph has to be done when the last strong pointer to a possible non-garbage node is deleted.

5 References

[Brus et al 87]
Brus, T., Eekelen, M.C.J.D. van, Leer, M. van, Plasmeijer, M.J., Clean - A Language for Functional Graph RewritingÓ, Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture (FPCA '87), Portland, Oregon, USA, LNCS 274, pp 364-384, September 1987.

[Brownbridge85]
D.R. Brownbridge, "Cyclic reference counting for combinator machines", Proceedings of the Second Conference on Functional Programming Languages and Computer Architecture (FPCA), Nancy, France, LNCS 201, pp 273-288 , September 1985.

[Salkild85]
J.D. Salkild, "Implementation and analysis of two cyclic reference counting algorithms", Msc. Computer Science, Department of Computer Science, University College London, 1985.

[Turner79]
Turner, D.A., "A new Implementation Technique for Applicative Languages", Softw. Pract. and Experience, Vol 9 (1), pp. 31-49, January 1979.