IMPROVED EFFECTIVENESS FROM A REAL TIME LISP GARBAGE COLLECTOR

Jeffrey L. Dawson

WANG Laboratories, Inc.
Lowell, Massachusetts 01851

INTRODUCTION

This paper describes a real-time garbage collection algorithm for list processing systems. We identify two efficiency problems inherent to real-time garbage collectors, and give some evidence that the proposed algorithm tends to reduce these problems. In a virtual memory implementation, the algorithm restructures the cell storage area more compactly, thus reducing working sets. The algorithm also may provide a more garbage-free storage area at the end of the collection cycle, although this claim really must await empirical verification.

Garbage collection

List processing systems comprise two related processes. The mutator allocates cells from free storage and links them together to form data structures representing the computation underway. When the mutator drops a data structure it no longer needs, the underlying cells remain in the system but are no longer accessible.

In order to prevent the mutator from running out of cells to allocate and thus blocking computation, a collector process gathers the inaccessible or garbage cells and returns them to free storage for reuse by the mutator. A number of collector algorithms have been proposed with a variety of performance objectives. [Cohen81] provides a recent survey of garbage collectors.

Copying garbage collectors

The collector which we will discuss is a copying collector. For classical copying garbage collection [Fenichel69, Cheney70], the cell storage is divided into two semispaces whose roles are reversed with each garbage collection cycle. When collection begins, the semispace labeled TOSPACE is empty, and all accessible and garbage cells are in the semispace labeled FROMSPACE. The collector copies the accessible cells out of FROMSPACE into TOSPACE. When this process is completed, FROMSPACE contains only inaccessible cells, while TOSPACE contains all and only accessible cells. Now computation can resume with the mutator allocating cells from the garbage free TOSPACE until it is again necessary to collect.

Baker's real-time collector

The classical copying collector is not well suited to real-time applications, since the mutator is subject to unpredictable garbage collection interludes whose duration is proportional to the number of accessible cells in the system. In order to meet real-time constraints, [Baker78] distributes the copying function among the mutator's primitives. For each cell the mutator allocates, it must copy a number of cells bounded by a constant, k . By carefully choosing k, one can be assured that the mutator will not run out of cells to allocate. Moreover, the mutator's primitive operators have well defined bounds on their execution times.

The trick is in maintaining the consistency of the data structures as the collector copies them and the mutator alters them. This is accomplished by giving the mutator the impression that garbage collection has completed and that it is working in the new semispace, although copying is in fact still going on. The mutator's access to the data structures is through a set of registers holding pointers to the root cells of the data structures. When collection begins, the cells pointed to by the registers are immediately copied into TOSPACE.

Thereafter, all new cells will be allocated in TOSPACE. Also, the data access primitives, CAR and CDR, are required to copy their returned values into TOSPACE. Thus the mutator's illusion that garbage collection is complete will be maintained. Meanwhile, the collector linearly scans TOSPACE following any pointer into FROMSPACE and copying the cell into TOSPACE. When all TOSPACE has been scanned, garbage collection is complete.

REAL-TIME PROBLEMS

Although Baker has cleverly overcome the major problem of getting the two processes to run in parallel, one process may interfere with the other's most efficient operation. We identify two inefficiencies in real-time garbage collection which could be viewed as penalties for concurrent operation. Our collector, which we will describe later, is intended to reduce these inefficiencies.

Floating garbage

All garbage collectors have a two-phase, cyclic action. They identify the inaccessible cells, then reclaim them. If the collector and mutator are processing in parallel, then during each of the collector cycles, the mutator continues to generate garbage cells. If these are not identified, they will remain in the system as garbage until discovery during some subsequent cycle. We will call these lingering cells "floating garbage". Analysis of a "mark and sweep" garbage collection algorithm in [Wadler76] suggests that the average performance is close to the worst case in which none of the floating garbage is reclaimed until the next cycle. An effective collector would minimize the amount and longevity of floating garbage.

The floating garbage in Baker's collector consists of those TOSPACE cells which are inaccessible immediately at the end of the collection cycle. They are of two different origins. The first type are cells whose access was dropped after they were copied. The second type are cells allocated after the beginning of the cycle whose access was dropped before the end of the cycle.

Asynchronous copying

Another important function of copying garbage collectors in virtual memory systems is the restructuring of the accessible data for more compact storage and easier access. This has lead to the practice of "linearization" in an attempt to keep the cells of each data structure on as few pages as possible, thus reducing the average working set during program execution. In CDR-linearizing, whenever possible the CDR pointer of a cell points to the next physical cell in storage. This scheme can be enhanced through "CDR-coding" to yield even more compact storage [Clark77].

An "asynchronous copy" occurs when cells must be copied as a result of the mutator's action, thereby confounding the collector's attempts at linearization. The collector could be linearly copying a list when the mutator stuffs a few unrelated cells in the middle; or the collector finds that a few cells, which would fit nicely into its current list, were already copied elsewhere by the mutator.

Baker's collector relieves this problem somewhat by forcing the mutator to allocate cells at a different point in cell storage than that to which cells are being copied. However, asynchronous copying still occurs when CAR or CDR would return a cell in FROMSPACE, or when either of the pointers within a newly allocated cell are in FROMSPACE.

OUR COLLECTOR

The differences between our collector and Baker's are the following. We redirect the registers into TOSPACE one at a time, rather than immediately at the beginning of the collection cycle. The registers are not forced into TOSPACE, but drift there with the tide of copied cells. Asynchronous and linear copying are performed at different locations in cell storage. These simple changes appear to redound in improvements in the areas of floating garbage and linearization of storage. In this section, we will describe the algorithm. CDR-coding is not employed in the algorithm described, although it could be added with an accompanying increase in complexity. A pseudo-Algol representation is listed in the Appendix.

Cell storage

The cell storage is again separated into two semispaces whose roles are reversed just before each collection cycle begins. All accessible and inaccessible cells are in FROMSPACE at the beginning of the collection cycle, while TOSPACE is empty. Five pointers into cell storage are maintained throughout the collection process:

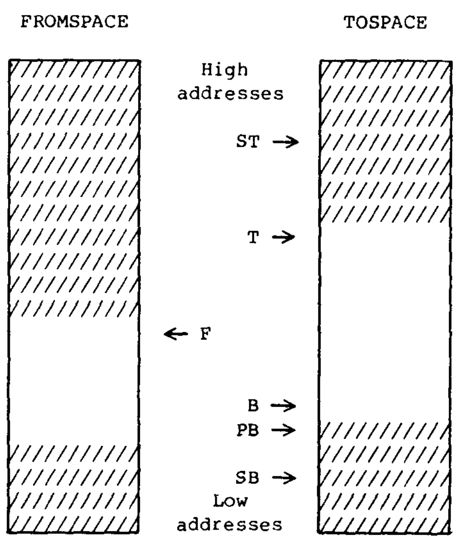
F	top free storage area in FROMSPACE,
T	top free storage area in TOSPACE,
B	bottom free storage area TOSPACE,
PB	linearizing pointer below B,
SB	scanning pointer below B,
ST	scanning pointer above T.

At the beginning of the cycle; SB, PB, and B point to the bottom cell of TOSPACE, while T and ST point to the topmost cell. F points to the cell that T pointed to before the cycle. The nonfree cells of

TOSPACE are separated into two regions with the free storage between. The cells below B have been copied linearly out of FROMSPACE. The cells above T have been put there as a result of the mutator's action. They were either newly allocated or asynchronously copied out of FROMSPACE. The TOSPACE pointers converge on the middle of the TOSPACE while maintaining the relationships $SB \leq PB \leq B < T \leq ST$. Collection is complete when $SB = PB = B$, $T = ST$, and all registers point into TOSPACE.

Figure 1. Diagram of cell storage area.

(///// indicates nonfree cells)



Forwarding addresses

When a cell is copied out of FROMSPACE, its CAR pointer is redirected to the TOSPACE version of the cell, and serves as the forwarding address. The test of a FROMSPACE cell to determine whether it is an authentic data cell or a forwarding address holder is to determine into which semispace its CAR pointer is directed. If it points into TOSPACE, it is a forwarding address.

Cell allocation

We assume that there is some free storage remaining in FROMSPACE, and that the mutator can allocate cells at either F or T. This is not a serious restriction in a virtual memory implementation where an additional page of free storage can be added at T or F as needed. A clairvoyant mutator would allocate at F all the cells that it knows it will drop during the course of garbage collection so that they will be left behind when FROMSPACE is abandoned at the end of the cycle, in which case, there would be no floating garbage of the second type.

In the absence of such clairvoyance, we try to allocate cells in FROMSPACE in the hope that they will become inaccessible before the end of the cycle. If they are not dropped, they will have to be copied. The maintenance of the forwarding address scheme puts a limitation on this allocation policy. If the CAR pointer of a newly created cell is directed into TOSPACE, the cell must be allocated in TOSPACE to avoid the CAR pointer being mistaken for a forwarding address to a copied cell.

The collection process

Always, the collector's first choice is to continue the linearization at B when possible. The linearization pointer PB is either equal to B or points to the last cell copied if the CDR of that cell has not been copied. The collector keeps copying the CDR of PB at B as long as it exists and has not already been copied. When this fails the collector looks for a single cell to copy at B as a new seed to the linearization process.

The collector attempts, in order, three methods for obtaining a new seed. It scans the pointers below B, examines the registers, and scans the pointers above T. The order of these attempts has effects upon performance which we will examine later. The scan pointer SB moves toward B until it finds a CAR pointer to a data cell in FROMSPACE. (Because of the linearization in this area of cell storage, the collector need not examine CDR pointers -- they are all directed into TOSPACE.) That cell is copied at B and linearization resumes. As SB scans it rewrites pointers into FROMSPACE with forwarding addresses if possible. If SB is equal to B, the registers are searched sequentially for a pointer to a data cell in FROMSPACE to serve as the new seed. If all registers are directed into TOSPACE, then the scan pointer ST is moved toward T until a new seed is found. When all three methods fail to provide a new seed, the collection cycle is finished.

Interaction of collector and mutator

In order to meet the real-time constraints, we adopt Baker's strategy of setting a scanning quota Q. For each new cell allocated, the collector examines a total of exactly Q cell pointers during the linearization process and the new seed search. The data access primitives CAR and CDR are asked to assist the collector by updating some of the pointers to forwarding address cells. Any CDR pointer is updated to point to a data cell, while only TOSPACE cells have their CAR pointers updated because of the forwarding address test. By convention the mutator primitives never return pointers to forwarding address cells. In this way one is assured that the arguments to any succeeding primitive and the registers will point to data cells and need not be tested.

In order to ensure correct concurrent operation of the mutator and collector, the following conditions are to be maintained throughout the collection cycle.

Condition 1: Any pointer in TOSPACE lying below SB or above ST points into TOSPACE.

We must be certain to maintain the self-referential quality of that portion of TOSPACE which lies outside the scan pointers; otherwise, we may end up with invalid pointers into FROMSPACE after the cycle. The only way a problem of this nature could arise is if one of the TOSPACE cells below S had one of its pointers redirected by the primitives, RPLACA or RPLACD, so special attention must be paid to these primitives. Whenever a scanned pointer is being redirected into FROMSPACE, the intended pointee is copied into TOSPACE at T before the redirection.

Condition 2: Any CAR pointer in FROMSPACE pointing into TOSPACE is a forwarding address for a copied cell.

We must exercise caution in the application of RPLACA and CONS, lest we create false forwarding addresses. We must not apply RPLACA to a FROMSPACE cell with a pointer into TOSPACE. So, RPLACA copies the FROMSPACE cell into TOSPACE at T before performing the redirection. Whenever a cell is created whose CAR pointer is directed into TOSPACE, the cell is allocated at T in TOSPACE.

Termination

If we view the collector's activity as tracing pointers and ensuring that the cell pointed to is in TOSPACE, the termination condition for the collection cycle is that all traceable pointers have been traced. If we choose the scanning quota Q to be larger than 2, we can be certain that the collection cycle will indeed terminate -- essentially, because the collector is able to trace the existing pointers faster than the mutator can add new ones. Suppose there are N accessible pointers at the beginning of the cycle, i.e., N/2 accessible cells. After N/Q cells have been allocated, N pointers will have been traced and there will be at most 2(N/Q) traceable pointers, as yet untraced. After another $2N/Q^2$ more cells have been allocated, there will be at most $N(2/Q)^2$ untraced pointers. Iterating this argument we see that the number of untraced pointers falls below one, and collection must be complete after

$$(N/2)\ [(2/Q) + (2/Q)^2 + \ldots + (2/Q)^s]$$

$$= \frac{N}{Q}\left[\frac{1-(2/Q)^s}{1-(2/Q)}\right] \approx \frac{N-1}{Q-2}$$

cells have been allocated, where s is equal to $\log_{Q/2} N$.

PERFORMANCE ANALYSIS

In this section, we will discuss the effectiveness of our collector, particularly in comparison to Baker's. Our algorithm is better at linearizing the copied data and may reduce the amount of floating garbage, although the latter is not as clearly established.

Linearization

Whenever cells from two different data structures are copied at B, the structures become intertwined as they are copied, thus diluting one another's locality of reference. Choosing one linearization seed at a time tends to reduce this mingling. Choosing the new seed by scanning the bottom of TOSPACE, we tend to copy the CAR cells immediately after the CDR chain of the list. This also improves the locality of reference. In fact, this approach would improve the classical non-real-time copying collectors.

By performing asynchronous copying and cell allocation at T, we avoid breaking up CDR chains under construction and introducing unrelated seeds. There will be very little copying done when the top of TOSPACE is being scanned -- mostly just redirection of CDR pointers through forwarding addresses. The cells copied will be those whose sole connection with TOSPACE is through a cell copied at T as a result of one of the replacement primitives, RPLACA or RPLACD. If one were to scan the top of TOSPACE before the registers in search of a new linearization seed, the parts of data structures attacked by RPLACA or RPLACD would be closer together. Going even further the asynchronous copy resulting from these primitives could be done at B with the same effect. Moreover, scanning the top of TOSPACE would be easier since we would only need to examine the CDR pointers. Since the replacement primitives are relatively infrequent, the degradation of the linearity should be small.

Asynchronous copying

There is another reason for trying to avoid asynchronous copying aside from not thwarting the collector's efforts at linearization. In an interactive application, some or all of the linear copying could be carried out during keyboard waits with no performance loss to the mutator. By reducing the asynchronous copying we can increase the possibilities for this "free" copying.

An asynchronous copy will occur in Baker's collector whenever an argument of CAR, CDR or CONS points to a cell in FROMSPACE. An asynchronous copy occurs in our collector only when the first argument of CONS points into FROMSPACE and for certain combinations of arguments in RPLACA and RPLACD -- approximately half the occurrences of RPLACA and one-fourth the occurrences of RPLACD during the cycle. Data from [Clark79, Table I] on the frequencies of these primitives within three large LISP programs suggests that by shifting the forced copying from CAR and CDR to the less frequent RPLACA and RPLACD, the amount of the asynchronous copying will be reduced substantially. Indeed, unless the scanning quota can be made fairly large, most of the copying in Baker's collector is asynchronous.

Floating garbage

The problem caused by floating garbage in a virtual memory implementation is that the number of pages required at the beginning of each cycle will be larger than need be, since the floating garbage cells will dilute the accessible cells.

With copying collectors, once the root cell of the data structure has been copied, the entire structure will be copied, even though access to the structure has been dropped as it is being copied or as another structure is being copied. By seeding the copying process out of only one register at a time, we increase our chances of leaving the garbage in FROMSPACE.

Register effects

We can further pursue this garbage abandonment policy through the order in which we scan the registers. Because of sharing of accessible cells among the registers, one would expect the number of cells copied per register to decrease as each register is scanned. Moreover, in some LISP implementations, one of the registers holds the list of bound global variables and their values. If that register is scanned first, a substantial portion of the copying will be done while the remaining registers are still freely dumping garbage into FROMSPACE.

In addition, one could write programs in such a way that lists which are likely to be shortlived are accessible through the registers scanned later, while the more durable data is accessible through registers scanned earlier in the cycle.

In summary, one should scan the registers pointing to the eternal data first and the ephemeral registers last. Thus, lists of arguments for function calls would go in the last scanned registers, and globally bound variables would go in the first scanned registers.

It is felt that there is a high early rate of droppage among the newly allocated cells. If cells are allocated in FROMSPACE early in the cycle, the shortlived cells can be left behind. While if cells are allocated in TOSPACE late in the cycle, they will not have to be copied. Choosing the allocation semispace by the CAR pointer effects just such a policy. Cell allocation will follow the migration of cells out of FROMSPACE into TOSPACE because of the growing preponderance of CAR pointers into TOSPACE. The effectiveness of this policy will depend on the longevity of newly allocated cells in relation to the length of the garbage collection cycle.

Although we may do better than Baker in the proportion of garbage that we leave in FROMSPACE, our collection cycle is generally longer, since every cell which was allocated in FROMSPACE and is accessible at the end of the cycle had to be copied, rather than allocated directly into TOSPACE. Because of the lengthened cycle, more garbage may be generated. The combined effect of these two tendencies is hard to judge. In the worst case, our cycle would last approximately $(N-1)/(Q-2)$ cell allocations; while the worst case for Baker's collector is N/Q cell allocations.

Conclusions

In summary, our algorithm does quite well in the linearization of the data. The only flaws come from RPLACA and RPLACD which would have destroyed the linearity of storage even if collection had finished instantly. The issue of floating garbage is not quite as easy to decide, and would require experimental data to clarify. In an earlier version of this algorithm, we had used a "semispace bit" associated with each register. If this bit is on, the register may point only into TOSPACE. As the data accessible out of each register is copied the semispace bit is turned on. This prevents any recidivist registers creeping back into FROMSPACE; and the collection cycle is shorter, although cell storage is not as well linearized. It is our feeling that the benefits of linearization probably outweigh the inefficiencies of the floating garbage. A successful resolution of this and other performance issues will have to wait for implementation. Neither our algorithm nor Baker's has been implemented and analyzed as yet.

Acknowledgment

This algorithm was developed as part of an effort to put together a LISP system at Wang Laboratories for in house use. I would like to thank Dan Corwin for his patience and perseverance in that effort. I would also like to thank Henry Baker for his thoughtful comments and encouragement.

REFERENCES

Baker77 Baker, H.G., Jr. and C. Hewitt, "The incremental garbage collection of processes," Memo No. 454, Artificial Intelligence Laboratory, M.I.T., Cambridge, MA, December 1977.

Baker78 Baker, H.G., "List processing in real time on a serial computer," Communications of the ACM, Vol. 21, No. 4, April 1978, pp 280-294.

Cheney70 Cheney, C.J., "A nonrecursive list compacting algorithm," Communications of the ACM, Vol. 13, No. 11, November 1970, pp. 677-678.

Clark77 Clark, D.W. and C.C. Green, "An empirical study of list structure in LISP," Communications of the ACM, Vol. 20, No. 2, February 1977, pp. 78-87.

Clark79 Clark, D.W., "Measurements of dynamic list structure use in LISP," IEEE Transactions on Software Engineering, Vol. SE-5, No. 1, January 1979, pp. 51-59.

Cohen81 Cohen, J., "Garbage collection of linked data structures," ACM Computing Surveys, Vol. 13, No. 3, September 1981, pp. 341-367.

Fenichel69 Fenichel, R.R. and J.C. Yochelson, "A LISP garbage-collector for virtual-memory computer systems," Communications of the ACM, Vol. 12, No. 11, November 1969, pp. 611-12.

Hood81 Hood, R. and R. Melville, "Real-time queue operations in pure LISP," Information Processing Letters, Vol. 13, No. 2, November 1981, pp. 50-54.

Steele75 Steele, G.L. Jr., "Multiprocessing compactifying garbage collection," Communications of the ACM, Vol. 18, No. 9, September 1975, pp. 495-508.

Wadler76 Wadler, P. L., "Analysis of an algorithm for real-time garbage collection," Communications of the ACM, Vol. 19, No. 9, November 1976, pp. 491-500.

APPENDIX: The collector algorithm

We present a pseudo-Algol implementation of our collector algorithm. We have tried to keep the notation very close to that of [Baker78] to facilitate comparison. This version of the algorithm does not use CDR-coding although that could be arranged with the accompanying complexity.

Symbol	Type	Usage
B	pointer	Bottom of free storage in TOSPACE
PB	pointer	Linearization pointer
T	pointer	Top of free area in TOSPACE
F	pointer	Top of free area in FROMSPACE
SB	pointer	Scan pointer into bottom of TOSPACE
ST	pointer	Scan pointer into top of TOSPACE
Q	integer	Scanning quota
r	integer	Number of registers
R[1:r]	array	Registers

```
boolean procedure TOSPACE(x) ::          % Returns true if x is in TOSPACE                  %


boolean procedure FROMSPACE(x) ::        % Returns true if x is in FROMSPACE                %


procedure FLIP( )                        % Relabels the two semispaces and initializes the
                                           pointers  F, T, B, SB, PB, and ST.               %


pointer procedure ALLOC(top,x,y) ::      % Allocates a new cell in either semispace.        %
  begin                                  % Allocate in either semispace                     %
    top := top-2;                        % Decrement top of free space                      %
    top[0] := x;                         % Write the cell                                   %
    top[1] := y;                         %                                                  %
    top;                                 % Return pointer to new cell                       %
  end
```

```
pointer procedure FORWARD(p) ::                 % Follows forwarding addresses if necessary and
                                                  returns pointer to data cell              %
    if fromspace(p) and                         % If p has been copied                      %
        tospace(p[0])
       then  p[0]                               % Point to copied version                   %
       else  p;                                 % Else return p                             %


pointer procedure COPY(p,d) ::                  % Copies the data cell pointed to by p at d %
  begin
    d[0] := p[0];                               % Copy CAR pointer of p at d                %
    d[1] := p[1];                               % Copy CDR pointer of p at d                %
    p[0] := d;                                  % Leave forwarding address in p             %
  end


pointer procedure CAR(x) ::                     % Redirect any CAR pointer in TOSPACE to a data cell
                                                  (qv. Condition 2).                        %
    if tospace(x)
       then  x[0] := forward(x[0])              % Rewrite CAR of if needed                  %
       else  forward(x[0])                      % Follow any forwarding address             %


procedure RPLACA(x,y)
    if fromspace(x) and                         % Check Condition 2                         %
        tospace(y)                              %    (False forward address)                %
       then begin
          copy(x,T);                            % Copy x into TOSPACE at T                  %
          T[0] := y;                            % Redirect copied cell                      %
          T := T-2;                             % Point T to top free cell                  %
        end
    else if tospace(x) and                      % Check Condition 1                         %
            (x less than SB  or                 %    (Keep TOSPACE                          %
             x greater than ST)  and            %    self referential)                      %
            fromspace(y)
       then begin
          x[0] := copy(y,T);                    % Copy pointee into TOSPACE                 %
          T := T-2;                             % Point T to top free cell                  %
        end
    else  x[0] := y;                            % Garden variety RPLACA                     %


pointer procedure CDR(x) ::                     % Redirect any CDR pointer to a data cell   %
    x[1] := forward(x[1])                       % Rewrite CDR pointer if needed             %


procedure RPLACD(x,y)
    if tospace(x)  and                          % Check Condition 1                         %
        (x less than SB  or                     %   (Keep TOSPACE                           %
         x greater than ST)  and                %   self referential)                       %
        fromspace(y)
       then begin
          x[1] := copy(y,T);                    % Copy new pointee and redirect             %
          T := T-2;                             % Point T to top free cell                  %
        end
       else  x[1] := y                          % Garden variety RPLACD                     %


pointer procedure CONS(x,y,d) ::
  begin
    collect( );
    if tospace(x) or                            % Check Condition 2                         %
       then alloc(T,x,y)                        % Allocate cell in TOSPACE                  %
       else alloc(F,x,y);                       % Else in TOSPACE                           %
  end
```

```
procedure COLLECT( )
    for i = 1 to Q do
      if not linearize(i) then              % Try to continue CDR-linearization          %
      if not bseed(i) then                  % Get linearization seed below B             %
      if not rseed(i) then                  % Get seed through a register                %
      if not tseed(i) then                  % Get seed from above T                      %
      flip()                                % Start next collection cycle                %


boolean procedure LINEARIZE(i) ::           % Extends linearization as long as possible. Returns
                                              false if a new seed is needed, true if the scan
                                              quota was met.                             %
    if  i greater than Q  then true         % Scanning quota met                         %
    else if PB = B then false               % Can't extend linearization                 %
    else if not
      fromspace(PB[1] := forward(PB[1]) )
         then begin                         % CDR of PB already in TOSPACE               %
              PB := B;                      % Stop linearization                         %
              i := i+1;                     % Increment scanning counter                 %
              false;                        % Look for new seed                          %
           end
         else begin
              PB[1] := copy(PB[1],B);       % Copy CDR of PB                             %
              PB := B;                      % Move linearization pointer                 %
              B := B+2;                     % Point to next free bottom cell             %
              i := i+1;                     % Increment scanning counter                 %
              linearize(i);                 % Try next linearization                     %
           end


boolean procedure BSEED(i) ::               % Try to get a new seed by scanning below B. Returns
                                              false if all cells have been scanned, true if
                                              a seed was found or the scan quota met.    %
    if i greater than Q then true           % Scan quota met                             %
    else if SB = B then false               % No linearization seeds below B             %
    else if
      fromspace( SB[0] := forward(SB[0]) )
         then begin                         % Found a new seed                           %
              B := copy(SB[0],B)+2;         % Copy the new seed at B                     %
              SB := SB+1;                   % Increment scan pointer                     %
              i := i+1;                     % Increment scanning counter                 %
              true;                         % Resume linearization                       %
           end
         else begin                         % Continue seed search                       %
              SB := SB+1;                   % Increment scan pointer                     %
              i := i+1;                     % Increment scanning counter                 %
              bseed(i);                     % Scan next cell                             %
           end


boolean procedure RSEED(i) ::               % Try to obtain a new seed from the lowest order
                                              register. Returns false if all registers point
                                              into TOSPACE, true if a new seed was found. %
    for j=1 to r do                         % Scan the registers                         %
      if i greater than Q then              % Scanning quota met                         %
         begin
           j := r;                          % Halt register check                        %
           true;                            % Set up return to mutator                   %
        end
      else if fromspace(R[j]) then          % Register pointing into FROMSPACE           %
         begin
           i: = i+1;                        % Increment the scanning counter             %
           R[j] := copy(R[j],B);            % Copy the new seed                          %
           j := r;                          % Halt register check                        %
           true;                            % Set up new round of linearization          %
        end
      else begin
           i := i+1;                        % Increment scanning counter                 %
           false                            % Continue search for seed                   %
        end
```

```
boolean procedure TSEED(i) ::                  % Try to get a new seed by scanning above T. Returns
                                                 false if all cells have been scanned, true if
                                                 a seed was found or the scan quota met.        %
    if i greater than Q then true              % Scanning quota met                             %
    else if ST = T then false                  % No new seeds above T                           %
    else if
      fromspace( ST[0] := forward(ST[0]) )
         then begin                            % Found a new seed                               %
              B := copy(ST[0],B)+2;            % Copy new seed at B                             %
              ST := ST-1;                      % Decrement top scan pointer                     %
              i := i+1;                        % Increment scanning counter                     %
              true;                            % Set up next round of linearization            %
           end
         else begin                            % Continue search for seed                       %
              ST := ST-1;                      % Decrement top scan pointer                     %
              i := i+1;                        % Increment scanning counter                     %
              tseed(i);                        % Scan next cell                                 %
           end
```