Garbage Collection of Linked Data Structures

JACQUES COHEN

Department of Physics, Brandeis University, Waltham, Massachusetts 02254

A concise and unified view of the numerous existing algorithms for performing garbage collection of linked data structures is presented. The emphasis is on garbage collection proper, rather than on storage allocation. First, the classical garbage collection algorithms are reviewed, and their marking and collecting phases, with and without compacting, are discussed. Algorithms describing these phases are classified according to the type of cells to be collected: those for collecting single-sized cells are simpler than those for varisized cells. Recently proposed algorithms are presented and compared with the classical ones. Special topics in garbage collection are also covered: the use of secondary and virtual storage, the use of reference counters, parallel and real-time collections, analyses of garbage collection algorithms, and language features which influence the design of collectors. The bibliography, with topical annotations, contains over 100 references.

Key Words and Phrases: garbage collection, list processing, marking, compaction, varisized cells, reference counters, secondary storage, parallel and real-time collection, analyses of algorithms, language implementation

CR Categories: 1.3, 4.10, 4.20, 4.34, 4.40

INTRODUCTION

Garbage collection—the process of reclaiming unused storage space—can be done by various algorithms. Since the late fifties and early sixties, when the first list-processing languages were implemented, many such algorithms have been proposed and studied.

Interest in garbage collection has increased considerably during the past decade with the introduction of records and pointers as data structures in new programming languages. The efficiency of programs written in these languages depends directly on the availability of fast methods for garbage collection. (Experience with large LISP programs indicates that substantial execution time—10 to 30 percent—is spent in garbage collection [STEE75, WADL76].)

Garbage collection has also become an important topic in data structures courses. Of the several books which have devoted entire sections to garbage collection [FOST68, KNUT73, BERZ75, ELSO75, HORO75, PFAL77, GOTL78, AUGE79, STAN80], Knuth's book, Section 2.3.5, is the most comprehensive. It contains detailed descriptions and analyses of some of the garbage collection algorithms that appeared prior to 1968, and, despite its age, it remains a standard reference for algorithms proposed before the seventies. Numerous papers have appeared since 1973.[1] However, no presentation has summarized and

[1] The book by Standish [STAN80], which appeared since this paper was submitted for publication, is a valuable reference on more recent work done in garbage collection.

CONTENTS

INTRODUCTION
1 COLLECTING SINGLE-SIZED CELLS
1 1 Marking
1 2 Reclaiming Marked Cells
1 3 Moving Collectors
2 COLLECTING VARISIZED CELLS
2 1 Marking
2 2 Reclaiming Marked Cells
2 3 Moving Collectors
3 COLLECTING IN VIRTUAL MEMORY
4 REFERENCE COUNTERS
5 PARALLEL AND REAL-TIME COLLECTIONS
6 ANALYSES
7 REMARKS ON LANGUAGE IMPLEMENTATION
8 FINAL REMARKS
ACKNOWLEDGMENTS
REFERENCES

classified the work done in the area. The purpose of this paper is to provide such a presentation. More specifically, the objectives of this paper are

(1) to review the classical algorithms for collecting linked data structures;
(2) to provide a unified description of recent garbage collection algorithms and to explain how they relate to the classical ones;
(3) to survey the related topics of real-time garbage collection, analyses of garbage collection algorithms, and language features which influence implementation;
(4) to present a comprehensive bibliography on the subject.

Although storage allocation and garbage collection are interrelated, the emphasis of this paper is on garbage collection proper, that is, on reclaiming storage; buddy systems [KNUT73] and related work are not covered.

It is assumed that the reader is familiar with at least one list-processing language and has some understanding of its implementations. (This level of proficiency may be acquired by studying the initial chapters of Weissman's book [WEIS67] and the interpreter described in COHE72.) This paper should be useful to readers interested in data structures and their application in compiler construction, language design, and database management.[2]

A *cell* is a number (≥ 1) of contiguous computer words which can be made available to a user. Cells are requested by the user's program from a supervisory program known as the storage allocator. Since the number of available cells is finite, a time may come when no cells remain available. When this occurs, experience indicates that some of the previously requested cells will be *unused* and can therefore be returned to the allocator. A cell becomes unused, or "garbage," when it can no longer be accessed through the pointer fields of any reachable cell. It is the garbage collector's task to reclaim this unused storage space.

Garbage collection is usually triggered automatically either when the allocator runs out of space or shortly before. Higher level languages often contain primitives for requesting groups of words from the allocator. Garbage collection may be triggered when one of these primitives is executed. For example, in LISP, the function *cons* also calls the garbage collector.

A most vexing aspect of garbage collection is that program execution comes to a halt while the collector attempts to reclaim storage space. On modern fast computers, the program interruption is noticeable even to interactive users of dedicated processors. Users of time-sharing systems may experience interruptions lasting minutes. In extreme cases, successive collections may take place with little actual program execution between them, making continued execution impractical. Because of this necessary halt, until recently, languages allowing automatic collection of linked structures could not be used to write programs with real-time constraints.

Methods for garbage collection usually comprise two separate phases:

(a) Identifying the storage space that may be reclaimed.
(b) Incorporating this reclaimable space into the memory area available to the user.

[2] Collections in very large databases or file systems are not covered in this survey

Phase (a) can be performed using one of two methods:

(a1) By keeping counters indicating the number of times cells have been referenced. Identification in this case consists of recognizing inaccessible cells (those whose reference count is zero).

(a2) By keeping a list of immediately accessible cells and following their links to trace and mark every accessible cell. This method of identification is usually called *marking*.

Phase (b) can also be subdivided into two classes:

(b1) Incorporation into a *free list* in which available cells are linked by pointers.

(b2) Compaction of all used cells in one end of the memory, the other end containing contiguous words which are made available to the allocator. There are various types of compaction, classified by the relative positions in which cells are left after compaction:

(b2.1) *Arbitrary*. Cells which originally point to one another do not necessarily occupy contiguous positions after compaction.

(b2.2) *Linearizing*. Cells which originally point to one another (usually) become adjacent after compaction.

(b2.3) *Sliding*. Cells are moved toward one end of the address space without changing their linear order.

It is also convenient to classify garbage collection according to the type of cells which are reclaimed. The early methods were applicable only to programs in which all cells were of the same size. With the introduction of records (or similar structures) into programming languages, it became important to perform garbage collection in programs involving cells of different sizes.

1. COLLECTING SINGLE-SIZED CELLS

1.1 Marking

LISP cells illustrate the problems involved in marking single-size cells. Each LISP cell has two fields: *left* (or *car*) and *right* (or *cdr*). These fields contain pointers either to other cells or to atoms, special kinds of cells containing no pointers. Each cell also contains two Boolean fields (bits): one to help differentiate between atomic and nonatomic cells,[3] and the other to be used in marking.

```
procedure mark(p); {p is a pointer that is called by value}
  begin
    if unmarked(p) then
      begin
        marknode(p);
        if nonatomic(p) then
          begin
            mark(left(p)),
            mark(right(p))
          end
      end
  end mark;
```

Figure 1

The algorithm shown in Figure 1 is a recursive procedure for marking LISP lists (including atoms). It utilizes three auxiliary procedures:

nonatomic(p): Boolean function which tests whether the cell pointed to by p is nonatomic;

unmarked(p): Boolean function which tests whether the cell pointed to by p is unmarked;

marknode(p): Procedure which marks the cell pointed to by p by turning on its marking bit (marking bits are initially turned off).

Note the similarity of the marking algorithm in Figure 1 with the classical preorder tree-traversal algorithm [KNUT73]. The one in Figure 1, however, can handle general lists, including circular ones.

An efficient nonrecursive version of this algorithm uses an explicit stack which only stores pointers to the cells being marked. No return addresses need to be stacked. A pointer is pushed onto the stack just before marking the cell's right branch. The algorithm terminates when the stack is empty. Consequently, each node of the list is visited twice: once before marking the left field and once before marking the right field.

[3] Some LISP systems carry this information in the pointers to the cells

The following predicament results from using the described algorithm in a collector operating exclusively in main memory: garbage collection is needed because of the lack of memory space; however, additional space is required by the stack of the marking algorithm. If the storage area consists of n LISP cells, the maximum depth required for the stack is then n. To reserve this much additional storage initially is uneconomical. Several algorithms have been proposed to circumvent this difficulty; all of them involve reducing the required storage by trading it for longer time needed in which to perform the marking.

The first of these algorithms (similar to Algorithm C, in KNUT73, p. 415) uses a stack of fixed length h, where h is substantially smaller than n. However, the pointers are stacked using mod h as the stack index. In other words, the stack can be thought of as being "circular," and when its index exceeds h, the additional information is written over previously stored information. The stack therefore only "recalls" the most recently stored h items and "forgets" the other ones.

First, the immediately accessible cells are marked. Marking then proceeds as in the algorithm in Figure 1. However, since some cells which should have been "remembered" have been "forgotten," the stack will become empty before the task is complete. When this happens, the memory is scanned from the lowest address, looking for any marked cell whose contents point to unmarked cells. If such a cell is found, marking resumes as before and continues until the stack becomes empty again. Eventually, a scan will find no marked cells referring to unmarked cells, and marking is complete.

Actually, the scanning need not start from the beginning of the memory each time. During marking, the algorithm can record the minimum address f of the forgotten nodes. The next scan will begin either just after the last address of the previous scan, or from f, whichever is smaller.

An elegant algorithm which dispenses with the use of a stack but which may require one additional bit per cell was developed independently by Deutsch and by Schorr and Waite (see SCHO67 and KNUT73). The main idea of this algorithm is that the nodes of a tree or of a directed graph can be inspected without using a stack by reversing successive links until leaves (i.e., atoms) or already visited nodes are found. The link reversal can then be undone to restore the original structure of the tree or graph. (One can view the stack of the classical marking algorithm as "moved" into the cells by the link-reversal technique.) The additional bit per cell (called a *tag* bit) indicates the direction in which the restoration of reversed links should proceed (i.e., whether to follow the left or the right pointer). Knuth [KNUT73] suggests a method for avoiding using a tag bit by instead using the bit already necessary for testing whether a cell is atomic.

Veillon [VEIL76] has shown that it is possible to transform the classical recursive algorithm in Figure 1 into the Deutsch-Schorr-Waite algorithm. First, the parameter of the recursive procedure is eliminated by introducing the link reversal feature. The two recursive calls of the resulting parameterless procedure, needed to mark the left and right fields, are eliminated by introducing the tag bits to differentiate between the returns from the two calls.

Knuth (KNUT73, p. 591) proves by induction the correctness of the link-reversal marking algorithm of Deutsch-Schorr-Waite. An alternate proof may be obtained by noting that the transformations suggested by Veillon preserve correctness. Other proofs have recently appeared in GERH79, GRIE79, LEE79, KOWA79, and TOPO79. Yelowitz and Duncan [YELO77] present proofs of correctness of several marking algorithms. Their approach consists of first proving the correctness of a general abstract marking algorithm and then extending that proof to cover specific concrete algorithms derived from the abstract one.

Wegbreit [WEGB72b] proposes a modification of the Deutsch-Schorr-Waite algorithm which uses a bit stack instead of a tag bit per cell. In the light of Veillon's program transformation, one sees that Wegbreit's stack simply implements the returns from the parameterless recursive procedure derived from Figure 1.

In the algorithm of Figure 1, each cell is

visited twice. In the Deutsch–Schorr–Waite algorithm, the cells are visited three times. This additional visit and the overhead for restoring pointers and for checking and setting bits render this algorithm less efficient than the classical algorithm. (Benchmarks taken by Schorr and Waite showed that this is indeed true.) Schorr and Waite then proposed using a hybrid algorithm which combines a fixed-size stack with their link-reversal technique. It consists of using the stack algorithm whenever possible. If stack overflow occurs, the tracing and marking proceed by the method of link reversal (see KNUT73, p. 592).

Other marking algorithms which use a fixed-length stack have been proposed. The one by Kurokawa [KURO75] also uses a tag bit, but differently. When the fixed-length stack overflows, it is possible to remove some of the pointers from the stack and preserve the information by turning on the tag bit of the unstacked cells. These cells form a chain, the pointer to which is left on the stack. The removal of stack elements makes more space available for resuming the marking scheme. Later, when a pointer is unstacked, it is examined to determine whether the cell it points to is tagged. If so, the linked tagged cells are retraced. Kurokawa also proposes a variant of the algorithm which dispenses with the tag bits, using the mark bits instead.[4]

Peter Bishop has proposed a variant of Kurokawa's algorithm which deserves further investigation: When the stack overflows, its contents are "moved" into the cells according to the link-reversal technique. Marking then proceeds using the now-free area of the stack. If stack underflow occurs, an element can be popped (in the manner of the Deutsch–Schorr–Waite algorithm) from the portion of the stack stored in the cells. A careful comparison of Kurokawa's and Bishop's algorithms has not yet been done, nor has either yet been proved correct.

We have seen that, at most, three bits per cell are necessary to perform LISP's garbage collection. The first two are used in recognizing atoms and in marking; the third one is used as a tag bit, if needed by the algorithm. It should be pointed out that these three bits need not be located within or near their corresponding cells. Special areas of the memory (bit maps or tables) may be allocated for this purpose. Whether or not this should be done is, of course, machine dependent. Some LISP processors, for example, avoid the need for an (explicit) atom bit by placing atomic cells in a special region of the memory.

A convenient manner of implementing the tag bit in certain machines is described in COHE72. It takes advantage of the fact that all pointers refer to only odd (or only even) addresses, since two words are always used to implement a LISP cell. Turning on a tag bit can thus be accomplished simply by adding one to the address contained in the *right* part of the cell.

The algorithms described in this section can be generalized to cover cells of a single-size m, with $m > 2$. The generalized version of the algorithm in Figure 1 would involve recursively marking each of the m fields of the cell. A generalized variant of the Deutsch–Schorr–Waite algorithm would require an additional $\log_2 m$ tag bits per cell, the number necessary to represent m.

1.2 Reclaiming Marked Cells

The simplest method for reclaiming the marked cells (see phase (b1) of the Introduction) consists of linearly sweeping the entire memory. After turning off their mark bit, unmarked cells are incorporated into the free list administered by the storage allocator.

If compacting is preferred (phase (b2)), it can be performed by scanning the memory twice. In the first scan, two pointers are used, one starting at the bottom of the memory (higher address), the other at the top. The top one is incremented until it points to an unmarked cell; the bottom pointer is then decremented until it points to a marked cell. The contents of the marked cell are thereupon moved to the unmarked cell, a pointer to the new cell is placed in the old, and the mark bits are turned off. By the time the two pointers

[4] LIND74 had shown how marking can be done without tag bits or a stack, at the expense of additional processing time.

meet, all marked cells have been compacted in the upper part of the memory.[5]

The second scan is needed for readjusting the pointers: since some cells have been moved, it is essential to update any pointers to obsolete cell locations. This scan sweeps only the compacted area. Pointers are readjusted whenever they point to cells whose contents have been moved from the liberated area to the compacted area of the memory. Each of these pointers is replaced by the contents of the cell to which it was pointing. According to Knuth [KNUT73, p. 421], this method was first proposed by D. Edwards. LISP and ALGOL 60 programs describing in detail this method of compacting have appeared in HART64 and COHE67b. Note that the two-pointer compactor is of the *arbitrary* type; after compaction, cells which originally point to one another do not necessarily occupy contiguous positions of the memory.

1.3 Moving Collectors

An obvious algorithm for garbage collection would be to output all useful (i.e., reachable) data to the secondary storage area and then to read them back to the main memory. This, however, has several drawbacks:

(1) It may require additional storage equally as large as the main memory.
(2) The time overhead for transferring between memories is (usually) considerable.
(3) Unless special precautions are taken, shared cells would be output more than once, in which case the main memory may not be sufficiently large for reading back the information. (This situation becomes critical when the main memory contains loops of pointers.)

Minsky [MINS63] proposes an algorithm which eliminates the difficulties described in (3). His algorithm does not use a stack, but requires one marking bit per LISP cell. Each cell is traced and marked if unmarked. Triplets (the new address of a cell and the contents of its left and right fields) are computed and output to the secondary storage. The new address is also placed in the marked cell, and whenever a pointer to that cell is encountered, the pointer is adjusted to reflect the move. When the triplets are subsequently read back into the main memory, the contents of the fields are stored in the specified new address. Minsky's algorithm has the advantage of compacting the useful information into one area of the main memory. After compaction, list elements which are linked are positioned next to each other, making Minsky's algorithm a *linearizing* compactor. These two properties are very important when virtual memory is used, as will be discussed in Section 3.

In Minsky's algorithm, fields of the original list are used to store information about the output list; consequently, the original list is destroyed. In this respect, it is convenient to distinguish between the terms *moving* and *copying*. The former implies a possible destruction of the original structure, whereas the latter does not. Minsky's algorithm can be used to move lists in contexts other than garbage collection. Since its appearance, several other algorithms have been proposed to perform moving or copying. They can be used for garbage collection purposes as well. Most are designed to move or to copy lists without resorting to mark bits or to a stack.[6] As in Minsky's algorithm, (1) a forwarding address is usually left in the old cell, and pointers referring to that cell are readjusted accordingly, and (2) the moved lists are compacted in contiguous positions of the memory.

A few algorithms have been proposed for *copying* lists without using a stack or mark bits. They differ from the moving algorithms in that the altered contents of old lists are later restored to their original values. Lindstrom [LIND74], Robson [ROBS77], Clark [CLAR75, CLAR78a], and Fisher [FISH75] discuss the copying of trees and general lists.

Fenichel and Yochelsen [FENI69] suggest a variant of Minsky's collector which uses

[5] This type of compaction is similar to that performed in solving a problem proposed by Dijkstra (see the Dutch flag problem in DIJK76a)

[6] The similar but simpler problem of traversing trees without a stack or mark bits has been considered in SIKL72, DWYE73, LIND73, ROBS73, and LEE80. A recent book by Standish [STAN80] contains detailed descriptions of some of these algorithms

an implicit stack but does not require mark bits. They divide the available memory into two areas called semispaces. At a given time, only one area is used by the allocator. When its space is exhausted, the reachable lists are moved to the other space in a linearized compacted form. The algorithm is intended for use in a paging environment.

Cheney's algorithm [CHEN70, WALD72], Reingold's algorithm [REIN73], and Clark's algorithm [CLAR76, GOTL78] all represent improvements over the previous algorithm: they require neither a stack nor mark bits. Cheney's algorithm is done by moving the list to a contiguous area; a simple test can establish whether a pointer refers to the old or the new region of the memory. Reingold's algorithm is achieved by using the Deutsch-Schorr-Waite link-reversal technique mentioned in Section 1.1. And Clark's algorithm moves a list into a contiguous area of the memory with the stack implicit in the list being moved. Clark shows that his algorithm is in most cases more efficient than both Cheney's and Reingold's.

Moving (or copying) algorithms may be classified according to the type of traversal used when inspecting the list being moved. Let us assume that most of the cells in a list are linked by their *right* fields, as is typical of LISP programs. A nonrecursive version of the marking algorithm of Figure 1 uses a list (stack) containing the addresses of cells whose *left* field has not yet been processed.[7] This list may be administered either as a true stack (on a "last-in, first-out," LIFO order) or as a queue (on a "first-in, first-out," FIFO order). According to this classification, the algorithms by Minsky, Fenichel-Yochelson, Reingold, and Clark use a LIFO order, whereas the one by Cheney uses a FIFO order. All of these algorithms move into adjacent locations the cells which originally were linked by the *right* field. These algorithms may therefore be classified as performing a linearizing type of compaction. Note that the algorithms which use a LIFO order will move closer together the cells corresponding to the sublists which terminate a list.

[7] This corresponds to calling *mark*(*right* (*p*)) before *mark*(*left* (*p*))

```
procedure mark(p); {p is a pointer that is called by
                    value}
  begin integer i;
    if unmarked(p) then
      begin
        marknode(p),
        if nonatomic(p) then
          begin
            for i ← 1 until number(p) do
                mark(field (p, i))
          end
      end
  end mark,
```

Figure 2

2. COLLECTING VARISIZED CELLS

2.1 Marking

Figure 2 shows a marking algorithm similar to that of Figure 1, but applicable to varisized cells. Two additional auxiliary procedures are used:

number (*p*): an integer function yielding the number of contiguous words (items) in the cell to which *p* points (this information may be stored in the cell itself); and

field (*p, i*): a function yielding the *i*th item of the cell pointed to by *p*.

It is assumed that *p* always points to the first item of the cell. The algorithm can be modified to handle pointers to cell parts. If so, care should be taken to avoid collecting chunks of cells. Under the modified algorithm each item of the cell needs to be marked; thus bit tables are economical. (Note that bit tables would be less useful in conjunction with the algorithm of Figure 2.)

The algorithm in Figure 2, like that in Figure 1, requires stack storage space when none may be available. If the memory contains *n* cells of various sizes, the maximum depth required for the stack is *n*. When most of the cells contain several items, it might be worthwhile to reserve two additional fields per cell for distributing stack storage among the cells. Essentially, these fields contain the quantities *p* and *i* needed to implement the recursive calls of the procedure in Figure 2. A description of an algorithm of this kind appears in THOR72.

The marking algorithms of Section 1 which use a fixed-length stack can also be

adapted to process varisized cells. They may then use a fixed-length stack of height h with stack index = mod h as before, but each stack position will contain information corresponding to p and i in the algorithm of Figure 2.

Variants of the Deutsch-Schorr-Waite link-reversal algorithm applicable to varisized cells are described in THOR72 and THOR76. Instead of using one tag bit, some of these algorithms use $\log_2 maxm$ bits per cell, where $maxm$ is the size of the largest cell. Other variants of the Deutsch-Schorr-Waite algorithm applicable when marking varisized cells have appeared in HANS77, MARS71, and WODO71.

2.2 Reclaiming Marked Cells

In opening, it should be mentioned that the method of compacting described in Section 1.2 is not applicable to varisized cells, since marked and unmarked cells cannot be swapped if they are of different sizes.

Several algorithms have been proposed for compacting varisized cells. One of the earliest is that of Haddon and Waite [HADD67, WAIT73]. This compactor is of the *sliding* type (see Section 1) and performs two scans of the entire memory. The objective of the first scan is to perform the compaction and to build a "break table," which the second scan uses to readjust the pointers.

The break table contains the initial address of each "hole" (sequence of unmarked cells) and the hole's size. An interesting feature of Haddon and Waite's algorithm is that no additional storage is needed to construct the break table since it can be proved that the space available in the holes suffices to store the table. However, from time to time the break table must be "rolled," that is, moved from one hole to a bigger one created through compaction. At the end of the first scan the break table occupies the liberated part of the memory. It is then sorted to speed up the pointer readjustment done by the second scan. Readjustment consists of examining each pointer, consulting the table (using a binary search) to determine the new position of the cell it used to point to, and changing the pointer accordingly.

The most unfavorable condition for Haddon and Waite's algorithm is when unit-size active cells alternate with unit-size inactive cells. It can be shown that the algorithm would take $O(n \log n)$ time, where n is the size of the storage (see FITC78).

Other compacting algorithms for varisized cells have been proposed. The LISP 2 garbage collection algorithm described in KNUT73, pp. 602-603, and those presented in WEGB72a and THOR76 have the following features in common.

Three (or more) linear scans are used. In the first scan the holes (inaccessible cells) are linked to form a free list. Two fields are reserved in each hole to store its size and a pointer to the next hole. A subsequent scan may combine adjacent holes into a single larger hole. The second scan consists of recognizing pointers and using the information contained in the free list to adjust them. This involves finding the ith hole whose address a_i is such that $a_{i-1} < p < a_i$, where p is the pointer being readjusted. The new value of the pointer can be computed by subtracting from p the sum of the sizes of the 1st, 2nd, ..., $(i-1)$th holes.[8] Once the pointers have been readjusted, a third scan takes care of moving the accessible cells to the compacted area. This compactor is therefore of the *sliding* type.

The second scan, which interpretively readjusts pointers, is the most time consuming of the three scans. Wegbreit [WEGB72a] proposes variants of this algorithm which make this scan more efficient. One variant consists of constructing a break table (called directory) which summarizes the information contained in the free list of holes. However, storage for the directory may be unavailable. Wegbreit suggests trying to use the largest hole for this purpose. When this is possible, binary search can speed up pointer readjustment.

Lang and Wegbreit [LANG72] suggest another variant of the algorithm, which subdivides the memory into a fixed number of equal segments. This variant requires a small additional area of memory to store the reduced break table, its initial address,

[8] It is therefore convenient to store these cumulative sums instead of recomputing them every time they are needed

and its size for each segment. A first scan compacts each segment toward its lower address and constructs its break table. Whenever possible, that break table is copied into the liberated area of the segment; otherwise, marks are set to indicate that the reduced break table is stored in the liberated area of another segment. The second scan performs pointer readjustments using the information in the individual break tables. A third and final scan compacts the segments.

Another variant for collecting varisized cells was proposed in ZAVE75. It requires that each cell have an additional field. In the marking phase, all active cells are strung together using the additional field. This list of active cells is sorted by increasing addresses.[9] Pointers can then be readjusted by consulting the addresses in the list. The final scan compacts the active cells.

Terashima and Goto [TERA78] propose two algorithms for the compacting phase of the collection of varisized cells. In the first, pointers are readjusted by recomputing, for each pointer, the needed part of the break table. This computation is sped up by organizing the holes in a balanced binary tree, with the necessary pointers stored within the holes themselves. The balanced tree form minimizes computation of the readjustments. An intermediate scan is needed to construct the balanced tree from the linear list of holes obtained just after the marking phase.

The second compacting collector proposed by Terashima and Goto assumes that all elements of a cell are marked, and a separate bit table is used for marking. The memory is subdivided into a number of equal segments, each as long as the number of bits in a word of the bit table. Thus the size of the free space within a segment can be efficiently computed by counting the number of inactive bits in a word of the bit table. Pointer readjustment is based on these bit counts. This method is suitable for hardware implementation.

An interesting algorithm for readjusting pointers and compacting varisized cells has recently been proposed by Morris [MORR78, MORR79]. It performs the compacting in linear time and it requires only one additional bit per pointer. No break tables are used. The algorithm is based on the following property: Assume that the contents of locations $a_1, \ldots, a_n$ point to location z. No information is lost if this tree structure with root z is transformed into a linear list by stringing together locations z, a_1, a_2, ... and placing the contents of z in a_n. Once the new position of z, say, z', is known, it is simple to reconstruct the original tree by making the a_i's point to z'. The extra bit is used to process the tree structures.

Morris' algorithm is of the *sliding* type and requires two scans. The first only readjusts forward-pointing references. The second updates references pointing backward and performs the compaction. Although Morris proves the correctness of the algorithm, no data are available comparing its efficiency to that of other compacting algorithms. An algorithm similar to Morris' but requiring only forward scans and no additional bits has been proposed by Jonkers [JONK79].

Marking, pointer readjustment, and compacting can be made simpler if the list processing "preserves address ordering." This means that nodes are allocated sequentially, from low to high address: when a cell is created, its descendants have addresses which are always smaller than its own, and circular lists are therefore excluded. Under these conditions, marking can be performed in a single scan through the entire memory without using a stack. This scan also finds the number of active cells, which the second scan then uses for readjusting the pointers. The third and final scan performs the compaction which is of the sliding type. Details are given in FISH74.

Proposals have been made to try to postpone, as much as possible, the compaction of varisized cells [KNUT73, Section 2.5; PARE68]. This may be accomplished by keeping several free lists, one for each cell size commonly used in a program. These are called homogeneous free lists, or simply H-lists. In addition, another free list, the M-list, contains cells of miscellaneous sizes.

[9] This sorting may be expensive if the memory is fragmented.

The cells in the M-list are linked according to increasing addresses; the ordering in the H-lists is irrelevant. An unused cell is returned to one of the H-lists if possible. Otherwise, the cell is returned to its appropriate position in the M-list.

Requests for new cells are handled according to their size. If there is a nonempty H-list of the desired size, the new cell is taken from that list. If not, the cell is taken from the first M-list cell as large or larger than the desired size. If the M-list cell is larger than needed, it is split into two cells, with the first used to satisfy the request, and the second returned to one of the free lists.

If a cell of the requested size cannot be found in the M-list, a semicompaction is attempted. It consists of returning all elements of the H-lists to their appropriate positions in the M-list, and whenever two or more cells in the M-list are adjacent, combining them into a single larger cell. The test for adjacency is simple since the M-list is ordered by address. It is of course possible that even after semicompaction, a cell with the requested size remains unavailable. Standard (full) compaction may then be the only way to avoid program termination.

2.3 Moving Collectors

Some of the moving algorithms mentioned in Section 1.3 may be adapted to handle varisized cells. A representative of this class of algorithms [FENI69, CHEN70, BAKE78b] is described in the next section since it is particularly suitable for operation in virtual memory.

An algorithm for copying varisized cells is described in STAN80. It requires cells to have an additional field large enough to store an address. A first pass consists of linking all used cells via the additional field (see THOR72). A second pass copies each cell c_i in the linkage and inserts the copy, c_i', as the successor of c_i. The successor of c_i' becomes the cell c_{i+1}. After this copying, the odd-numbered elements of the new linkage contain the original cells and the even-numbered ones contain the copies. Finally, a third pass is used to readjust the pointers in each copied element and to separate the copy from the original. The degree of linearization achieved by this algorithm depends on the manner by which the cells are linked during the first pass. (Fisher [FISH75] and Robson [ROBS77] also describe algorithms for copying LISP cells which can be generalized for copying varisized cells.)

3. COLLECTING IN VIRTUAL MEMORY

The ratio of the size of secondary memory to the size of main memory is an important factor in designing collectors which operate in virtual memory. When this ratio is small, some of the algorithms described in the previous sections may be used. The methods described in this section, though, are suitable when the ratio is large.

The use of secondary storage through paging [COHE67a] changes the design considerations for implementing garbage collection algorithms in important ways. First, it is no longer necessary to try to avoid using additional storage for a stack, since the size of the available virtual memory in current systems is considerable.[10] Avoiding page faults and thrashing (caused by having structures whose cells are scattered in many pages), on the other hand, becomes a critical factor in improving the efficiency of garbage collection. Compaction is for this latter reason important when collecting in this environment. Cohen and Trilling [COHE67b] show that garbage collection with compaction brings about significant time gains in performance of LISP programs. They also found that a direct transcription of the classical garbage collection algorithms to a virtual memory environment can lead to unbearably slow collection times. CLAR79 contains additional useful information about the performance of compacting collectors operating in virtual memory.

Compaction of cells in virtual memory should not only eliminate unused holes but should also construct the compacted area so that pointers refer, if possible, to neighboring cells. As mentioned in Section 1.3,

[10] It is therefore doubtful that the link-reversal technique of Deutsch–Schorr–Waite should be used for marking

Minsky's algorithm [MINS63] satisfies this requirement.

Measurements in actual LISP programs show that about 97 percent of list cells have just one reference to them [CLAR77, CLAR78b]. This property is important when designing garbage collection algorithms which operate in virtual memory.

Bobrow and Murphy [BOBR67] show that the use of a selective *cons* (the LISP function which requests a cell from the allocator) can improve the efficiency of subsequent processing and garbage collection. Basically, they advocate keeping one free-list per page. A new cell requested by a call of *cons*[*x*, *y*] is taken from the free area of a page according to the following strategy.

(1) First, if possible, take from the page containing the cell pointed to by *y*; otherwise,
(2) take from the page containing the cell pointed to by *x*; otherwise,
(3) take from the page containing the most recently created cell; otherwise,
(4) take from any page containing a fair number (say, 16) of free cells.

The purpose is to minimize page faults in manipulating linked lists. Additional information on garbage collection using virtual memory can be found in BOBR67, BOBR68a, BOBR68b, ROCH71, and BAEC72.

An important design consideration for implementing garbage collection algorithms in a paging environment is deciding when collection should be invoked. Since very large memories are currently available, it seems reasonable to collect whenever page faults render the program processing unbearably slow.

A class of algorithms suitable for use in virtual memory is the one described by Baker [BAKE78b]. It is based on the copying collector proposed by Fenichel-Yochelson [FENI69] and by Cheney [CHEN70] which was briefly described in Section 1.3. What follows is a more detailed presentation of this type of algorithm. Although it is applicable in collecting varisized cells, this presentation applies only to LISP cells.

The available memory is divided into two areas called semispaces. At a given time, only one is used by the allocator. During garbage collection, the reachable lists are moved to the other space in a compacted form. The heart of the algorithm is the procedure *move* presented in Figure 3. The following description is based on Baker's paper [BAKE78b].

```
pointer procedure move(p),
  begin
    if newspace(p)
      then return p
      else
        begin
          if old space(left[p])
            then left[p] ← copy(p);
          return left[p]
        end
  end move,
pointer procedure copy(p),
  begin pointer q;
    {The following statement assigns to q the address
     of a new cell taken from a contiguous area in the
     new space; as explained in the text this action
     implies incrementing the pointer B}
    q ← new,
    left[q] ← left[p];
    right[q] ← right[p],
    return q
  end copy,
```

Figure 3

The procedure *move* moves a cell from the old semispace to the new one. The Boolean functions *oldspace*(*p*) and *newspace*(*p*) are used to test whether the cell pointed to by *p* is in the corresponding semispace. The auxiliary function *copy*(*p*) copies the cell whose address is *p* into the new semispace. After the copying, the procedure *move* stores the address of the new cell into the *left* field of the old cell.

The collector calls the procedure *move*(*p*) for all accessible cells *p* in the old semispace. This task is similar to that of marking, but in this case the cells are moved instead of marked. A stack is avoided by using two pointers, *B* and *S*, both of which initially point to the bottom of the new semispace. *B* points to the next free cell in the new semispace and is thus incremented by *copy*. First the immediately accessible cells are moved to the new semispace. The area between *S* and *B* now contains cells which have been moved into the new space but whose contents have not. This area is scanned (by incrementing *S*),

and the contents of the area's cells are updated by calls to the procedure *move*. This in turn may result in incrementing *B*. Collection ends when *S* meets *B*.

In his dissertation, Bishop [BISH77] proposed an approach for designing collectors which operate in a very large virtual memory (of the order of 10^{12} bits). Even using the real-time approaches discussed in Section 5, it would be impractical to garbage collect the entire memory at one stretch. Since large portions of memory may remain unchanged during program execution, Bishop suggests collecting only in parts of the address space rather than in the entire space. (A similar approach is used in Ross' AED system [ROSS67].) The memory is divided into areas which can be collected independently, and a variant of the Fenichel and Yochelson collector is used. This collector increases the locality of reference, an important factor in a paging environment.

Tracing and copying are performed only within a given area. The system keeps lists of all interarea references, both incoming and outgoing. Incoming references are modified to point to the area's new copy; they define the immediately accessible cells from which collection starts. Before discarding the old copy of an area, its useless outgoing references are removed from the corresponding lists of incoming references.

Bishop developed a method for maintaining the lists of interarea references and indicated that this can be done automatically without incurring substantial run-time overhead. He advocates altering the virtual memory mechanism to cause traps when interarea references are stored into cells, and shows how virtual memory hardware can be constructed to perform this extra service efficiently.

4. REFERENCE COUNTERS

The use of reference counters (advocated by COLL60 and WEIZ63) has recently attracted renewed interest. An extra field, called *refcount*, is required for each cell to indicate the number of times the cell is referenced. This field has to be updated each time a pointer to the cell is created or destroyed. When *refcount* becomes equal to zero, the cell is inactive and can be collected. At least theoretically, *refcount* must be large enough to hold the number of cells in the memory and therefore must be as large as a pointer. The disadvantages of this approach are (1) the extra space needed for the counters, (2) the overhead required to update the counters, and (3) the inability to reclaim general cyclic structures.[11]

However, reference counters can conveniently be used to distribute garbage collection time as an overhead to processing. Every time a cell becomes inactive, it is pushed into a stack. When the cell is needed, it is popped from the stack and *then* the *refcounts* of its descendants are decremented. An advantage of this arrangement is that no new space is needed for the stack, since it can be simulated by stringing together the freed cells using the *refcount* fields. (Recall that these fields have to be big enough to hold a pointer.)

Deutsch [DEUT76], Knuth [KNUT73], and Weizenbaum [WEIZ69] suggest combining the reference counter technique with classical garbage collection. The former would be utilized during most of the processing time; the latter, being more expensive, would be performed as a last resort. This allows the use of small *refcounts* (thus reducing the storage requirements) because counters which reach their maximum value remain unmodified. Classical garbage collection, called when the free list is exhausted, starts by resetting all counters to zero. The counters of the accessible cells are restored during the marking phase of the collection by incrementing a cell's counter every time the cell is visited. The collection reclaims inactive circular list structures and cells with maximum *refcount* which have become unreachable. A recent paper [WISE79] shows that this restoration can be done efficiently when using Morris's compaction algorithm (see Section 2.2 and MORR78).

The hybrid approach suggested by Deutsch and Bobrow [DEUT76] is particu-

[11] BOBR80 and FRIE79 describe how reference counting can be used to manage certain classes of cyclic structures.

larly applicable to LISP. It is based on statistical evidence [CLAR77, CLAR78b] that in most LISP programs, most reference counts (about 97 percent) are one. The authors propose three hash tables (see BOBR75):

(1) The multiple reference table (MRT). Its key is a cell address and the associated value is the cell's reference count. Only cells whose reference counts are two or greater are listed in the MRT.
(2) The zero count table (ZCT) containing the addresses of cells whose *refcount* is zero. These cells may be of two types: those which are referred to only by the variables of a program (still active), and those which are truly unreferenced and can be reclaimed. It follows from (1) and (2) that if a cell's address is not in the MRT or the ZCT, its reference count is one.
(3) The variable reference table (VRT) contains the addresses of cells referred to by program variables (including the temporary variables in the recursion stack).

Deutsch and Bobrow note that there are three types of operations, called transactions, which may affect the accessibility of data. These are (1) allocation of a new cell, (2) creation of a pointer, and (3) destruction of a pointer.

Instead of updating the hash tables as the transactions occur, Deutsch and Bobrow propose storing them in a sequential file. The transactions are examined at suitable time intervals and then the tables are updated. This scheme has the advantage of minimizing paging overhead.

When a new cell is allocated, its address should be placed in the ZCT. Since this is usually followed by the creation of a pointer to the newly allocated cell (which implies removal from the ZCT), the pair of transactions can be ignored.

When a pointer is created, it is examined prior to its insertion into a cell or pointer variable. Three cases are possible:

(a) The pointer refers to a cell in the MRT. The corresponding *refcount* value is then increased by one if it has not already reached its maximum; otherwise, it is left unchanged.
(b) The pointer refers to a cell in the ZCT. The cell is then removed from that table, since its count becomes one.
(c) If tests (a) and (b) fail, the pointer refers to a cell having a *refcount* of one. It must then be placed in the MRT with a *refcount* of two.

When a pointer is destroyed (removed from a cell), two cases are possible:

(a) The pointer refers to a cell in the MRT. The cell's *refcount* value is decreased by one, except when it has reached its maximum, in which case it is left unchanged. If the new value of *refcount* is one, the cell is removed from the MRT.
(b) The pointer does not refer to a cell in the MRT. Its count is one by default, and should be reduced to zero. The cell is therefore entered in the ZCT.

The VRT is used when incorporating new cells into the free list. Since the stack is constantly being updated, the VRT is only computed periodically. A cell is reclaimed when its address is listed in the ZCT but *not* in the VRT. The ZCT is updated by eliminating the entries of reclaimed cells which are not pointed to by program variables.

Deutsch and Bobrow [DEUT76] designed their hybrid collector for operating in a paging environment, so that space availability is not at stake. For the classical collection they advocate using a variant of the two-semispace collector of Fenichel and Yochelson [FENI69]. The authors also point out that an auxiliary processor could speed up the collection. Its task would be to scan the ZCT and VRT tables to determine which cells could be incorporated into the free list.

Wise and Friedman [WISE77] propose a variant of the hybrid algorithm of Deutsch and Bobrow which is useful when only fast memory is available. Only one bit is assigned to the field *refcount*, and when that bit is one, the cell is referenced more than once. This is analogous to storing the cell in the MRT of the Deutsch–Bobrow algorithm.

Nodes whose reference counts are greater than two can be reclaimed only by a classical collection with a marking phase. An interesting feature of the one-bit *refcount* is that this bit can be re-used as a tag bit when using the link-reversal marking technique of Deutsch, Schorr, and Waite (see Section 1).

In order to delay the classical collection as much as possible, Wise and Friedman propose using tables to temporarily list cells whose *refcounts* are still one but are likely to be changed to two or zero. This situation occurs when performing assignments of the kind $r \leftarrow f(r)$, where r is a pointer. Assignments of this type are quite common in LISP, for example, $r \leftarrow cons(a, r)$ and $r \leftarrow right(r)$. The first often increases to two the *refcount* of the cell originally referred to by r. The second often reduces the count to zero. Unfortunately, no experimental data are available on the efficiency of the hybrid techniques described in this section.

Barth [BART77] considers reference counters in relation to shifting garbage collection overhead to compile time. He shows that savings in collection time are sometimes possible by carefully studying, at compile time, the program's assignments. For example, in the case of $r \leftarrow right(r)$, the cell originally pointed to by r may be incorporated into the free list if it is known that it will not be referenced by other pointers.

5. PARALLEL AND REAL-TIME COLLECTIONS

Two proposals have been made to circumvent the onerous garbage collection interruptions. The first is to allow garbage collection to proceed simultaneously with program execution by using two parallel processors: one is responsible for collection, the other for program execution. When collection actually takes place, it is bound by a known, tolerable, maximum time.

Minsky is credited by Knuth with initiating the development of algorithms for time-sharing garbage collection and list-processing tasks (see KNUT73, pp. 422, 594). If two processors are available, these tasks can be performed in parallel, with one of these processors, the collector, responsible for actual garbage collection, and the other performing the list processing and providing the storage requested by a user's program. Dijkstra [DIJK76b] calls this latter processor the *mutator*. The collector performs the basic tasks of marking and incorporating unmarked cells to a free list (see Section 1), during which time the mutator is active. The mutator may not, therefore, request cells until the collector makes them available.

The marking phase of Dijkstra's algorithm is more complex than the classical serial marking explained in Section 1. Two mark bits are required (instead of one) because a cell may be in one of three states. These states are represented by colors: *white* (unmarked), *black* (marked), and *gray* (indicating that the cell has been requested and used by a program). Intuitively, gray nodes are good candidates for becoming black. The mutator helps the marking phase of the collector by turning a white cell gray when the cell is requested and used by a program. The mutator is also responsible for triggering an interruption whenever the free list contains only one cell. Mutator processing resumes when the collector returns at least one more cell to the free list.

One of the collector's tasks is to mark the used cells, the cells in the free list, and any gray cells. This is done by initially graying the first used cell and the first free-list cell. Tracing proceeds by graying any cells linked to a gray cell c, and then blackening c. When the tracing ends, the white cells are incorporated to the free list and the black cells are whitened. As a result, inactive gray cells are first blackened by the collector and then whitened. During the *next* cycle of the collector these cells are incorporated into the free list.

France [FRAN78], Gries [GRIE77], and Muller [MULL76] provide detailed descriptions of Dijkstra's algorithm, but their main concern is to prove correctness. An extension of Dijkstra's algorithm with multiple mutators is considered in LAMP76.

Steele [STEE75] has independently developed a method for parallel garbage collection based on the Minsky–Knuth suggestion. He was one of the first to propose

actual algorithms for collecting in parallel. Steel's collector makes exclusive use of semaphores and requires two bits per cell, which are used not only for marking but also for compacting and for readjusting pointers. Compaction is done using the two-pointer technique described in Section 1.2.

Comparing Dijkstra's to Steele's algorithm is difficult because these authors had different objectives. The former wanted to assure the correctness of his algorithm (regardless of its efficiency), whereas the latter had in mind an implementation using special hardware, possibly microcoded.

In a recent paper, Kung and Song [KUNG77] propose a variant of Dijkstra's method which uses four colors for marking and which does not need to trace the free list. The authors prove the correctness of the algorithm and show that it is more efficient than Dijkstra's. To this author's knowledge, none of the parallel garbage collection algorithms has been implemented, nor are any detailed results from simulation yet available.[12]

An alternative to using two processors is to have *one* processor time-share the duties of the mutator and the collector. Wadler [WADL76] shows (analytically) that algorithms for performing garbage collection with time-sharing demand a greater percentage of the processing time than does classical sequential garbage collection. This is because the collection effort must proceed even when there is no demand for it.

A second approach for avoiding substantial program interruptions due to garbage collection has been proposed by Baker [BAKE78a, BAKE78b]. His method is an interesting modification of the collector described in Section 3. Baker's modification is such that each time a cell is requested (i.e., a *cons* is executed) a fixed number of cells, k, are moved from one semispace to the other. This implies that the two semispaces are simultaneously active. In a paging environment, the extra memory required is of less significance than the possible increase in the size of the average working set. Since the moved lists are compacted, page faults are likely to be minimized.

[12] A simulation is briefly reported in KUNG77

The moving of k cells during a *cons* corresponds to the tracing of that many cells in classical garbage collection. By distributing some of the garbage collection tasks during list processing, Baker's method provides a guarantee that actual garbage collection cannot last more than a fixed (tolerable) amount of time: the time to flip the semispaces and to readjust a fixed number of pointers declared in the user's program. Thus his algorithm may be used in real-time applications.

A characteristic of Baker's real-time algorithm is that the size of the semispaces may have to be increased, depending on the value of k and the type of list processing done by the program. In other words, the choice of k expresses the trade-off between the time to execute a *cons* and the total storage required. For example, for $k = \frac{1}{3}$, a cell is moved every third time a *cons* is called. This would speed up the computation but increase the amount of storage required.

In his paper Baker offers an informal proof of his algorithm's correctness and shows how it can be modified to handle varisized cells and arrays of pointers. He also presents analyses of storage requirements of the algorithm and how they compare with those of other garbage collection methods. A LISP machine built at M.I.T. used Baker's approach [BAWD77];[13] its memory is subdivided into areas, and a list of outgoing references is kept for each. Those areas which do not change during program execution are not copied: tracing starts from their corresponding list of outgoing references. This approach, which has been further developed by Bishop (see Section 3 and BISH77), is a possible alternative for real-time collection. Another alternative is the use of an auxiliary processor as suggested by Deutsch and Bobrow [DEUT76] in their incremental garbage collection technique mentioned in the previous section.

[13] This machine is a dedicated processor now in experimental operation The builders report that, immediately following a semispace flip, the system performance may be degraded. This is due to the copying of objects from the old semispace into the new one. A variant of Baker's approach [LIEB80] is now being implemented in the MIT LISP machine

6. ANALYSES

Execution of a list-processing program typically involves many garbage collections. Let n be the average number of cells which are marked in one classical collection of single-sized cells. Let m be the total number of cells in the memory. Therefore, on the average, $m - n$ cells are recovered during one garbage collection. Collection time can be expressed by

$$\text{collection time} = \alpha n + \beta(m - n),$$

where α is the average time taken to mark (and subsequently unmark) a used cell and β is the average time taken to collect a free cell. Each inaccessible cell is inspected only once. Since the time for marking is much greater than the time for reclaiming inaccessible cells, it is not unreasonable to assume that β is considerably smaller than α. If compaction is used, the pointers of n cells may have to be readjusted, thereby increasing even more the ratio of the coefficient of n and $m - n$. Detailed estimates for α and β have appeared in KNUT73, p. 592, and in BAER77.

The cost of collection per collected word is

$$\text{collection cost per collected word} = \frac{\alpha\rho}{1 - \rho} + \beta,$$

where ρ is the ratio n/m. If $\rho = \frac{1}{4}$, the memory is one-fourth full, and the cost is $\frac{1}{3}\alpha + \beta$. A larger value of ρ, for example, $\frac{3}{4}$, yields a larger cost $(3\alpha + \beta)$. This type of analysis, presented in KNUT73, shows how inefficient garbage collection can be when the memory becomes full.

Two new quantities N and T are now introduced. N stands for the total number of cells collected in the entire run of the program. T is the total time spent in useful program execution, excluding garbage collection. Then the total time for program execution is

$$\text{total program execution time} = N\left(\frac{\alpha\rho}{1 - \rho} + \beta\right) + T.$$

Let γ be the ratio T/N, that is, the useful computing time per word collected. Hoare [HOAR74] posits that the total cost of a program is proportional to the *product* of space and time:

$$\text{cost} = Nm\left(\frac{\alpha\rho}{1 - \rho} + \beta + \gamma\right). \qquad (1)$$

This function reaches a minimum with respect to m when

$$\rho = \frac{1}{1 + r} \quad \text{where} \quad r = \sqrt{\frac{\alpha}{\beta + \gamma}}.$$

Hoare's paper presents curves indicating how the cost varies with $1/\rho$ for various values of r. He points out that when $\alpha = 1$ and β is small compared to α, the extreme values of r are 1, and $\frac{1}{4}$. For these values of r, the cost curves are rather shallow around the optimum. Hoare suggests that a simple strategy for minimizing costs is to ensure that, after each collection, ρ lies between 0.6 and 0.8. If this does not occur, he recommends expanding or releasing the available memory so that ρ becomes approximately equal to 0.7. Hoare's analysis also indicates that the use of reference counters is justified only for programs whose value of r is close to one (i.e., γ is small).

Campbell [CAMP74] argues that Hoare's hypothesis of costs proportional to the product of time and space may be unrealistic. Campbell claims that, in certain large symbolic computations, *time*, rather than the product of space and time, should be minimized, since the amount of space needed to solve a problem is not subject to reduction. In these cases, the optimal strategy is to maximize the ratio of T to garbage collection time, that is,

$$\frac{\gamma}{\alpha\rho/(1 - \rho) + \beta}.$$

The above function has no extremum; according to Campbell, the recommended strategy is the "counsel of despair": choose m, the number of available cells, as large as possible.

Campbell also proposes a refinement of Hoare's analysis, that is, the one which minimizes the product of space and time. He notes that after a collection, a certain percentage, f, of the free list remaining from the previous collection is still free. Another percentage, g, of the rest of the storage

corresponds to allocated but inactive cells. Let F_j be the size of the free list after the jth collection. Then

$$F_j = fF_{j-1} + g(m - F_{j-1})$$
$$= gm + hF_{j-1}$$

and

$$F_0 = m.$$

Campbell uses the above difference equation in connection with Eq. (1) to obtain optimal strategies similar to Hoare's but involving the quantities f and g. He claims that when $f = 0.4$ and $h = 0.2$, the optimal costs correspond to values of ρ below 0.6. Campbell then suggests that the best rule of thumb is to consider $\rho = \frac{1}{2}$—to insure that, after each collection, half of the total number of cells be available in the free list.

In the final part of his paper, Campbell proposes yet another variant of Hoare's analysis. This variant is applicable when a user knows the approximate total number of cells, W, the program will request during its execution. Campbell points out that there are several symbolic computations for which W can be estimated, and this may be used to develop optimal strategies for selecting ρ.

Arnborg's analytical study of optimal strategies [ARNB74] yields results similar to Hoare's. Arnborg considers the time to collect to be a linear function of n only, n being the number of marked (or active) cells. Like Campbell, he establishes difference equations which express storage availability between successive collections. Arnborg, however, uses smooth functions to approximate the difference equations. His results are obtained by minimizing an integral which expresses the total costs of collecting and actual computing. Arnborg's strategy, like Hoare's, is to determine the best size for storage after each collection. The strategy has been implemented in a SIMULA compiler running on a PDP-10. He claims that his strategy gave consistently better results than ad hoc policies designed for specific programs.

In a recent paper, Larson [LARS77] proposes still another method for minimizing garbage collection time by suitably choosing the size m of storage available. Collection time is expressed by

$$\text{collection time} = \alpha' n + \beta' m,$$

where α' and β' are quite similar to a and b as defined in the beginning of this section: α' is the time to mark, compact, readjust pointers, and unmark an active cell; β' is the time to inspect each cell. As indicated previously, α' is substantially greater than β'.

Larson measures the computation effort by the amount of data which are produced by a program. In LISP, for example, this corresponds to the number of *cons*. Larson proposes using a smooth function $n(x)$ expressing the number of active cells at the point in the computation at which x cells have been produced.

The total garbage collection time is expressed by an integral of a function of $n(x)$, α', β', and m. When α' and β' are independent of m, the minimization of the integral leads to a strategy identical to Campbell's: m should be as large as possible. The results are somewhat different when α' and β' vary with m. This occurs when virtual memory is used, since the values of α' and β' depend on the relative amounts of fast and slow memory available. Larson's strategy is summarized as follows: if the number of active cells n approaches the number of cells in the fast memory (m_0), minimization occurs when $m = m_0$, and it is therefore preferable to use fast storage only.

The cost of garbage collection when using very large virtual memories has been studied by Bishop [BISH77]. He argues that there are two components of the cost: the time to perform the collection, and the overhead caused by the increase in page faults when garbage is left uncollected. As seen previously, the first component increases linearly with the number of active cells. Bishop claims that the second component increases more than linearly with the amount of existing garbage. He expresses the second component in the form cx^a, where x is the number of uncollected cells and c and a are parameters. Another important variable is r, the rate at which garbage is generated by a program. Bishop assumes that garbage collection is performed periodically, and he minimizes the

cost of collection with respect to the collection frequency. The optimal frequency is expressed as a function of the parameters a, c, r and the number of active cells n. Bishop's main result is that when $a \approx 1$ the cost of garbage collection (per cell collected) is proportional to n but inversely proportional to r, the rate of garbage generation. He then shows that the cost of collection can be reduced by segregating cells with different rates of garbage generation in separate areas of the memory.

Wadler [WADL76] presents two analyses of algorithms for real-time garbage collection. One applies to the Dijkstra-Steele method, which uses two parallel processors: the mutator and the collector. A *c-time* is defined by Wadler as the beginning of the collector's cycle. He also defines a *floating cell* as a cell which is marked by the mutator or collector at a *c-time* but is released before the beginning of the next cycle. Floating cells are momentarily useless since they are neither accessible by the collector nor available to the mutator. An extremely unfavorable situation for parallel garbage collection occurs when the only cells that are returned to the free list are the ones which were floating at a *c-time*. Even more unfortunately, Wadler's analytical study of the algorithm's average performance indicates that this unfavorable situation happens quite often.

Wadler then proceeds to define *power drain* as the ratio of the collector time to the mutator time. Using this definition, it is easy to show that the ratio of power drains between parallel and classical garbage collection can even be infinite: Consider, for example, the case where no cells are used or released. In classical garbage collection, the power drain is zero since the collector is never called. In parallel garbage collection, the power drain is one since the collector is kept busy even if it cannot retrieve any cells.

Wadler shows that when the two processors operate at maximum capacity,[14] the ratio of power drains is 2. This means that parallel garbage collection requires at least twice as much processing power as sequential garbage collection. He claims that with the falling cost of processors, this drawback is amply offset by the advantage of avoiding garbage collection interruptions.

Wadler also analyzes the algorithm in which the tasks of the mutator and the collector are time-shared by a single processor. He finds that in this case also, the power drain is 2 if the collector is not wasting time attempting to do unnecessary garbage collection.

7. REMARKS ON LANGUAGE IMPLEMENTATION

Recursion is frequently utilized in programs which manipulate linked-list structures. A stack is indispensable for executing these programs. Therefore, separate regions for the allocated cells and for the stack must coexist in the memory. It is true that stacks can be "simulated" by linked lists, so that the memory stores only list structures. However, this is both space- and time-consuming, because an extra field is required to link together the data in the stack and more complex operations are needed for pushing and popping. It is therefore simpler to implement the stack in contiguous positions of the memory.

It has become current practice to divide the available memory into two areas which are allowed to grow from opposite ends. One of these is reserved for a stack using contiguous locations. The other, called the *heap*, is available to the allocator for providing new cells, also from contiguous locations. With this arrangement, a simple test can be used to trigger garbage collection. When the pointer to the next free stack position meets the pointer to the next available position in the heap, collection with compaction is invoked to retrieve space for new cells or for stacking. Therefore, the functions *push* and *new* (for requesting new cells) may trigger garbage collection.

In the case of LISP programs, the function *new* corresponds to a *cons*, and *push* is used internally by the compiler or interpreter. Collection can be started either by a *cons* or by a stack overflow caused by situations such as great recursion depth or reading long atoms (see BERK64 and COHE72).

[14] This maximum capacity is determined analytically.

Since the stack is used in implementing recursion, it usually contains pointers to active, useful cells. The marking algorithms of Section 1 are used to mark not only the structures referred to by the pointer variables of a program but also those structures which are referred to by pointers on the stack. Therefore, means must be provided to recognize whether a stacked quantity is a pointer (tag bits may be used for this purpose).

LISP processors sometimes allow a user to invoke the collector. This is useful when he has an idea of the most propitious time for triggering the collection. Also, the function *return* may be made available to the user. When a free list is used, the returned cells can be immediately incorporated into the list. However, when compaction is required (e.g., with the heap and stack arrangement), the returned cells may not be available to the allocator until after the next collection. Another problem with the function *return* is that a cell may be explicitly returned even though there is still a pointer to it. (This is sometimes called the *dangling reference* problem.) Thus care must be taken not to reuse the cell until there are no pointers to it.

Processors for languages like PL/I and PASCAL allow a user to call the function *new* and provide messages when storage is exhausted. The use of the functions *return* or *collect* is implementation dependent. This author is unaware of PASCAL runtime systems which perform fully automatic garbage collection. It is the user's responsibility to keep free lists of unused cells and to check whether a new cell may be obtained from a free list or must be requested from the allocator. The techniques for doing this kind of storage management are beyond the scope of this paper.

Arnborg [ARNB72] described the implementation of a SIMULA compiler designed to operate in a virtual memory environment. SIMULA is a language with block structure: variables declared in a block or procedure exist only when the block or procedure is activated. Although it would seem at first sight that one could collect the structures referred to by pointer variables upon exiting from the block in which they are declared, this is not the case. SIMULA also allows variables of such types as classes, arrays, and texts which may have longer life spans than their originating blocks; if these variables share linked structures with local block variables, collection cannot be done when exiting from a block.

Since Arnborg's proposed implementation operated in a paging environment, one of the objectives of the collection is to reduce the number of page faults. To perform the collection, Arnborg uses a variant of the method proposed by Fenichel and Yochelson [FENI69] described in Section 3. The variant can handle varisized cells rather than only simple LISP cells.

A SNOBOL implementation proposed by Hanson [HANS77] uses a variation of the garbage collection techniques for collecting varisized cells described in Section 2. It is assumed that additional space for the heap and for the stack can be requested from the operating system, although such requests should be kept to a minimum. An effort is made to reduce collection time by avoiding marking cells which are known to be used throughout the program's execution.[15] For this purpose the heap is subdivided into two areas of consecutive locations: *heap* 1 and *heap* 2. The first contains information which is constantly active and never needs to be marked; the second may contain inactive cells which can be collected. New cells may be requested from either area.

Collection is triggered when one of the heaps runs out of space. The phases of marking and compacting are applied only to the information in *heap* 2, although tracing and pointer readjustment in *heap* 1 may be necessary. A fourth phase, "*moving heap* 2," may be necessary to make room for *heap* 1 when an overflow of the latter triggered the collection. The allocator of the operating system is called when no cells can be collected from *heap* 2.[16]

Certain list processors, including SNOBOL and LISP, need to keep symbol tables which are updated at execution time when new atoms are read. These symbol tables

[15] Certain implementations of LISP's list of atoms may take advantage of this feature.

[16] Note that Bishop's technique of keeping lists of interarea links (see BISH77) could be used to administer these heaps.

often utilize hashing techniques and keep linear linked lists of identifiers (atoms) having the same hash value. The linear lists are stored in the heap, and means must be provided to reclaim inactive list elements. This reclamation may be crucial in applications which use a large number of atoms. A scheme for collecting these atoms is proposed in FRIE76. A related problem is that of collecting LISP atoms whose property lists are shared by other atoms. A method for collecting nonshared atoms and their property lists is described in MOON74.

Next to LISP, ALGOL 68 is the language for whose garbage collection implementation the most literature exists. This is not surprising, since ALGOL 68 allows for a variety of complex situations because of the interaction of such features as block structure, references that can point to different types of cells, linked structures that may reside in the stack or in the heap, and sharing of arrays (slices). Both the implementor and the user of the language can take advantage of some of these features to minimize collection time.

In ALGOL 68, each element of a cell must be marked since structures may share parts of cells. (A separate bit table may be used for this purpose.) Wodon [WODO69] suggests two possible approaches for marking varisized cells in ALGOL 68. One is the "interpretive" approach represented by the program of Figure 2. The parameter p is specified by two components: the pointer, and the type of the cell being pointed to. This latter information could be stored in the cell itself, but it is more economical to precompute, at compile time, *templates* which list the characteristics of each cell type: size and a bit pattern specifying which elements of the cell are pointers. Determining these quantities is more complex when a pointer can refer to cells of different types. (Templates may also contain pointers to other templates which describe the kind of cell referenced by each pointer.) This marking approach is called interpretive because the information in the templates may have to be processed several times during execution.

The second approach suggested by Wodon is to *compile*, for each program, a more efficient marking routine specific for tracing the cells used in that program. Detailed descriptions (in ALGOL 68) of the interpretive and compiling approaches appear in BRAN71 and WODO71. It is believed that the compiling approach is more efficient than the interpretive one but requires additional storage for the local marking routines. The given references also propose using a compacting procedure requiring an external break table for readjusting pointers.

The collectors for ALGOL 68 proposed by Marshall [MARS71] and by Goyer [GOYE71] are also based on the classical techniques described in Section 2. The first uses the link-reversal technique for marking and Haddon and Waite's method [HADD67] for compaction and pointer readjustment. The second uses a stack for marking and a simplified version of Haddon and Waite's compacting procedure which requires an external break table. Note that the space used by the stack can be reused later by the break table. This additional space is needed only during garbage collection and can be returned to the operating system thereafter.

Baecker [BAEC70] makes recommendations on how to implement the ALGOL 68 heap in a computer with multilevel storage and which uses segmentation (i.e., addresses are given by an integer, referring to a segment, and an offset which specifies the location of a word within the segment). He also proposes introducing language constructs to allow a user to define different heap *areas* and to request that cells be allocated in specific areas of his choice [BAEC75].

8. FINAL REMARKS

Tables 1–5 summarize the characteristics of the main algorithms described in the corresponding Sections 1–5. The number of references presented in the bibliography bears witness to the importance of and interest in garbage collection. In spite of this activity, many facets of garbage collection remain to be investigated. In particular, no comparison has been made of the relative efficiencies of many of the algorithms described in Sections 1–5.

New developments in hardware are likely

Table 1

Algorithm	Main references	Auxiliary storage	Mark (M) or tag (T) bits	Complexity	Comments	Related work
Marking Single-Sized Cells[a] *(n is the total number of accessible cells)*						
Classical	KNUT73	Stack	M	$O(n)$	Stack may be as large as n	
Bounded workspace	KNUT73 (p. 415)	Limited-size stack	M	$O(n)$	Requires more time than the classical algorithm	KURO75, KURO79
Link reversal (Deutsch–Schorr–Waite)	SCHO67, KNUT73	No stack is needed	M and T	$O(n)$	As above (the tag bit may be replaced by the atom bit [KNUT73])	VEIL76, KOWA79, GERH79, GRIE79, LEE 79, TOPO79
Link reversal with bit stack	WEGB72b	Bit stack	M	$O(n)$	Similar to the above algorithm	SCHO67
Hybrid	KNUT73 (p. 592)	Limited-size stack	M and T	$O(n)$	Uses a combination of stack and link reversal	
Compaction of Single-Sized Cells (m is the total number of available cells)						
Two pointer	KNUT73 (p 421)	None	None	$O(m)$	Only applicable to single-sized cells Compaction is of arbitrary type[b]	HART64, COHE67b, DIJK76a
Moving Collectors of Single-Sized Cells (n is the total number of accessible cells)						
Minsky	MINS63	None	M	$O(n)$	Compaction of linearizing[b] type using LIFO traversal	
Fenichel–Yochelson	FENI69	Stack	None	$O(n)$	Uses two semispaces. Compaction of linearizing type[b] using LIFO traversal	MINS63
Cheney	CHEN70	None	None	$O(n)$	Compaction using FIFO traversal	WALD72
Reingold	REIN73	None	None	$O(n)$	Compaction using LIFO traversal and link reversal	SCHO67
Clark	CLAR76	None	None	$O(n)$	Compaction using LIFO traversal	

[a] These marking algorithms may be extended to mark varisized cells.
[b] See Section 1.

Table 2

Algorithm	Main references	Auxiliary storage	Mark (M) or tag (T) bits	Complexity	Comments	Related work
Marking Varisized Cells (n is the total number of accessible cells)						
Classical	KNUT73	Stack	M	$O(n)$	Stack can be stored using an additional field of each cell	THOR72, WODO71
Link reversal	THOR72, THOR76	No stack is needed	M and several T's	$O(n)$	Requires log *maxm* bits per cell, where *maxm* is the size of largest cell	FENI71, MARS71, WODO71, HANS77
Compacting Varisized Cells (m is the total number of available cells) All these compactors are of the sliding type[a]						
Rolling table (Haddon-Waite)	HADD67	None	None	$O(m \log m)$[b]	Break table (see Section 2.2) has to be rolled and sorted	FITC78
LISP 2	KNUT73 (p 602)	Additional word per cell	None	$O(m)$	Requires three or more scans	WEGB72a, THOR76, LANG72, FITC78, TERA78
Morris	MORR78	None	T	$O(m)$	Requires two scans	JONK79, WISE79
Jonkers	JONK79	None	None	$O(m)$	Requires two scans	
Moving (Copying) Collectors of Varisized Cells (n is the total number of accessible cells)						
Variant of Fenichel-Yochelson and Cheney	BAKE78	None	None	$O(n)$	Moves cells in a breadth-first order	FENI69, ARNB72, CHEN70
Standish	STAN80	One field per cell	None	$O(n)$	Copies using three passes	THOR72

[a] See Section 1.
[b] Worst case.

Table 3. Collecting in Virtual Memory

Algorithm	Main references	Auxiliary storage	Comments	Related work
Baker	BAKE78	None	Uses two semispaces	FENI69, CHEN70
Bishop	BISH77	Space for keeping interarea lists	Designed for use in very large virtual memories	

Table 4. Reference Counters (m is the number of available cells)

Algorithm	Main references	Storage needed	Comments	Related work
Classical	COLL60, WEIZ63	An extra field (of size m) per cell	Cannot handle general circular lists	KNUT73, WEIZ69
Hybrid	KNUT73, DEUT76	An extra field (of size $m' \ll m$) per cell Auxiliary Tables	Combines reference counters with classical compacting garbage collection	WISE77

Table 5. Parallel and Real-Time Collection

Algorithm	Main references	Storage needed	Comments	Related work
Parallel (Dijkstra)	DIJK76b	No stack and two bits per cell	Main objective is to prove correctness; uses a free list	MULL76, GRIE77, FRAN78, KUNG77
Parallel (Steele)	STEE75	Stack, two bits per cell, and several semaphores	Designed to be microcoded, does compacting as well	WADL76, DIJK76b
Baker	BAKE78	Two semispaces whose sizes vary at execution time	Moving of accessible cells is done when a new cell is requested	MINS63, FENI69, CHEN70

to play an important role in speeding up collection. It has already been suggested that new machines should contain extra bits per word to be used for marking, tagging, or counting references. Machines with special hardware for segmentation and list processing have recently been constructed [BAWD77] and are now in experimental operation.

There has been an undeniable trend toward designing and implementing collectors for varisized cells stored in large virtual memories. No explicit guidance based on experimental evidence is yet available on how to do this collection efficiently or in real time. Two promising directions, discussed in Section 5, involve either using parallel processors or distributing some of the garbage collection tasks during the actual processing. It is hoped that this will allow the collection to be performed within a known, tolerable, maximum time.

Collection in very large virtual memories is another subject which will become increasingly important. The suggested approaches for these collections deserve further study [BISH77].

If these efforts in the direction of achieving efficient garbage collection succeed, they are bound to have an impact on the design of future programming languages.

ACKNOWLEDGMENTS

Joel Katcoff scrutinized every paragraph of the original and revised manuscripts. His stress on clarity and simplicity, coupled with his constructive remarks, was of great value in producing a better paper The referees' and editor's comments made many other improvements possible. In particular, one of the referees, Peter Bishop, provided several pages of detailed suggestions on how to reorganize and make more precise the contents of the paper. A second referee also provided numerous constructive remarks. The author

learned a great deal from these people, and this paper benefited greatly from their help. Carolyn Boettner's aid in preparing the final version of the manuscript is gratefully acknowledged. Finally, the author wishes to thank Jane Jordan for the care and patience with which she typed the text and its several revisions.

This work was supported by the National Science Foundation under grants MCS 74-24569 A01 and MCS 79-05522.

REFERENCES

The bibliography which follows includes a few references which are not explicitly mentioned in the text. Each reference is associated with a profile consisting of a sequence of letters between braces. The letters characterize the contents of the paper and their relationship to the topics covered in this survey.

A: Analysis
B: Benchmarks
C: Compacting
G: General
L: Language Features and Implementation
M: Marking
N: Reference Counters
P: Parallel and Real-Time Processing
R: Records or Varisized Cells
S: Copying and Secondary Storage
V: Virtual Memory

ARNB72 ARNBORG, S. "Storage administration in a virtual memory simulation system," *BIT* **12,** 2 (1972), 125–141 {CGLMRV}

ARNB74 ARNBORG, S. "Optimal memory management in a system with garbage collection," *BIT* **14,** 4 (1974), 375–381. {A}

AUGE79 AUGENSTEIN, M. J., AND TENENBAUM, A. M. *Data structures and PL/I programming*, Prentice-Hall, Englewood Cliffs, N.J., 1979 {GMN}

BAEC70 BAECKER, H D. "Implementing the ALGOL 68 heap," *BIT* **10,** 4 (1970), 405–414. {GLV}

BAEC72 BAECKER, H.D "Garbage collection for virtual memory computer systems," *Commun. ACM* **15,** 11 (Nov. 1972), 981–986. {BCMRV}

BAEC75 BAECKER, H.D. "Areas and record-classes," *Comput J.* **18,** 3 (Aug. 1975), 223–226. {GL}

BAER77 BAER, J.L., AND FRIES, H. "On the efficiency of some list marking algorithms," in *Information processing 1977*, B. Gilchrist (Ed.), IFIP, North-Holland, Amsterdam, 1977, pp 751–756. {ABM}

BAKE78a BAKER, H.G. "Actor systems for real time computation," Lab. for Computer Science, MIT Rep. TR-197, M.I.T., Cambridge, Mass., March 1978 (see BAKE78b)

BAKE78b BAKER, H.G. "List-processing in real time on a serial computer." *Commun. ACM* **21,** 4 (April 1978), 280–294 {ACGMNPRSV}

BART77 BARTH, J.M. "Shifting garbage collection overhead to compile time," *Commun. ACM* **20,** 7 (July 1977), 513–518. {LN}

BAWD77 BAWDEN, A., GREENBLATT, R., HOLLOWAY, J., KNIGHT, T., MOON, D., AND WEINREB, D. "Lisp machine progress report," Memo 444, A.I Lab, M.I.T., Cambridge, Mass., Aug. 1977 {G}

BERK64 BERKELEY, E.C., AND BOBROW, D G. (Eds.) *The programming language LISP*, M.I.T., Cambridge, Mass. 1974, 4th printing. {GL}

BERR78 BERRY, D M., AND SORKIN, A. "Time required for garbage collection in retention block-structures languages," *Int. J. Comput. Information Sci.* **7,** 4 (1978), 361–404. {AL}

BERZ75 BERZTISS, A.T. *Data structures theory and practice*, 2nd ed., Academic Press, New York, 1975. {G}

BISH77 BISHOP, P.B. "Computer systems with a very large address space and garbage collection," Lab. for Computer Science, MIT Rep., TR-178, M.I.T., Cambridge, Mass., May 1977. {ACGLMRSV}

BOBR67 BOBROW, D.G., AND MURPHY, D.L. "Structure of a LISP system using two-level storage," *Commun. ACM* **10,** 3 (March 1967), 155–159. {V}

BOBR68a BOBROW, D.G. *Storage management in Lisp, in symbol manipulation languages and techniques*, D. G. Bobrow (Ed.), North-Holland, Amsterdam, 1968. {CGMV}

BOBR68b BOBROW, D.G., AND MURPHY, D.L. "A note on the efficiency of a LISP computation in a paged machine," *Commun. ACM* **11,** 8 (Aug. 1968), 558–560. {V}

BOBR75 BOBROW, D.G. "A note on hash linking," *Commun. ACM* **18,** 7 (July 1975), 413–415 {N}

BOBR80 BOBROW, D.G. "Managing reentrant structures using reference counts," *ACM Trans. Programming Lang. Syst.* **2,** 3 (July 1980), 269–273. {N}

BRAN71 BRANQUART, P., AND LEWI, J. "A scheme of storage allocation and garbage collection for Algol 68," in *Algol 68 implementation*, J. E L. Peck, (Ed.), North-Holland, Amsterdam, 1971, pp. 199–238. {CGLMR}

CAMP74 CAMPBELL, J.A. "Optimal use of storage in a simple model of garbage collection," *Inf. Process Lett.* **3,** 2 (Nov. 1974), 37–38. {A}

CHEN70 CHENEY, C.J. "A nonrecursive list compacting algorithm," *Commun. ACM* **13,** 11 (Nov. 1970), 677–678. {CRSV}

CLAR75 CLARK, D.W. "A fast algorithm for copying binary trees," *Inf. Process. Lett* **9,** 3 (Dec. 1975), 62–63. {AC}

CLAR76 CLARK, D.W. "An efficient list moving algorithm using constant workspace," *Commun. ACM* **19,** 6 (June 1976), 352–354. {CRS}

CLAR77 CLARK, D.W., AND GREEN, C.C. "An empirical study of list structure in Lisp," *Commun. ACM* **20,** 2 (Feb. 1977), 78–86. {BV}

CLAR78a CLARK, D.W. "A fast algorithm for copying list structures," *Commun ACM* **21,** 5 (May 1978), 351–357. {ACRS}

CLAR78b CLARK, D.W., AND GREEN, C.C. "A note on shared list structure in Lisp," *Inf. Process. Lett* **7,** 6 (Oct. 1978), 312–314. {B}

CLAR79 CLARK, D.W. "Measurements of dynamic list structure in Lisp," *IEEE Trans Softw. Eng.* **SE-5,** 1 (Jan 1979), 51–59. {BV}

COHE67a COHEN, J. "Use of fast and slow memories in list-processing languages," *Commun. ACM* **10,** 2 (Feb. 1967), 82–86. {V}

COHE67b COHEN, J., AND TRILLING, L. "Remarks on garbage collection using a two level storage," *BIT* **7,** 1 (1967), 22–30. {BCMV}

COHE72 COHEN, J., AND ZUCKERMAN, C. "Evalquote in simple Fortran A tutorial on interpreting Lisp," *BIT* **12,** 3 (1972), 299–317. {CGM}

COLL60 COLLINS, G.E. "A method for overlapping and erasure of lists," *Commun ACM* **3,** 12 (Dec 1960), 655–657. {N}

DEUT76 DEUTSCH, L.P., AND BOBROW, D G "An efficient incremental automatic garbage collector," *Commun ACM* **19,** 9 (Sept. 1976), 522–526. {CGLNV}

DIJK76a DIJKSTRA, E.W. *A discipline of programming*, Prentice-Hall, Englewood Cliffs, N J., 1976, Chap. 14. {G}

DIJK76b 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," in *Lecture Notes in Computer Science*, No. 46, Springer-Verlag, New York, 1976, also appeared in *Commun. ACM* **21,** 11 (Nov. 1978), 966–975. {P}

DWYE73 DWYER, B. "Simple algorithms for traversing a tree without an auxiliary stack," *Inf. Process. Lett.* **2,** 5 (Dec. 1973), 143–145. {M}

ELSO75 ELSON, M. "Data structures," Science Research Associates, 1975. {CGMN}

FENI69 FENICHEL, R., AND YOCHELSON, J. "A LISP garbage-collector for virtual-memory computer systems," *Commun. ACM* **12,** 11 (Nov. 1969), 611–612. {CSV}

FENI71 FENICHEL, R. "List tracing in systems allowing multiple cell-types," *Commun. ACM* **14,** 8 (Aug. 1971), 522–526. {MLR}

FISH74 FISHER, D.A. "Bounded workspace garbage collection in an address order preserving list processing environment," *Inf. Process. Lett.* **3,** 1 (July 1974), 29–32. {CMR}

FISH75 FISHER, D.A. "Copying cyclic list structure in linear time using bounded workspace," *Commun. ACM* **18,** 5 (May 1975), 251–252. {CS}

FITC78 FITCH, J.P., AND NORMAN, A.C. "A note on compacting garbage collection," *Comput. J.* **21,** 1 (Feb. 1978), 31–34. {ABCR}

FOST68 FOSTER, J.M. *List processing*, Elsevier Computer Monographs, Elsevier-North Holland, New York, 1968. {G}

FRAN78 FRANCEZ, N. "An application of a method for analysis of cyclic programs," *IEEE Trans. Softw. Eng.* **4,** 5 (Sept. 1978), 371–377. {P}

FRIE76 FRIEDMAN, D.P., AND WISE, D.S. "Garbage collecting a heap which included a scatter table," *Inf. Process. Lett.* **5,** 6 (Dec 1976), 161–164. {LM}

FRIE79 FRIEDMAN, D.P., AND WISE, D.S. "Reference counting can manage the circular environments of mutual recursion," *Inf. Process. Lett.* **8,** 1 (Jan. 1979), 41–45. {N}

GERH79 GERHART, S.L. "A derivation oriented proof of Schorr-Waite marking algorithm," in *Lecture notes in computer science*, vol. 69, Springer-Verlag, New York, 1979, pp 472–492. {M}

GOTL78 GOTLIEB, C C., AND GOTLIEB, L.R *Data types and structures*, Prentice-Hall, Englewood Cliffs, N.J., 1978. {CGMR}

GOYE71 GOYER, P. "A garbage collector to be implemented on a CDC 3100," in *Algol 68 implementation*, J. E. L. Peck (Ed.), North-Holland, Amsterdam, 1971, pp 303–317. {CLMR}

GRIE77 GRIES, D. "An exercise in proving parallel programs correct," *Commun. ACM* **20,** 12 (Dec. 1977), 921–930 {P}

GRIE79 GRIES, D. "The Schorr-Waite graph marking algorithm," *Acta Inf.* **11,** 3 (1979), 223–232. {M}

GRIS72 GRISWOLD, R.E. *The macro implementation of Snobol 4*, W. H. Freeman, San Francisco, 1972. {GM}

HADD67 HADDON, B.K., AND WAITE, W M. "A compaction procedure for variable length storage elements," *Comput. J.* **10** (Aug. 1967), 162–165. {CR}

HANS69 HANSEN, W.J. "Compact list representation· Definition, garbage collection, and system implementation," *Commun. ACM* **12,** 9 (Sept. 1969), 499–507. {CGRS}

HANS77 HANSON, D.R. "Storage management for an implementation of Snobol 4," *Software: Practice and Experience* **7,** 2 (1977), 179–192. {BCMLR}

HART64 HART, T.P., AND EVANS, T.G. "Notes on implementing lisp for the M 460 computer," in BERK64. {C}

HOAR74 HOARE, C.A.R. "Optimization of store size for garbage collection," *Inf Process Lett* **2,** 6 (April 1974), 165–166. {A}

HORO77 HOROWITZ, E., AND SAHNI, S *Fundamentals of data structure*, Com-

puter Science Press, Woodland Hills, Calif., 1977. {ACGMR}

JONK79 JONKERS, H.B.M. "A fast garbage compaction algorithm," *Inf Process. Lett.* **9,** 1 (July 1979), 26-30. {CR}

KAIN69 KAIN, Y. "Block structures, indirect addressing, and garbage collection," *Commun. ACM* **12,** 7 (July 1969), 395-398. {L}

KNUT73 KNUTH, D E. *The art of computer programming, vol. I. Fundamental algorithms,* Addison-Wesley, Reading, Mass., 1973 {ACGMNPRS}

KOWA79 KOWALTOWSKI, T "Data structures and correctness of programs," *J. ACM* **26,** 2 (April 1979), 283-301 {M}

KUNG77 KUNG, H.T., AND SONG, S.W "An efficient parallel garbage collection system and its correctness proof," Dep Computer Sci., Carnegie-Mellon Univ., Pittsburgh, Sept. 1977. {AP}

KURO75 KUROKAWA, T. "New marking algorithms for garbage collection," Collection, in *Proc. 2nd USA-Japan Computer Conf*, 1975, pp. 580-584. {BM}

KURO79 KUROKAWA, T. "A new fast and safe marking algorithm," Toshiba R&D Center, Kawasaki 210, Japan, Jan. 1979. {BM}

LAMP76 LAMPORT, L. "Garbage collection with multiple processes: An exercise in parallelism," *Proc. IEEE Conf. Parallel Processing,* Aug 1976. {P}

LANG72 LANG, B., AND WEGBREIT, B. "Fast compactification," Rep. 25-72, Harvard Univ., Cambridge, Mass., Nov. 1972. {CMR}

LARS77 LARSON, R.G "Minimizing garbage collection as a function of region size," *SIAM J. Computing* **6,** 4 (Dec. 1977), 663-668. {AV}

LEE79 LEE, S., DE ROEVER, W.P, AND GERHART, S. "The evolution of list-copying algorithms," in *6th ACM Symp Principles of Programming Languages* (San Antonio, Tex), Jan. 1979, pp. 53-56. {MS}

LEE80 LEE, K.P. "A linear algorithm for copying binary trees using bounded workspace," *Commun. ACM* **23,** 3 (March 1980), 159-162 {S}

LIEB80 LIEBERMAN, H., AND HEWITT, C. "A real-time garbage collector that can recover temporary storage quickly," MIT Lab for Computer Science Rep TM-184, M.I.T., Cambridge, Mass., July 1980. {PV}

LIND73 LINDSTROM, G. "Scanning list structures without stacks or tag bits," *Inf. Process Lett* **2,** 2 (June 1973), 47-51. {M}

LIND74 LINDSTROM, G "Copying list structures using bounded workspace," *Commun. ACM* **17,** 4 (April 1974), 198-202. {CS}

MARS71 MARSHALL, S. "An Algol-68 garbage collector," in *Algol 68 implementation,* J. E L. Peck (Ed), North-Holland, Amsterdam, 1971, pp. 239-243. {CLMR}

MINS63 MINSKY, M.L. "A Lisp garbage collector algorithm using serial secondary storage," Memo 58 (rev.), Project MAC, M.I T., Cambridge, Mass., Dec. 1963. {CS}

MOON74 MOON, D.A. "MACLisp reference manual," Project MAC, M.I.T, Cambridge, Mass., April 1974. {GM}

MORR78 MORRIS, F L. "A time- and space-efficient garbage compaction algorithm," *Commun. ACM* **21,** 8 (Aug 1978), 662-665. {CGR}

MORR79 MORRIS, F.L. "On a comparison of garbage collection techniques," technical correspondence, *Commun ACM* **22,** 10 (Oct. 1979), 571. {C}

MULL76 MULLER, K G. "On the feasibility of concurrent garbage collection," Ph.D. thesis, Tech Hogeschool Delft, March 1976. {P}

OWIC81 OWICKI, S "Making the world safe for garbage collection," in *Proc ACM Symp. Principles of Programming Languages* (Williamsburg), Jan. 1981. {LP}

PARE68 PARENTE, R J. "A simulation-oriented memory allocation algorithm," in *Simulation Programming Languages,* J. N. Buxton (Ed), North-Holland, Amsterdam, 1968, pp 198-209 {CL}

PFAL77 PFALZ, J L *Computer data structures,* McGraw-Hill, New York, 1977. {CGMR}

REIN73 REINGOLD, E.M. "A nonrecursive list moving algorithm," *Commun. ACM* **16,** 5 (May 1973), 305-307. {CS}

ROBS73 ROBSON, J M. "An improved algorithm for traversing binary trees without auxiliary stack," *Inf. Process Lett.* **2,** 1 (March 1973), 12-14 {M}

ROBS77 ROBSON, J.M. "A bounded storage algorithm for copying cyclic structures," *Commun. ACM* **20,** 6 (June 1977), 431-433 {CSR}

ROCH71 ROCHFELD, A "New LISP techniques for a paging environment," *Commun. ACM* **14,** 12 (Dec 1971), 791-795. {V}

ROSS67 ROSS, D.T "The AED free storage package," *Commun. ACM* **10,** 8 (Aug. 1967), 481-492. {GR}

SCHO67 SCHORR, H., AND WAITE, W. "An efficient machine-independent procedure for garbage collection in various list structures," *Commun. ACM* **10,** 8 (Aug. 1967), 501-506. {MR}

SIKL72 SIKLOSSY, L "Fast and read-only algorithms for traversing trees without an auxiliary stack," *Inf Process. Lett.* **1,** 4 (June 1972), 149-152. {M}

STAN80 STANDISH, T.A. *Data structures techniques,* Addison-Wesley, Reading, Mass., 1980. {ACGMNPRSV}

STEE75 STEELE, G.L. "Multiprocessing compactifying garbage collection," *Commun ACM* **18,** 9 (Sept. 1975), 495–508. {CGP}

TERA78 TERASHIMA, M., AND GOTO, E. "Genetic order and compactifying garbage collectors," *Inf. Process. Lett.* **7,** 1 (Jan. 1978), 27–32. {CR}

THOR72 THORELLI, L.E. "Marking algorithms," *BIT* **12,** 4 (1972), 555–568. {MR}

THOR76 THORELLI, L.E "A fast compactifying garbage collector," *BIT* **16,** 4 (1976), 426–441 {CMR}

TOPO79 TOPOR, R. "The correctness of the Schorr–Waite list marking algorithm," *Acta Inf.* **11,** 3 (1979), 211–221 {M}

VEIL76 VEILLON, G. "Transformations de programmes recursifs," *R.A.I.R.O. Informatique* **10,** 9 (Sept. 1976), 7–20. {M}

WADL76 WADLER, P.L. "Analysis of an algorithm for real time garbage collection," *Commun. ACM* **19,** 9 (Sept 1976), 491–500. {AP}

WAIT73 WAITE, W.M. *Implementing software for non-numeric applications*, Prentice-Hall, Englewood Cliffs, N.J, 1973. {GCR}

WALD72 WALDEN, D.C. "A note on Cheney's nonrecursive list-compacting algorithm," *Commun ACM* **15,** 4 (April 1972), 275. {CS}

WEGB72a WEGBREIT, B. "A generalized compactifying garbage collector" *Comput. J.* **15,** 3 (Aug. 1972), 204–208. {CGMR}

WEGB72b Wegbreit, B. "A space efficient list structure tracing algorithm," *IEEE Trans Computers* **C21** (Sept. 1972), 1009–1010 {M}

WEIS67 WEISSMAN, C. *Lisp 1.5 primer*, Dickenson Publ., Belmont, Calif, 1967. {G}

WEIZ63 WEIZENBAUM, J. "Symmetric list processor," *Commun ACM* **6,** 9 (Sept. 1963), 524–544. {LN}

WEIZ69 WEIZENBAUM, J. "Recovery of reentrant list structures in SLIP," *Commun. ACM* **12,** 7 (July 1969), 370–372. {LMN}

WISE77 WISE, D.S., AND FRIEDMAN, D.P. "The one-bit reference count," *BIT* **17,** 4 (1977), 351–359. {GLN}

WISE79 WISE, D.S. "Morris' garbage compaction algorithm restores reference counts," *ACM Trans. Programm Lang Syst.* **1,** 1 (July 1979), 115–120. {CNR}

WODO69 WODON, P.L. "Data structure and storage allocation," *BIT* **9,** 3 (1969), 270–282. {CGLMR}

WODO71 WODON, P.L. "Methods of garbage collection for Algol 68," in *Algol 68 implementation*, J. E. L Peck (Ed.), North-Holland, Amsterdam, 1971, pp. 245–262. {CGLMR}

YELO77 YELOWITZ, L., AND DUNCAN, A G "Abstractions, instantiations and proofs of marking algorithms," in *Proc. Symp. Artificial Intelligence and Programming Languages, Sigplan Notices* (ACM) **12,** 8 (Aug. 1977), 13–21

ZAVE75 ZAVE, D.A. "A fast compacting garbage collector," *Inf. Process Lett.* **3,** 6 (July 1975), 167–169. {CMR}