On-the-fly garbage collection: an exercise in cooperation.

by

Edsger W.Dijkstra *)

Leslie Lamport **)

A.J.Martin ***)

C.S.Scholten ****)

E.F.M.Steffens ***)

*) Burroughs, Plataanstraat 5, NL-4565 NUENEN, The Netherlands

**) Massachusetts Computer Associates Inc., 26 Princess Street, WAKEFIELD, Mass. 01880, U.S.A.

***) Philips Research Laboratories, EINDHOVEN, The Netherlands

****) Philips-Electrologica B.V., APELDOORN, The Netherlands

Abstract. A technique is presented which allows nearly all of the garbage detection and collection activity to be performed by an additional processor, operating concurrently with the processor carrying out the computation proper. Exclusion and synchronization contraints between the processors have been kept weak.

Key Words and Phrases: garbage collection, multiprocessing, cooperation between sequential processes with minimized mutual exclusion, program correctness for multiprocessing tasks.

CR Categories: 4.32, 4.34, 4.35, 4.39, 5.23.

20th of October 1975

On-the-fly garbage collection: an exercise in cooperation.

Introduction.

In any large-scale computer installation today, a considerable amount of time of the (general purpose) processor is spent on "operating the system". With the emerging advent of multiprocessor installations the question arises to what extent such "housekeeping activities" can be carried out concurrently with the computation(s) proper. Because the more intimate the interference, the harder the organization of the cooperation between the concurrent processes, the problem of garbage collection was selected as one of the most challenging --and, hopefully, most instructive!-- problems. (Our exercise has not only been very instructive, but at times even humiliating, as we have fallen into nearly every logical trap that we could possibly fall into.) In our treatment we have tried to blend a condensed design history --in order not to hide the heuristics completely-- with a rather detailed justification of our final solution. Whether the following solution, which is the result of many iterations, is of any economic significance, is a question beyond the scope of this paper.

We tackled the problem as it presents itself in the traditional implementation environment for pure LISP (and shall describe our solution in the usual terminology, leaving the natural generalizations to the reader). The data structure to be stored consists of a directed graph in which each node has at most two outgoing edges, more precisely: may have a left-hand outgoing edge and may have a right-hand outgoing edge. In the original problem statement, either of them or both could be missing; for the sake of homogeneity we follow the --not unusual-- practice of introducing a special purpose node called "NIL" and we represent an originally missing outgoing edge by an edge with the node called NIL as its target. As a result, each node has now always exactly two outgoing edges; the outgoing edges from NIL point to NIL itself. At any moment in time all the nodes must be "reachable" (via a directed path along the directed edges) from one or more fixed nodes --called "the roots"-- with a constant place in memory. The storage allocated to each node is constant in time and equal in size, viz. sufficient to accommodate two pointers --one for each outgoing edge-- pointing to the node's immediate successors. Given (the address of) a node, finding (the

address of) its left- or right-hand successor node can be regarded as an atomic, primitive action; finding its predecessor nodes, however, would imply a search through memory.

In the original problem statement, for a reachable node an outgoing edge could be deleted, changed or added. The effect of the special node NIL is that now all three modifications of the data structure take the same form, viz. the change of an outgoing edge of a reachable node. Note that such a change may turn a number of formerly reachable nodes into unreachable ones: they then become what is called "garbage". Changing an edge may direct the new edge towards a target node that was already reachable or towards a new node that has to be added to the data structure; such a new node --which upon creation has only NIL as successor node-- is taken from the so-called "free list", i.e. a linearly linked list of nodes that are currently not used for storing a node of the data structure. By linking the "free" nodes linearly --via their left-hand outgoing edge, say-- and introducing a special root pointing to the begin node of the free list, also the nodes of the free list can be regarded as reachable. By furthermore declaring that also the node called NIL is a root, we achieve our next homogenizing simplification: a change redirects for a reachable node one of its outgoing edges to a reachable node. (See Appendix.)

Garbage may arise anywhere in store, and it is the purpose of the so-called "garbage collector" to detect such disconnected and therefore obsolete nodes and to append them to the free list. In classical LISP implementations the computation proceeds until the free list is exhausted (or nearly so). Then the computation proper comes to a grinding halt, during which the processor is devoted to garbage collection. Starting from the roots, all reachable nodes are marked --because we have made the nodes of the free list reachable from a special root, nodes of the free list (if any) will in our case be marked as well-- . Upon completion of this marking phase, all unmarked nodes can be concluded to be garbage and are appended to the free list, after which the computation proper is resumed.

The minor disadvantage of this arrangement is the central processor time spent on the collection of garbage; its major disadvantage is the unpredictability of these garbage collecting interludes, which makes it hard

to design such a system so as to meet real time requirements as well. It was therefore tempting to investigate whether a second processor --called "the collector"-- could collect garbage on a more continuous basis, concurrently with the activity of the other processor --for the purpose of this discussion called "the mutator"-- which would be dedicated to the computation proper. We have imposed upon our solution a few constraints (compare [2]). The interference between collector and mutator should be minimal --i.e. no highly frequent mutual exclusion of elaborate activities, as this would defy our aim of concurrent activity-- , the overhead on the activity of the mutator (as required for the cooperation) should be kept as small as possible, and, finally, the ongoing activity of the mutator should not impair the collector's ability to identify garbage as such as soon as logically possible. (One synchronization measure is evidently unavoidable: when needing a new node from the free list, the mutator may have to be delayed until the collector has appended some nodes to the free list. This is the now traditional producer/consumer coupling; in the context of this article it must suffice to mention that this form of synchronization can be achieved without any need for mutual exclusion.)

Preliminary investigations.

A counterexample taught us that the goal "no overhead for the mutator" is unattainable. Suppose that nodes A and B are permanently reachable via a constant set of edges, while node C is reachable only via an edge from A to C . Suppose furthermore that from then on the mutator performs with respect to C repeatedly the following sequence of operations:

1) making an outgoing edge from B point to C
2) deleting the edge from A to C
3) making an outgoing edge from A point to C
4) deleting the edge from B to C .

The collector, which observes nodes one at a time, will discover that A and B are reachable from the roots, but never needs to discover that C is reachable as well: while A is observed by the collector, C may be reachable via B only, and the other way round. We may therefore expect that the mutator may have to mark in some way target nodes of changed edges.

Marking will be described in terms of colours. When we start with all

nodes white, and, furthermore, the combined activity of collector and mutator can ensure that eventually all reachable nodes become black, then all white nodes can be identified as garbage. For each repetitive process --and the marking process certainly is one-- we have always two concerns (see [1]): firstly we must have a monotonicity argument on which to base our proof of termination, secondly we must find an invariant relation which, initially true and not being destroyed, will still hold upon termination. For the monotonicity argument we suggest (fairly obviously)

during marking each node will darken monotonically.

For the invariant relation --a relation which must be satisfied both before and after the marking cycle-- we must generalize initial and final state of the marking process and our first guess was (perhaps less obvious, but not unnatural)

P1: during marking there will be no edge pointing from a black node to a white one.

Additional action is then required from the mutator when it is about to introduce an edge from a black node to a white one: just placing it would cause a violation of P1 . The monotonicity requirement tells us, that the black source node of the new edge has to remain black, and, therefore, P1 tells us that the target node of the new edge cannot be allowed to remain white. But the mutator cannot make it just black, because that could cause a violation of P1 between that new target node and its immediate successors. For that reason grey has been introduced as intermediate colour and the overhead considered for the mutator was

A1: when introducing an edge, the mutator shades its target node.

Note 1. Shading a node is defined to make a white node grey and to leave the colour of a grey or a black node unchanged. (End of note 1.)

The choice of the invariant relation P1 has been sufficient for a solution --not published here-- with a rather coarse grain of interleaving (in which, for instance, A1 was assumed to be available as a single, indivisible action). We could not use it, however, as a stepping stone towards a solution that allowed a finer grain of interleaving, because total absence

of an edge from a black node to a white one was a stronger relation than we managed to maintain. We could, however, retain the notion "grey" as "semi-marked", more precisely, as representing our unfulfilled marking obligation: as before, the marking activity of the collector remains localized at grey nodes and their possibly white successors.

A coarse-grained solution.

In our unpublished solution we made essential use of the fact that after the collector had initialized the marking phase by shading all roots, the validity of P1 allowed us to conclude that the existence of a white reachable node implied the existence of a grey node (even of a grey reachable node, but the reachability of such an existing grey node was not essential). A weaker relation from which the same conclusion can be drawn is

P2: during the marking cycle (that the collector has initialized by shading all roots) there exists for each white reachable node a so-called "propagation path", leading to it from a (not necessarily reachable) grey node, and consisting solely of edges with white targets (and, as a consequence, without black sources).

Note 2. In the absence of edges from a black node to a white one, relation P2 is clearly satisfied. (End of note 2.)

The existence of edges from a black node to a white one is restricted by

P3: during the marking cycle only the last edge placed by the mutator may lead from a black node to a white one.

Note 3. In the absence of black nodes, P3 is trivially satisfied. (End of note 3.)

When the mutator redefines an outgoing edge of a black node, it may direct it towards a white node: this new edge from a black node to a white one is permitted by P3 , but because the previously placed one could still exist and be of the same type, we consider for the mutator the following indivisible action:

A2: shade the target of the edge previously placed by the mutator and redirect for a reachable node one of its outgoing edges towards an already reachable node.

Note 4. For the very first time the mutator changes an edge we can assume that, for lack of a previously placed edge, the shading will be suppressed or an arbitrary reachable node will be shaded; the choice does not matter for the sequel. (End of note 4.)

Action A2 has been carefully chosen in such a way that it leaves P3 invariant; it leaves, however, the stronger relation P2 and P3 invariant as well.

Proof. The action A2 cannot introduce new reachable nodes; it, therefore, does not introduce new white ones for which extra propagation paths must exist. If the node whose successor is redefined is black, its outgoing edge that may have disappeared as a result of the change was not part of any propagation path, and the edges of the old propagation paths will be sufficient to provide the new propagation paths. (Possibly we don't need all of them as a result of the shading and/or white reachable nodes having become unreachable.) If the node whose successor is redefined was white or grey to start with, it will become at most grey and the resulting graph has no edge from a black node to a white one --if one existed, it has been removed by the shading and the change has not introduced a new one-- and (see Note 2) P2 holds upon completion. (End of proof.)

We have now reached the stage where we can describe our first collector, which repeatedly performs the following program. (Our bracket pairs "if...fi" and "do...od" delineate our alternative and repetitive constructs respectively (see [1]), comments have been inserted between braces and labels have been inserted for the discussion.) The program has two local integer variables i and k ; the nodes in memory are assumed to be numbered from 0 through M - 1 .

```
marking phase:
begin {there are no black nodes}
  C1: "shade all the roots" {P2 and P3};
      i:= 0; k:= M;
  marking cycle:
      do k > 0 → {P2 and P3}
            if C2: "node nr. i  is grey" →
                        k:= M;
                  C3: "shade the successors of node nr. i  and make node
                        nr. i  black" {P2 and P3}
             [] C2: "node nr. i  is not grey" →
                        k:= k - 1 {P2 and P3}
            fi {P2 and P3};
            i:= (i + 1)mod M
      od {P2 and P3 and there are no grey nodes, hence all white nodes
          are garbage}
end;
appending phase:
begin i:= 0;
      do i < M → {a node with a number  < i  cannot be black;
                  a node with a number  ≥ i  cannot be grey,
                  and is garbage, if white}
            if C2: "node nr. i  is white" →
                  C4: "append node nr. i  to the free list"
             [] C2: "node nr. i  is black" →
                  C5: "make node nr. i  white"
            fi;
            i:= i + 1
      od {there are no black nodes}
end
```

The indivisible actions of the collector --between the execution of which actions A2 of the mutator may occur-- are

1) "shading of a single root" (from which C1 is composed: the order in which the roots are shaded is irrelevant)
2) establishing the current colour of node nr. i (labeled "C2").
3) the total actions C3, C4 (see, however, the Appendix) and C5.

Remark 1. With a more elaborate administration local to the collector --a list of grey or possibly grey nodes-- a probably much more efficient marking phase could have been designed. For the sake of simplicity we have not done so. (End of remark 1.)

We observe that (even independent of the colour of node nr. i !) action C3: "shade the successors of node nr. i and make node nr. i black" can never cause a violation of P2 and P3 : the shading of the successors can never do any harm, as a result of the shading the outgoing edges of node nr. i are no longer needed for a propagation path, and making node nr. i black maintains the existence of the propagation paths needed without introducing an edge from a black node to a white one.

The state characterized by the absence of grey nodes, which implies on account of P2 that all white ones are garbage and that all reachable ones are black, is stable, because the absence of white reachable nodes prevents the mutator from introducing grey nodes, and the absence of grey nodes prevents the collector from doing so. Because, when a grey node is encountered, k is reset to M , the marking cycle can only terminate with a scan past all nodes, during which no grey node is encountered. Because the mutator leaves grey nodes grey, no grey node can have existed at the beginning of such a scan, i.e. the stable state must have been reached at that moment. Termination of the marking cycle is guaranteed because of the monotonicity of the colouring history of each node and because of the fact that resetting k to M is always accompanied by effective darkening of at least one node (nr. i to be precise).

When the appending phase starts, all reachable nodes are black and all white nodes are garbage. Note that the existence of black garbage is not excluded. The appending phase deals with each node in turn: as long as it has not been dealt with (i.e. has a number $\geq i$) it cannot change colour: if black, it remains black because the mutator can only shade it, and if it is white, it is garbage and, by definition, the mutator won't touch it. As soon as it has been dealt with (i.e. has a number $< i$), it has been white and can at most have been shaded by the mutator. Black garbage at the beginning of the appending phase will not be appended during that appending phase, it will only be made white; during the next marking phase it will remain white,

and the next appending phase will indeed append it. Therefore, no garbage, once created, will escape being collected.

A solution with a fine-grained collector.

We would like to break C3 open as a succession of five indivisible subactions, say (m1 and m2 being local variables of the collector):

C3.1: m1:= number of the left-hand successor of node nr. i ;

C3.2: shade node nr. m1 ;

C3.3: m2:= number of the right-hand successor of node nr. i ;

C3.4: shade node nr. m2 ;

C3.5: make node nr. i black

None of the actions C3.1 , C3.2 , C3.3, and C3.4 can cause violation of P2 and P3 . The actions C3.1 and C3.3 cannot do so because they leave no trace in common memory, and the actions C3.2 and C3.4 cannot do so because shading cannot do so. Besides that, because shading of a node commutes with any number of actions A2 , we have, by the time that the collector starts with C3.5, a state as if the shading of node nr. m1 had been part of C3.1 and the shading of node nr. m2 had occurred simultaneously with C3.3 . Without loss of generality we can continue our discussion as if "shade left-hand successor" and "shade right-hand successor" are available as indivisible actions. The problem, however, lies with C3.5 : can we safely make node nr. i black? Note that neither m1 , nor m2 needs still to be one of its successors: m1 and m2 even never need to have been its left- and right-hand successor simultaneously! A more thorough study of the mutator, however, reveals that it is safe.

Proof. During the marking phase we define a changing set of edges to which we give --in order to avoid false connotations-- the rather meaningless name "dodo-edges". (Note that we only define the set of dodo-edges for our benefit. The mutator and collector would have a hard time if they had to update it explicitly: in the jargon the term "ghost variable" is sometimes used for such an entity.) The set of dodo-edges is defined as follows as a function of the evolving computations:

1) at the beginning of the marking phase the set of dodo-edges is initialized with all the edges with a grey target

2) each time a white node becomes grey, all its incoming edges (that were

not already a dodo-edge) are added to the set of dodo-edges

3) when the action A2 , seen as a replacement of an outgoing edge, removes a dodo-edge --or an edge that, according to the second rule, would have become one as a consequence of A2's shading act-- the new edge that replaces it is also a dodo-edge: it "inherits the dodo-ness" of the edge it replaces.

The above rules imply that a dodo-edge is never needed for a propagation path. The last one all by itself implies that once the left-hand outgoing edge of a node is a dodo-edge, it will remain so, no matter how often redirected by the mutator, until the end of the marking phase, and that the same holds for the right-hand outgoing edge. In short: when, since the beginning of the marking phase, a given node has had a grey left-hand successor and has had a grey right-hand successor, it has two outgoing dodo-edges and making it black will never cause violation of P2 . It won't violate P3 either: if it has a white successor, the corresponding edge must have been the last one placed by the mutator (it can therefore have at most one white successor) and that edge from a black node to a white one is the one explicitly allowed by P3. (End of proof.)

The above argument sheds another light upon the action C3. Instead of waiting until it has seen both successors of a node to be non-white, it forces termination of that waiting process by shading its successors itself. It refrains from shading the successors of a white node, as that would defeat garbage detection, it also refrains from shading the successors of a black node (although such a black node could have a white successor) because that is unnecessary. It is in this sense that the grey nodes represent our unfulfilled marking obligation.

Note 5. In breaking up C3 we have placed C3.5 "make node nr. i black" at the end. As making a node black commutes with all other actions A2 and C3.1 through C3.4 , we could also have placed it at the beginning, before dealing (in some order) with the successors; P2 and P3 could then be violated temporarily. (End of note 5.)

A solution with a fine-grained mutator as well.

From the above it is obvious that no harm is done if at random moments a daemon would shade a reachable node. We now assume a very friendly daemon

that between any two successive actions A2 of the mutator shades the target node of the last placed edge. For the initial state of an action A2 during a marking cycle, we can now assert (besides P2 and P3) the absence of an edge from a black node to a white one, regardless of the question whether the last shading by the friendly daemon took place during the current marking phase, or earlier. As a result, the proof that A2 leaves P2 and P3 invariant is now also valid if A2 does not shade at all! Thanks to the daemon, it does not need to do so anymore! We can therefore replace A2 by the succession of the following two separate indivisible subactions:

"redirect for a reachable node an outgoing edge towards a reachable node" ;

"shade the target of the edge just placed".

Remark 2. The detailed implementation of what we have described as "a grain of interleaving" falls very definitely outside the scope of this paper: many techniques --even allowing concurrent access to the same unit of information-- are possible (see [3], [4]). (End of remark 2.)

In retrospect.

It has been surprisingly hard to find the published solution and justification. It was only too easy to design what looked --sometimes even for weeks and to many people-- like a perfectly valid solution, until the effort to prove it to be correct revealed a (sometimes deep) bug. Work has been done on formal correctness proofs ([5], [6]), but a shape that would make them fit for print has, to our tastes, not yet been reached. Hence our informal justification (which we do not regard as an adequate substitute for a formal correctness proof!). Whether its stepwise approach --which this time seems to have been successful in reducing the case analyses-- is more generally applicable, is at the moment of writing still an open question.

When it is objected that we still needed rather subtle arguments, we can only agree whole-heartedly: all of us would have preferred a simpler argument! Perhaps we should conclude that constructions that give rise to such tricky problems are not to be recommended. One firm conclusion, however, can be drawn: to believe that such solutions can be found without a very careful justification is optimism on the verge of foolishness.

History and acknowledgements. (As in this combination this is our first exercise in international and inter-company cooperation, some internal credit should be given as well.) After careful consideration of a wider class of problems the third and the fifth authors selected and formulated this problem and did most of the preliminary investigations; the first author found a first solution during a discussion with the latter, W.H.J.Feijen and M.Rem. It was independently improved by the second author --to give the free list a root and mark its nodes as well, was his suggestion-- and, on a suggestion made by Jack Mazola, by the first and the third author. The first and the fourth merged these embellishments, but introduced a bug that was found by N.Stenning and M.Woodger [7]. The final version and its justification are the result of a joint effort of the four authors in the Netherlands. The active and inspiring interest shown by David Gries is mentioned in gratitude.

References.

1. Dijkstra, Edsger W., Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Comm. ACM 18, 8 (Aug. 1975), 453-457.
2. Steele Jr.,Guy L., Multiprocessing Compactifying Garbage Collection. Comm. ACM 18, 9 (Sep. 1975), 495-508.
3. Lamport, Leslie. On Concurrent Reading and Writing. (Submitted to the Comm. ACM.)
4. Scholten, C.S., Private Communication
5. Gries, David, An Exercise in Proving Parallel Programs Correct. (Submitted to the Comm.ACM.)
6. Lamport, Leslie, Report CA-7508-0111, Massachusetts Computer Associates, Inc.
7. Woodger, M., Private Communications.

Appendix.

Here we give an example of how the free list and the operations such as taking a node from or appending a node to the free list can be implemented. We consider the nodes of the free list ordered according to "age". For each node in the free list, the right-hand successor is NIL, the left-hand successor is NIL for the youngest node and is the next-younger one for the others. We have a root called TAKE , its left-hand successor and its right-hand successor are both the oldest free node; we have a second root called APP , whose left-hand and right-hand successor are both the youngest free node.

Taking a free node --and making it the left-hand successor of some reachable node X , say-- can be done in the following steps (shown in a hopefully self-explanatory notation):

```
X.left:= TAKE.left;
TAKE.left:= TAKE.right.left;
TAKE.right.left:= NIL;
TAKE.right:= TAKE.left
```

(All four actions should follow the shading convention chosen.)

To append, say, node Y --in action C4-- could be done by:

```
Y.left:= NIL; Y.right:= NIL;
APP.left:= Y;
APP.right.left:= Y;
APP.right:= APP.left
```

When a minimum of two free nodes is maintained, the collector that appends is certain only to deal with nodes that are left alone by the mutator, and the action C4 need not be regarded as a single, indivisible action, but is trivially allowed to be broken up in the above subactions. The synchronization guaranteeing the lower bound for the length of the free list is here supposed to be implemented by other, independent means.

20th of October 1975