Algorithms for On-the-fly Garbage Collection

MORDECHAI BEN-ARI
Tel Aviv University

A new algorithm is described for on-the-fly garbage collection. The new algorithm uses only two colors and has a simple correctness proof. Two variations on the algorithm are then derived: One attempts to minimize the amount of marking that must be done, and the other is an incremental garbage collector.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Program Verification; D.4.2 [**Operating Systems**]: Storage Management

General Terms: Algorithms, Verification

Additional Key Words and Phrases: Garbage collection, correctness proofs

1. INTRODUCTION

In [2] an algorithm was presented for on-the-fly garbage collection. This algorithm (henceforth called the DLMSS algorithm) is one of the most difficult concurrent programs ever studied. It has been the subject of two formal proofs [3, 5]. The proofs are extremely complex; it would be difficult to check them mechanically. The informal proof of the algorithm for the same problem given by Kung and Song [6] is even longer and more complex.

We looked for and found a new algorithm for on-the-fly garbage collection. The criterion that the algorithm was to satisfy was that the correctness proof should be elementary. The algorithm is not much simpler than DLMSS. Much logical machinery is still required to express the invariants. However, each step in the verification of the invariants is almost trivial and could be checked mechanically.

Our algorithm is probably better than DLMSS in that it uses two colors instead of three (four colors are used in [6]). Our algorithm is probably worse than DLMSS in that an implementation might be less efficient. Neither point is as important to us as the simplicity of the proof. A significant point in favor of our algorithm is that it is robust with regard to the seemingly innocent variation that introduces a bug into DLMSS.

A preliminary version of this paper was presented at the 9th International Colloquium on Automata, Languages and Programming, Aarhus, Denmark, July 1982.
Author's present address: HaGedud HaIvri 81A, Kiryat Haim 26306, Israel.

The new algorithm was then used to develop other algorithms that might be significantly better in practice. The simplicity of the correctness proof was crucial: A new idea for modifying the algorithm could be easily checked to see if the proof could be similarly modified. One variation of the algorithm attempts to minimize the number of nongarbage nodes that must be marked. The second variation is an incremental garbage collector that does not collect all the garbage nodes on a cycle but is likely to collect a few nodes very rapidly.

On-the-fly garbage collection has been of theoretical interest only until recently. The Intel iAPX-432 computer and iMAX operating system contain a hardware/software implementation of the DLMSS algorithm [9]. The hardware does the marking required by the mutator, while the software runs the collector as a concurrent process. The deficiency of our algorithm—it requires at least one extra pass over the cells—would not be a factor in the Intel system, which performs garbage collection on a relatively small number of objects (code and data segments, activation records, etc.). On the other hand, our algorithm would save 1 bit in the object descriptors and thus simplify the mutator marking circuitry. All the algorithms we give are compatible with the current mutator marking and can be implemented by changing the software only.

2. ON-THE-FLY GARBAGE COLLECTION

We follow the model of the problem from [2]. (A survey article on garbage collection was recently published by Cohen [1].) In a system such as a LISP interpreter, pointer manipulations can cause certain nodes to become inaccessible from a designated root. Such nodes, called garbage, must be identified and recycled by being linked to a list of free nodes. Thus we have two actors in the system. The mutator is that part of the system doing "useful" work, and the collector is that part of the system that recycles garbage. An on-the-fly garbage collector is a system that allows concurrent execution of the mutator and the collector.

Garbage collection is done by a two-phase algorithm. The collector marks all nodes accessible from the root, and then it appends all unmarked (and hence inaccessible) nodes to the free list. We are given an array of nodes representing the linearly addressable address space. Each node has a field that contains a color (white or black) and a fixed number of fields that contain pointers (indices) to other nodes. Some of the nodes are designated as roots. In [2] it is shown that by considering both the pointer to the free list and the special cell NIL to be roots, then the only mutator instruction that modifies the data structure is one that chooses two accessible nodes i and k and causes k to become a son (say the jth of i. Of course, if some node l was previously the jth son of i, it could happen that l is no longer accessible from the root. The task of the collector is to identify these garbage nodes and to append them to the free list.

Following [2], we abstract the problem by omitting all other details of the mutator's program as well as the (straightforward) details of the pointer manipulations required to implement the free list. A further abstraction is to ignore the synchronization that must be done when the mutator attempts to remove a node from an empty free list. If the removal is done at one end of the list and

the appending at the other, this should happen infrequently and any convenient synchronization primitive can be used.

The computational environment that the algorithms are directed to is a real-time system where we assume that a processor (or background process) is dedicated to garbage collection. Thus the mutator contribution to garbage collection must be minimized, whereas collector time is essentially free. Similarly, scratch memory is limited and no queues or stacks are maintained in order to shorten traversal of the data structure. This contrasts with [6]: Kung and Song even require the mutator to enqueue the nodes that it marks.

3. THE ALGORITHM

The algorithm is given as fragments of an Ada[1] program. The line numbers on the executable statements are used in the proof.

```
-- The data structure.
    type Hue is (White, Black);
    type Index is new Integer range 1..Number_of_Nodes;
    subtype Roots is Index range 1..Number_of_Roots;
    type Sons is new Integer range 1..Number_of_Sons;
    type Node is
         record
             Son:   array(Sons) of Index;
             Color: Hue := White;
         end record;
M:  array (Index) of Node;
             -- M is initialized so that all nodes are
             -- linked on the free list and all links
             -- not so used are pointing to the root NIL.
-- The mutator.
    -- The mutator executes the following instructions at its
    -- discretion. It ensures that both R and T point to nodes
    -- accessible from a root when instruction a0 is executed.
a0  M(R).Son(S) := T;
a1  M(T).Color := Black;
-- The collector.
    -- The collector executes procedure Marking_Phase and then
    -- appends white nodes to the free list.
    procedure Marking_Phase is
        Black_Count, Old_Black_Count: Integer := 0;
b0  begin
        -- Blacken the roots.
b1          for I in Roots loop
b2              M(I).Color := Black;
b3          end loop;
b4      Main: loop
        -- Propagate the coloring.
```

[1] Ada is a trademark of the U.S. Department of Defense.

```
b5          for I in Index loop
b6              if M(I).Color = Black then
b7                  for J in Sons loop
b8                      M( M(I).Son(J) ).Color := Black;
b9                  end loop;
b10             end if;
b11         end loop;
        -- Count the number of Black nodes.
b12         Black_Count := 0;
b13         for I in Index loop
b14             if M(I).Color = Black then
b15                 Black_Count := Black_Count + 1;
b16             end if;
b17         end loop;
        -- Repeat the Main loop if more Black nodes than before.
b18         if Black_Count > Old_Black_Count
b19             then Old_Black_Count := Black_Count;
b20             else exit;
b21         end if;
b22     end loop Main;
b23 end Marking_Phase;
        -- Collector cycle.
c0      Marking_Phase;
        -- Collect the white nodes.
c1          for I in Index loop
c2              if M(I).Color = White
c3                  then Append_to_Free(I);
c4                  else M(I).Color := White;
c5              end if;
c6          end loop;
```

Remark. The algorithm can be made more efficient by repeating the propagation loop b_5–b_{11} until no new nodes are colored (as in the DLMSS algorithm) and only then proceeding to count the black nodes.

4. THE SAFETY OF THE ALGORITHM

In this section we prove that the following safety property holds: When `Marking_Phase` terminates, all white nodes are garbage. Once this has been shown, the proof of the correctness of the algorithm is straightforward. The safety property will be proved by the method of invariants of concurrent programs [7, 8]. We attach invariants to points in the program and then show that execution of an instruction of either process preserves the truth of the invariant. We use explicit propositions for the locations of the program counters [4], although auxiliary variables could be used as well [8].

So as not to obscure the main ideas, the exposition is limited to the critical facets of the proof. A mechanically verifiable proof would need all sorts of trivial invariants (e.g., `Marking_Phase` does not change the data structure) and elementary transformations of our invariants (e.g., the invariants we give for line b_8 have counterparts at other points in the loop b_5–b_{11} with appropriate adjustments of the indices).

The following propositions are used in the proof.

$b_8(i, j)$ $\leftrightarrow$ (the collector is at b_8) $\wedge$ $(I = i)$ $\wedge$ $(J = j)$.
$BW(i, j, k)$ $\leftrightarrow$ $(M(i).\ \mathrm{Son}(j) = k)$ $\wedge$ $(M(i).\ \mathrm{Color} = \mathrm{Black})$ $\wedge$ $(M(k).\ \mathrm{Color} = \mathrm{White})$ $\wedge$ ($M(i)$ is accessible from the root).

In words: k is a white jth son of an accessible black node i.

$$a_1(r, s, t) \leftrightarrow (\text{the mutator is at } a_1) \wedge (R = r) \wedge (S = s) \wedge (T = t).$$

In words: The mutator has made t the sth son of r but has not yet colored node t.

We also use the following notation:

$$\mathrm{Blacks} = |\ \{i \mid M(i).\ \mathrm{Color} = \mathrm{Black}\}\ |$$

This is the number of black nodes in the data structure and may be larger than Black_Count: the number of black nodes that the collector has counted.

$(i, j) \ll (i', j')$: Lexicographical order on nodes and sons. Similarly for the other relations.

LEMMA 1. *The following formula is invariant*:

$$[b_8(i, j) \wedge (\mathrm{Blacks} = \mathrm{Old\_Black\_Count})] \rightarrow \forall(r, s, t)\ (\{[(r, s) \ll (i, j)] \wedge BW(r, s, t)\} \rightarrow a_1(r, s, t)).$$

PROOF (*Informal*). There can be a triple such that $BW(r, s, t)$ and $(r, s) \ll (i, j)$ if after the collector has passed (i, j) the mutator executes the assignment but not the coloring. Since the mutator always "covers its tracks" immediately, at most one such triple can be in existence. Once a white node has been colored black, Blacks = Old_Black_Count is false and the invariant trivially holds. Thus we do not care if coloring (by either the mutator or the collector) introduces new black nodes and hence possibly new triples (u, v, w) such that $BW(u, v, w)$.

(*Formal*). During the initial execution of the collector loop b_5–b_{11}, there are no values (r, s) such that $(r, s) \ll (1, 1)$. Assume now as an induction hypothesis that the invariant is true. It could be falsified only if (1) both the antecedent and the consequent are false and the execution of an instruction "suddenly" makes the antecedent true (without simultaneously making the consequent true), or (2) both are true and the execution of an instruction "suddenly" makes the consequent false (without falsifying the antecedent).

Let us look at the invariant in the following equivalent form.

$$\forall(r, s, t)\ (\{[b_8(i, j) \wedge (\mathrm{Blacks} = \mathrm{Old\_Black\_Count})] \wedge [(r, s) \ll (i, j)] \wedge BW(r, s, t)\} \rightarrow a_1(r, s, t)).$$

We now show that neither (1) nor (2) can occur for any possible step of either process. Consider first the collector. (1) If $BW(r, s, t)$ becomes true for $(r, s) \ll (i, j)$ because the collector blackens r, then (Black = Old_Black_Count) is simultaneously falsified. If $BW(r, s, t)$ becomes true for $(r, s) \ll (i, j)$ because (r, s, t) is (i, j, k) and (i, j) is incremented, then k is blackened and (Blacks = Old_Black_Count) is falsified. Thus we have failed to make the antecedent alone true. (2) The collector cannot affect the truth of a_1 (r, s, t).

Now for the mutator. (1) As for the collector, if the mutator blackens a node, then the invariant trivially holds. The mutator can make the antecedent true by executing the assignment at a_0, but that makes the consequent a_1 (r, s, t) true also. (2) The mutator can falsify the consequent (when the antecedent is true for (r, s, t)) only by falsifying Blacks = Old_Black_Count. □

LEMMA 2. *The following formula is invariant*:

$$[b_8(i, j) \wedge (\text{Blacks} = \text{Old\_Black\_Count})] \rightarrow (\exists\ (r, s, t)\ \{[(r, s) \ll (i, j)] \wedge BW(r, s, t)\} \rightarrow \exists\ (u, v, w)\ \{[(u, v) \geq\geq (i, j)] \wedge BW(u, v, w)\}).$$

PROOF. As in the proof of the previous lemma, any coloring will falsify Black = Old_Black_Count and preserve the invariant. If $BW(r, s, t)$ for $(r, s) \ll (i, j)$ and the node (u, v) promised by the consequent happens to be (i, j), then execution of a cycle of the collector could falsify the consequent but only if it simultaneously falsifies Black = Old_Black_Count.

So suppose that an assignment of the mutator causes $BW(r, s, t)$ to become true for $(r, s) \ll (i, j)$, and that for all (u, v, w) $([(u, v) \geq\geq (i, j)] \rightarrow \sim BW(u, v, w))$, thus falsifying the invariant.

But the mutator ensures that $M(t)$ is accessible before the execution of this instruction. Since the roots are black and $M(t)$. Color = White (by $BW(r, s, t)$), there is a triple (r', s', t') such that $BW(r', s', t')$ and (incidently) there is a path from t' to t (or possibly $t = t'$). By the previous paragraph, we assume that $(r', s') \geq\geq (i, j)$ is not true. If (r', s', t') is the only such triple and $(r, s) == (r', s')$, then the invariant was false before executing this step—contradicting the induction hypothesis. The only possibility left is $(r', s') \ll (i, j)$ and $(r', s') \not= (r, s)$, but this contradicts Lemma 1 since it is not possible that $a_1(r, s, t)$ and $a_1(r', s', t')$ for distinct triples. □

Safety now follows from the fact that if Blacks > Old_Black_Count at b_{12}, then Black_Count > Old_Black_Count at b_{18} since the mutator can only make more black nodes, not fewer. Thus if the main loop is exited, it is only because, following b_{12}, Blacks = Old_Black_Count. Then Lemma 2 implies that there are no triples such that $BW(u, v, w)$; otherwise we would have $(u, v) \gg$ (Number_of_Nodes, Number_of_Sons), which is not possible. Since all roots are black, we can conclude that all white nodes are garbage. More formally, we would deduce from Lemma 2 the following lemma.

LEMMA 3. *The following formula is invariant*:

$$[b_{12} \wedge (\text{Blacks} = \text{Old\_Black\_Count})] \rightarrow \sim\exists(i, j, k)\ (BW(i, j, k)).$$

Then, to relate the number of black nodes to Black_Count, we need Lemma 4.

LEMMA 4. *The following formula is invariant*:

$$\{[b_{18} \wedge (\text{Blacks} = \text{Old\_Black\_Count})] \rightarrow [(\text{Black\_Count} = \text{Old\_Black\_Count}) \wedge \sim\exists(i, j, k)\ (BW(i, j, k))]\} \wedge$$

$$\{[b_{18} \wedge (\text{Blacks} > \text{Old\_Black\_Count})] \rightarrow (\text{Black\_Count} > \text{Old\_Black\_Count})\}.$$

PROOF. The first half follows from Lemma 3 since if there are no $BW(i, j, k)$, then all accessible nodes are black and thus the mutator cannot change the number of black nodes during the execution of b_{12}–b_{17}. The second half is immediate. □

LEMMA 5. *The following formula is invariant*:

$$b_{23} \rightarrow \forall(i)[(M(i).\text{Color} = \text{White}) \rightarrow M(i) \text{ is garbage}].$$

PROOF. By Lemma 4, we exit `Marking_Phase` if and only if there are no triples (i, j, k) such that $BW(i, j, k)$. Since the roots are black, there are no accessible white nodes. □

5. ON WOODGER'S SCENARIO

In [2] the authors note that they originally had the mutator execute its instructions in the opposite order—first coloring the node and then executing the pointer assignment:

```
M(T).Color  := Black;
M(R).Son(S) := T;
```

Woodger then found a bug by a scenario in which the mutator is suspended for a long period of time between the coloring and the assignment. The coloring can be lost by a complete execution of the collector, which whitens all nodes.

In our solution, exchanging the mutator's instructions cannot lead to a bug. Informally, the collector visits each node at least twice. Woodger's scenario shows that the collector may not mark an accessible node on the first visit. But if the mutator attempts to fool the collector a second time, it must first mark the target node—defeating the scenario.

LEMMA 6. *The following formula is invariant*:

$$\forall(r, s, t)\ \{[b_8(i, j) \wedge (\text{Blacks} = \text{Old\_Black\_Count}) \wedge BW(r, s, t)] \rightarrow [(i, j) \leq\leq (r, s)]\}.$$

PROOF. There is no way for either the collector or the mutator to create a new triple (u, v, w) such that $BW(u, v, w)$ without falsifying Blacks = Old_Black_Count. For any triple (r, s, t) existing at the beginning of a collector cycle such that $BW(r, s, t)$ and $(i, j) \leq\leq (r, s)$, when the collector increments (i, j) so that $(i, j) \gg (r, s)$, it also colors t and falsifies $BW(r, s, t)$. □

Lemma 3 follows immediately from Lemma 6. The proof of safety is extremely simple for this variation of the algorithm.

6. TERMINATION

Termination is easy to prove since we do not exit the main loop if Black_Count is larger than it was before. Since Black_Count is bounded from above by

Number_of_Nodes, the number of iterations of the main loop is also bounded. We express this formally in the following sequence of temporal logic formulas [10].

LEMMA 7.

$$\{b_{18} \wedge (\text{Black\_Count} = k) \wedge \Box \sim b_{23}\} \rightarrow \Diamond\{b_{18} \wedge (\text{Old\_Black\_Count} = k)\}.$$

PROOF. b_{20} is the only exit to b_{23}, so if we do not take that branch, then we assign k to Old_Black_Count. The rest of the code in the Main loop necessarily returns to b_{18} without assigning to Old_Black_Count. □

LEMMA 8.

$$\{b_{18} \wedge (\text{Black\_Count} = k) \wedge \Box \sim b_{23}\} \rightarrow \Diamond \{b_{18} \wedge (\text{Black\_Count} > k)\}.$$

PROOF. From Lemma 7 we have that $\{b_{18} \wedge (\text{Black\_Count} = k) \wedge \Box \sim b_{23}\}$ implies $\Diamond \{b_{18} \wedge (\text{Old\_Black\_Count} = k) \wedge \Box \sim b_{23}\}$ and then $\Diamond \{b_{18} \wedge (\text{Old\_Black\_Count} = k) \wedge (\text{Black\_Count} > \text{Old\_Black\_Count})\}$. □

We also need to prove that, if a node is garbage, it will eventually be collected. This follows from the fact that neither the mutator nor the collector ever colors a garbage node. Thus what is garbage (and trivially white) at the beginning of `Marking_Phase` will remain white, and hence (Lemma 5) is garbage at the end of `Marking_Phase`. Since every node is white at the beginning of a collector cycle, every garbage node will be collected no later than the cycle following that in which it becomes garbage. We leave it to the reader to formally express and prove these facts using the UNTIL operator.

7. A MORE EFFICIENT ALGORITHM

One problem that most garbage collection algorithms seem to have in common is that they spend a lot of time coloring nongarbage. This can be inefficient if the application contains large data structures that are modified only occasionally. The next algorithm that we describe is designed to be efficient in this situation. It goes as follows (ignoring for now the synchronization considerations).

Initially, all nodes are accessible and black. When the mutator is about to redirect an edge, it colors the old target white. This is intended to mean that the node is suspected of being garbage. Since this node may be the root of a subgraph of gargage nodes, the collector propagates the white coloring as far as possible. Then the collector applies the marking algorithm above and collects and blackens the remaining white nodes.

In the worst case (where some white node points to each root) this algorithm is no improvement, but it is hoped that if only a small set of nodes is accessible from each suspected node, the marking phase will converge rapidly. As an implementation note, it should be possible to identify situations that will not produce garbage and refrain from marking them as suspected. For example, a

LISP CONS—removing a node from the free list, directing it to two accessible nodes, and returning a pointer—cannot produce garbage.

The algorithm sketched above is not correct. The mutator may continue to whiten suspected nodes while the collector is blackening nodes. There are two possible bugs: (1) if the collector is convinced that all accessible nodes are now black, it may collect a new white node, which is actually accessible; (2) if the mutator created several garbage nodes by the redirection, only the first "suspected" node will be collected and is descendants will be lost. We need to be able to temporarily ignore the mutator's accusations without losing track of them. For this we introduce the color gray.

```
type Hue is (White, Gray, Black); -- Initial color is Black.
-- The mutator.
M( M(R).Son(S) ).Color := Gray;   -- Color old target.
M(R).Son(S) := T;
-- The collector.
procedure Propagate_White is
    Changed_Color: Boolean;
    Temp: Index;
begin
    loop
        Changed_Color := False;
        for I in Index loop
            if M(I).Color = White then
                for J in Sons loop
                    Temp := M(I).Son(J);
                    if M(Temp).Color = Black then
                        M(Temp).Color := White;
                        Changed_Color := True;
                    end if;
                end loop;
            end if;
        end loop;
        exit when not Changed_Color;
    end loop;
end Propagate_White;
-- Collector cycle.
    -- Make gray nodes white.
    for I in Index loop
        if M(I).Color = Gray
            then M(I).Color := White;
        end if;
    end loop;
    Propagate_White;
    New_Marking_Phase;
-- Collect white nodes.
    for I in Index loop
        if M(I).Color = White then
            Append_to_Free;
            M(I).Color := Black;
        end if;
    end loop;
```

`New_Marking_Phase` is like `Marking_Phase` except:

(1) b_6 and b_{14} become: **if** `M(I).Color` $\geq$ `Gray` **then**,
(2) b_8 becomes: `White_to_Black( M(I).Son(J) )`.

`White_to_Black(N)` is a primitive instruction equivalent to

```
if M(N) = White then M(N) := Black; end if;
```

Informally, whatever is garbage at the start of `Propagate_White` must be a gray node or a descendant of a gray node and will be whitened and collected. `New_Marking_Phase` treats black and gray as the "same" color. If the mutator generates a garbage node during a collector cycle, it will remain as such until the next cycle. Since it is not accessible, it will not be blackened and hence will remain suspicious for the next cycle of the collector. If the mutator colors an accessible node gray, the node will not be collected because it is not white and no nodes are whitened during `New_Marking_Phase`.

The formal proof of `New_Marking_Phase` is identical to that of `Marking_Phase` except that whenever "Black" is mentioned, "Gray or Black" should be used. The safety of the algorithm is immediate, since if a node is accessible, at the beginning of `New_Marking_Phase`, it will not be collected regardless of its color.

`Propagate_White` terminates because the number of cycles of its main loop is bounded by the number of black nodes upon entry. This follows since the mutator can only color nodes gray, and the collector can only color nodes white.

Note. The temporary variable both simplifies the proof and improves the efficiency of the algorithm. If it were not present, then the mutator could cut the link from $M(I)$ to $M(\ M(I).Son(J)\ )$ between the test and the coloring. If the old target is now garbage, it will have to wait an extra cycle to be collected since it has been colored gray by the mutator. Also, we have to add to the proof that the new son of $M(I)$, although colored white, is by definition reachable and will thus be marked during `New_Marking_Phase`.

The eventuality property needs to be proved: If a node is garbage, then in fact it will be collected. We leave a formal proof to the reader. Prove that (1) if a node becomes garbage, then it is colored gray or is a descendant of a gray node; (2) if the node is not collected, it remains so UNTIL the collector begins a cycle; (3) when the collector reaches `New_Marking_Phase`, the node is white; (4) the node is still white after `New_Marking_Phase`; (5) the node is collected during the loop following `New_Marking_Phase`. Note that only (1)–(3) are new for this version of the algorithm.

The primitive instruction `White_to_Black` is used to ensure that a gray node is not blackened. This can produce uncollectable garbage. If `White_to_Black` were not primitive then between the test and the assignment, the mutator could cut the edge from $M(I)$ to $M(\ M(I).Son(J)\ )$ and the mutator's gray coloring would be lost to the assignment of the color black.

8. IMPLEMENTATION

The mutator in the DLMSS algorithm is required to execute the primitive instruction At_Least_Gray, which colors a white node gray but leaves the color

of a black node unchanged. The obvious implementation is to code (White, Gray, Black) as (00, 01, 11) and to OR a 1 into the low-order bit. Our algorithm requires an assignment of gray to a node regardless of its previous color. However, mutator hardware for DLMSS can still be used by having the collector software recognize the following encoding: (00, {01, 11}, 10). Then an OR into the low-order bit turns both white and black into a representation of gray.

This representation allows the entire collector algorithm of the previous section to be implemented using the common instructions OR-to-Memory and AND-to-Memory. This is especially important if the color field is not separately addressable but is part of a word.

`M(N).Color := White;`	is	*M*(*N*) AND	00.
`M(N).Color := Gray;`	is	*M*(*N*) OR	01.
`White_to_Black(N);`	is	*M*(*N*) OR	10.

When appending to the free list, `M(I).Color := Black` can be replaced by `White_to_Black(I)`.

9. INCREMENTAL GARBAGE COLLECTION

If we delete the call to `Propagate_White` in the previous algorithm (or limit the propagation to a fixed depth), the algorithm becomes a good candidate for an incremental garbage collector. Very few nodes are whitened and hence the marking phase converges rapidly to recover a few nodes. Unfortunately, if we do not propagate the white color, garbage nodes can be lost.

There is a simple solution to this problem. The collector simply grays the sons of a white node—for they are also suspect and should be checked on the next cycle. Make sure that a white node is never grayed—otherwise, a garbage node pointing to itself will never be collected. The proof of the safety of this algorithm is identical to that of the previous section.

If there is garbage, then some garbage will be collected, although it is possible to devise scenarios that show that some specific node will never be collected. This is not a problem since, if free space is exhausted, the mutator will halt and the collector can collect as many nodes as needed before releasing the mutator.

```
-- The collector.
    for I in Index loop
        if M(I).Color = Gray
            then M(I).Color := White;
        end if;
    end loop;

    for I in Index loop
        if M(I).Color = White then
            for J in Sons loop
                if M( M(I).Son(J) ).Color ≠ White then
                    M( M(I).Son(J) ).Color := Gray;
                end if;
            end loop;
        end if;
    end loop;

    New_Marking_Phase;
    for I in Index loop
```

```
        if M(I).Color = White then
            Append_to_Free(I);
            M(I).Color := Black;
        end if;
    end loop;
```

10. CONCLUSION

New algorithms for on-the-fly garbage collection have been presented. The basic algorithm has a correctness proof that is much simpler than that of the DLMSS algorithm. The simplicity has paid off since we are able to obtain other improved algorithms whose proofs are simple modifications of the original proof. The relative performance of the algorithms needs to be investigated. Performance can be measured by simulations or even better by implementing the algorithms on computer systems such as the Intel iAPX-432.

ACKNOWLEDGMENTS

I would like to thank Tmima Olshansky for noting that the algorithm is impervious to Woodger's scenario. I am grateful to Amir Pneuli for his assistance in the formulation of the proofs. I am indebted to Rob Gerth for discovering two errors in the algorithm of Section 7. Working from the preliminary version, he has been able to discover other variations of these algorithms.

REFERENCES

1. COHEN, J. Garbage collection of linked data structures. *Comput. Surv. 13*, 3 (Sept. 1981), 341–367.
2. DIJKSTRA, E.W., LAMPORT, L., MARTIN, A.J., SCHOLTEN, C.S. AND STEFFENS, E.F.M. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM 21*, 11 (Nov. 1978), 966–975.
3. FRANCEZ, N. An application of a method for analysis of cyclic programs. *IEEE Trans. Softw. Eng.* SE-4(5), (1978), 371–378.
4. FRANCEZ, N. AND PNUELI, A. A proof method for cyclic programs. *Acta Inf. 9* (1978), 133–157.
5. GRIES, D. An exercise in proving parallel programs correct. *Commun. ACM 20*, 12 (Dec. 1977), 921–930.
6. KUNG, H.T. AND SONG, S.W. An efficient parallel garbage collection system and its correctness proof. In *Proceedings of the IEEE Symposium on Foundations of Computer Science* (Providence, R.I.). IEEE, New York, 1977, pp. 120–131.
7. LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.* SE-3, (1977), 125–143.
8. OWICKI, S. AND GRIES, D. An axiomatic proof technique for parallel programs I. *Acta Inf. 6* (1976), 319–340.
9. POLLACK, F.J., COX, G.W., HAMMERSTEIN, D.W., KAHN, K.C., LAI, K.K. AND RATTNER, J.R. Supporting Ada memory management in the iAPX-432, In Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems. *SIGPLAN Not.* (ACM) *17*, 4 (1982), 117–131.
10. PNEULI, A. The temporal semantics of concurrent programs. *Theor. Comput. Sci. 13* (1981), 45–60.

Received November 1982; revised October 1983; accepted October 1983