M. D. McILROY, Editor

An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures

H. Schorr
IBM Corp., Menlo Park, California
AND
W. M. Waite
University of Colorado, Boulder, Colorado*

A method for returning registers to the free list is an essential part of any list processing system. In this paper, past solutions of the recovery problem are reviewed and compared. A new algorithm is presented which offers significant advantages of speed and storage utilization. The routine for implementing this algorithm can be written in the list language with which it is to be used, thus insuring a degree of machine independence. Finally, the application of the algorithm to a number of different list structures appearing in the literature is indicated.

1. Introduction

One of the most important features of a list language is its ability to allocate storage dynamically during the running of the object program. This is accomplished by means of a list of *available space* (or *free* list) which contains those registers not being used. Initially, the free list contains all storage not occupied by the program [1, 2, 11], and registers are detached from it and formed into list structures as the program is executed. The execution of the program must usually be suspended when the free list is exhausted, and the problem arises of reclaiming those parts of the list structure which are no longer needed (if any such exist). In this paper a statement of the difficulties involved is followed by a brief review of the solutions which have been proposed. A new, machine-independent procedure and the results obtained using the 7094 version of this procedure are presented. Finally, the application and modification of this routine for a variety of list structures is discussed.

* Department of Electrical Engineering

2. Statement of the Problem and Review of Past Solutions

The major problem which arises when attempting to reclaim a part of a list structure is that of knowing which part is no longer needed. This has received considerable attention in the literature [1–8] and three solutions have been proposed. The first, by Newell, Simon and Shaw [1], places the responsibility on the programmer. Their language (IPL-V) includes instructions which cause lists and list structures to be erased, thereby returning their registers to the free list. This approach is unattractive because it requires the programmer to keep track of the status of lists, sublists, etc. For example, part of a list may be shared with several other lists and might still be needed, while the remainder could be erased.

A second solution for systems which use shared sublists, originally due to Gerlernter et al. [3], extended by Collins [4], and used by Weizenbaum [5, 6], requires keeping a count of the references made to a list and salvaging the registers when the count reaches zero. In a one-way list structure it is impossible to locate the head of a list when a reference is made to some register along the list. Thus the part which starts from the referenced register must be treated as a new list, and a new reference counter must be set up. The proliferation of reference counters, and the large amount of bookkeeping involved, makes this method extremely cumbersome. In a two-way list [6] it is always possible to locate the head, and thus it is not necessary to set up a new reference counter. However, the head of the list must be found and the reference count increased by 1. Besides being time-consuming, this may prevent returning part of a list to the free list. For example, in Figure 1 there is no way of discarding the top part of list B if list A still needs the bottom part of B. Thus in practice this part of B has to be treated as a separate list and a new reference counter is needed [5, 6]. The reference counter method breaks down completely in the case of a circular list (i.e., one in which the list is a sublist of itself). In this situation the reference counter cannot be decreased to zero, even though the entire list may become inaccessible [13].

The third solution, which appears to be the most attractive, was proposed by McCarthy [2] and is considered in detail below. In this method, no reference counters are kept and registers are not returned to the free list until the latter has been exhausted. Then a procedure known as "garbage collection" is initiated, which traces the entire list structure, marking those registers which are attached to some list. Registers no longer needed will not be attached to any list and will thus remain unmarked. When all lists have been traced, a routine is entered to form all unmarked registers into a new free list and erase the marks in all others (in case the garbage collector has to be used again later).

Several difficulties arise in using and implementing a garbage collection procedure.

(A) The basic problem is the tracing of the lists. In general the lists will be branched and all branches must be traced. Several methods have been suggested, but all require either a significant amount of additional storage (for remembering the branch points encountered) or the retracing of large portions of the list structure many times.

(B) A second problem arises when the data consists of signed numbers which are stored in a whole word. In McCarthy's method of garbage collection, the sign of each register that is attached to a list is set to minus, while unattached registers remain positive. The sign of an attached register is then reset to plus after the new free list has been formed. Clearly, this procedure, unless modified, will result in reversing the sign of any negative numbers. The modification proposed by McCarthy is to reserve a block of storage that is to be used exclusively for whole-word data items (*full-word space*). A second part of the store is then set aside for a bit table, to be used during garbage collection to record which of the full words are still part of active list structures. This modification is not completely satisfactory since (1) it violates the dynamic storage allocation principle of list processing languages, and (2) the storage used by the bit table and the additional part of the garbage collection program which utilizes it must be taken away from the free list.

(C) Besides single full-word data items, multiple-word list elements have been proposed, and the number of

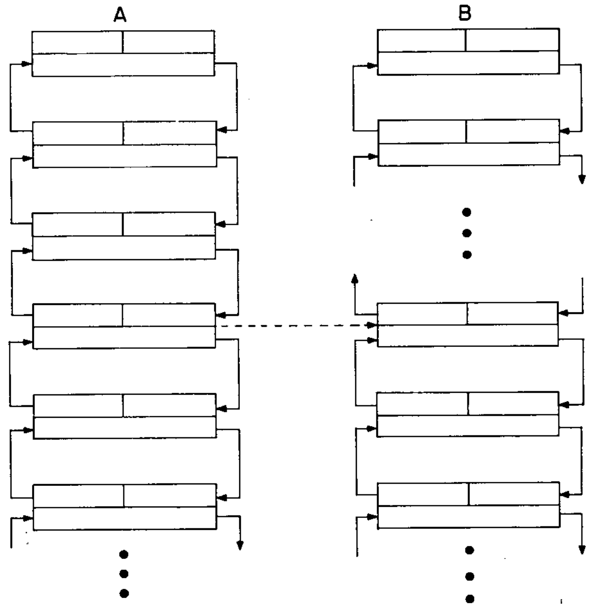


FIG. 1. Sharing a sublist in a two-way list structure

words making up such an element can be allowed to vary from element to element. These variable-length elements were proposed by Comfort [9], who also gave a solution to the problem which is created of maintaining a generalized free list. In such a free list, an item of arbitrary length may be taken off, and to insure that large elements are available it is important to try to reconstruct the largest possible blocks of consecutive free registers when returning registers to the free list. The problem of using a garbage collection procedure to achieve this is discussed below.

3. A Technique for Garbage Collection in a One-Way List

Solution of the basic problem (see Section 2A) of garbage collection requires a routine that is economical both in its use of temporary storage and in the number of times it traverses the list structure. In addition, of course, it must be able to trace any possible list. A routine which uses a limited amount of temporary storage for remembering the location of branch points cannot trace a list which has too many such points, and a recursive routine requires a push-down stack of indeterminate length on which to store return addresses. Moreover, if a routine must traverse a particular list an indeterminate number of times it will fail when it encounters a circular list. In connection with an implementation of the WISP language [7, 8] for the IBM 7094, a method has been developed which uses two index registers and the accumulator for temporary storage and which traverses the list structure twice. This routine is capable of tracing any list structure (including circular lists) which has any number of branch points.

In the WISP system as programmed for the 7094, the address field (TAIL) of a list element contains the location of the next register on the list and is called a pointer. The decrement field (HEAD) contains either an atom [2] or the location of a branch of the list. A WISP list is shown in Figure 2(a). The prefix and tag fields are unused and hence are available for use by the garbage collector. The algorithm employed here is to move down a list ignoring sublists and reversing the pointers as one goes. This reversal of pointers in a one-way list structure with shared sublists permits a return to be made to the head of a list during garbage collection. During this first pass, the sign of each register is made negative. The end of the first pass is reached when either (1) the end of the list is reached, or (2) a register on the list is encountered whose sign is minus (indicating that this part of the list is a sublist of some other list that has already been traversed). The result of applying this forward scan to the list D shown in Figure 2(a) is given in Figure 2(b).

During the forward scan of the list, the contents of the HEAD of each list register are ignored. The reverse scan, which is entered when the forward scan is terminated, moves back up the list restoring the original pointers. In addition, it examines the HEAD of each register, checking to see if it contains a reference to a sublist. (In the 7094 implementation of WISP, any number larger than 2200 is a pointer to a sublist, while any number less than this is an atom.) If such a register is found, its prefix is changed to indicate that the register is a branch point (see Figure 2(c)). The scan is then reversed and a forward scan of the sublist made. The list structure which results after the second forward scan has been completed upon the list D of Figure 2(a) is shown in Figure 2(d). The reverse scan of a sublist will pick up any sublists of that sublist. Eventually all sublists will have been traced, and the reverse scan will return to the first register of the main list being traced, whereupon the next list may be marked. After all the lists are traced, a sequential scan of memory is made. Any register encountered whose sign is positive is placed on the free list; any negative register must be attached to a list still in use and hence its sign is reset to plus. After construction of the new free list is completed, execution of the program is resumed.

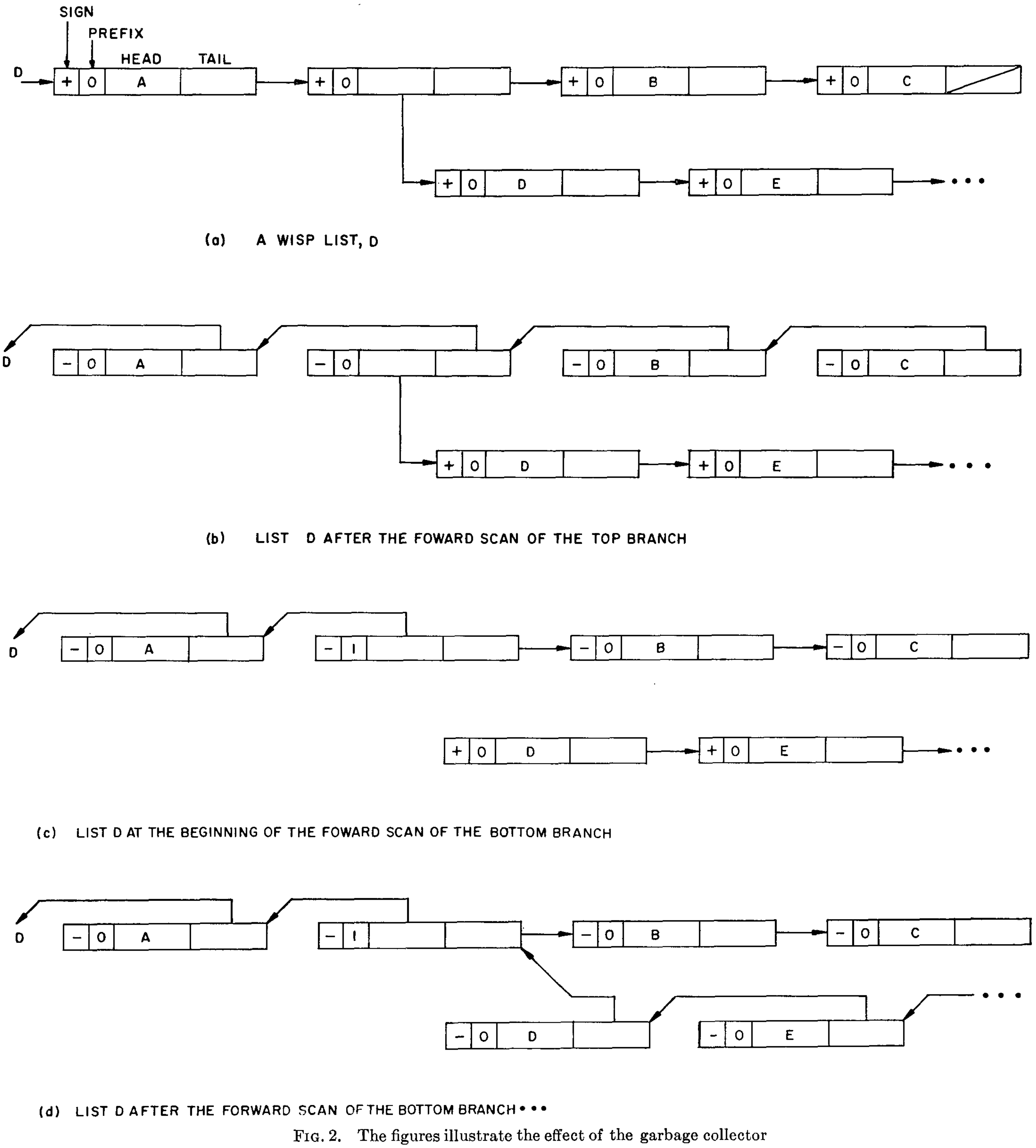

SIGN
PREFIX
HEAD
TAIL
D
+ 0 A
+ 0
+ 0 B
+ 0 C
+ 0 D
+ 0 E
(a) A WISP LIST, D
- 0 A
- 0
- 0 B
- 0 C
(b) LIST D AFTER THE FOWARD SCAN OF THE TOP BRANCH
- 1
(c) LIST D AT THE BEGINNING OF THE FOWARD SCAN OF THE BOTTOM BRANCH
- 0 D
- 0 E
(d) LIST D AFTER THE FORWARD SCAN OF THE BOTTOM BRANCH • • •

Fig. 2. The figures illustrate the effect of the garbage collector

4. The Routine and its Efficiency

The routine itself is flowcharted in Figure 3. For convenience in drawing the flowchart, the list elements are assumed to be numbered sequentially. Thus, list element I +1 follows list element I (i.e., list element I +1 is pointed to by the TAIL of list element I). The routine uses two index registers and the accumulator for temporary storage, though in general any three storage locations could be used. One contains the address of the previous list element examined, the second the address of the element currently being examined and the third the address of the next element on the list. This is necessary for the reversal of the pointers during the forward scan and their restoration during the reverse scan.

In order to evaluate the speed of the routine, a WISP program was written which created five complete binary trees of depth 12. The remaining registers were discarded and the garbage collector called. Thus it was forced to trace a list structure containing over 20,000 registers, half of which were branch points. The elapsed time, according to the system clock, was 1.85 seconds. This list structure seems far more complex than any which would be encountered in practice, and therefore a normal garbage collection should take far less time. The space occupied is also nominal—68 words for the routine itself, in addition to the two index registers and accumulator.

For purposes of comparison, a trace routine proposed by Wilkes was coded for the 7094. It required only two temporary storage locations, and occupied 35 words of memory. This routine traversed a path from the head of a list to each terminal register separately. Thus much of the list was scanned many times, and the program would fail by entering a loop when attempting to trace a circular list. When run under the above conditions (5 binary trees of depth 12) the routine required 2.75 seconds for a complete trace.

As a final comparison, a routine which stored branch points was coded and run using the same list structure. The program occupied 34 words, and an additional 48 words were allotted as a storage area for the branch points. Any given part of the list structure was only traversed once, so that this routine could trace any list for which the number of branch points it was required to store was less than 49. Only .448 seconds were required to complete the trace of the test structure.

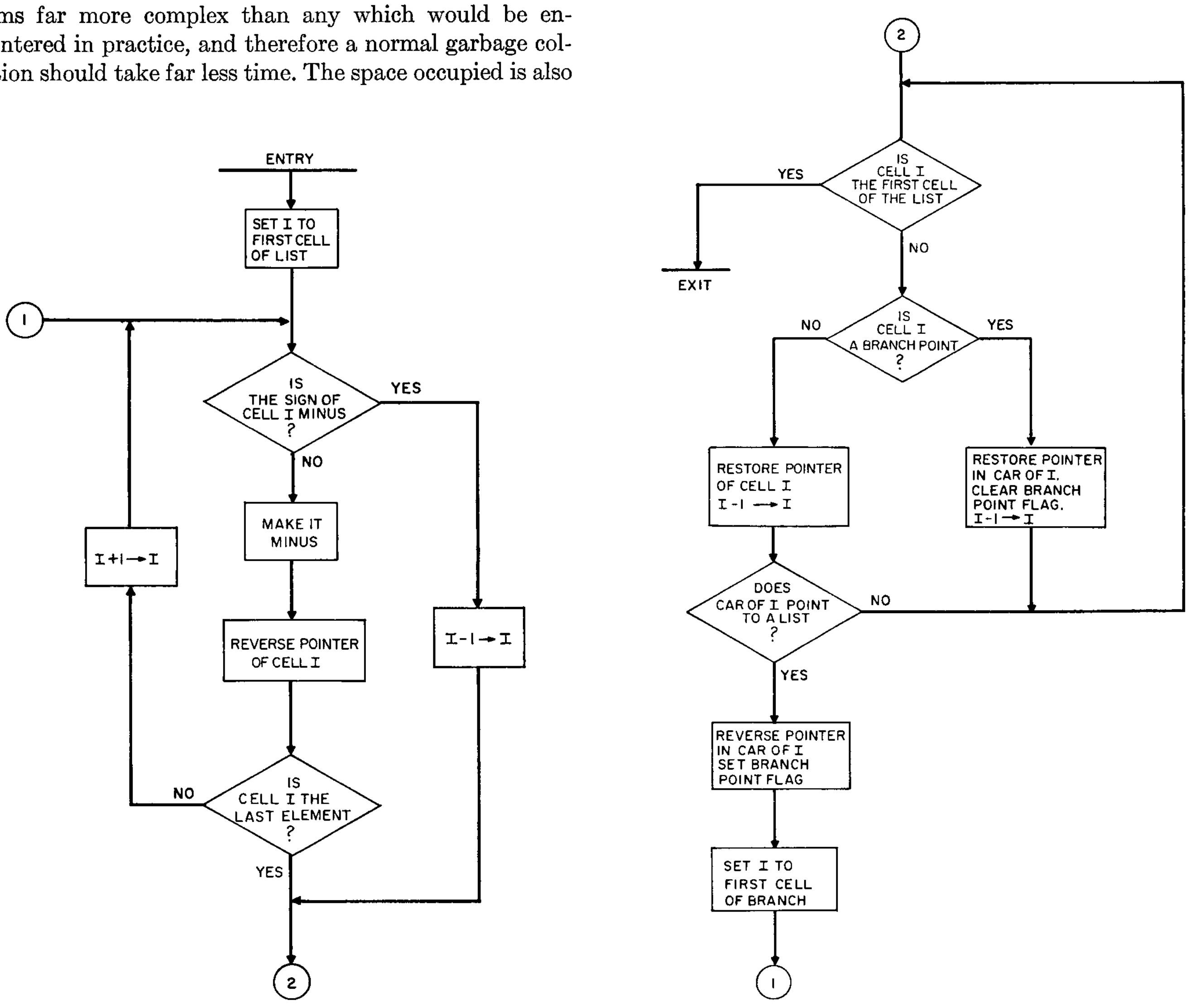

ENTRY
SET I TO FIRST CELL OF LIST
1
IS THE SIGN OF CELL I MINUS ?
YES
NO
MAKE IT MINUS
I+1→I
REVERSE POINTER OF CELL I
I-1→I
IS CELL I THE LAST ELEMENT ?
NO
YES
2
2
IS CELL I THE FIRST CELL OF THE LIST
YES
NO
EXIT
IS CELL I A BRANCH POINT ?
NO
YES
RESTORE POINTER OF CELL I I-1→I
RESTORE POINTER IN CAR OF I. CLEAR BRANCH POINT FLAG. I-1→I
DOES CAR OF I POINT TO A LIST ?
NO
YES
REVERSE POINTER IN CAR OF I SET BRANCH POINT FLAG
SET I TO FIRST CELL OF BRANCH
1

FIG. 3. Trace algorithm for a one-way list. Forward scan; reverse scan

On the basis of these results, it seems that the best garbage collection procedure is to use the last routine with as much temporary storage as possible, and when this storage is full to trace the remainder of the current branch using the algorithm given in Section 4. In this way one is able to realize the most efficient collection for the amount of temporary memory available, and yet there is no possibility of failure for any structure. If, however, no temporary storage area can be excluded from the free list or it is inconvenient to code two garbage collectors, then the algorithm of the previous section offers a reasonably efficient fail-safe solution.

5. Symbolic Garbage Collector

The transfer of a programming system from one machine to another can be more easily achieved if the compiler and its utility routines are written in a higher level (nonmachine) language [7, 12]. For this reason it is desirable to program the garbage collection routine for a list processing language in the language itself. The routine, if it is to work in all cases, must use a fixed amount of temporary storage. A recursive routine does not satisfy this condition because it requires a pushdown stack of indeterminate length on which to store return addresses. We have seen that the WISP garbage collection is divided into two phases: that in which the list structure is traced and marked, and that in which the new free list is formed from the unmarked registers. The second phase is severely machine-dependent, but the first can easily be written in a list language. The trace algorithm presented in Section 3 requires the addition of several elementary operations to those normally found in WISP. These were:

(1) set an element minus,
(2) set a branch point flag,
(3) delete a branch point flag,
(4) test the sign,
(5) test the branch point flag, and
(6) sequence from one list to the next.

Once these functions have been defined in machine language, the WISP compiler can translate the entire routine (which contains 35 WISP statements).

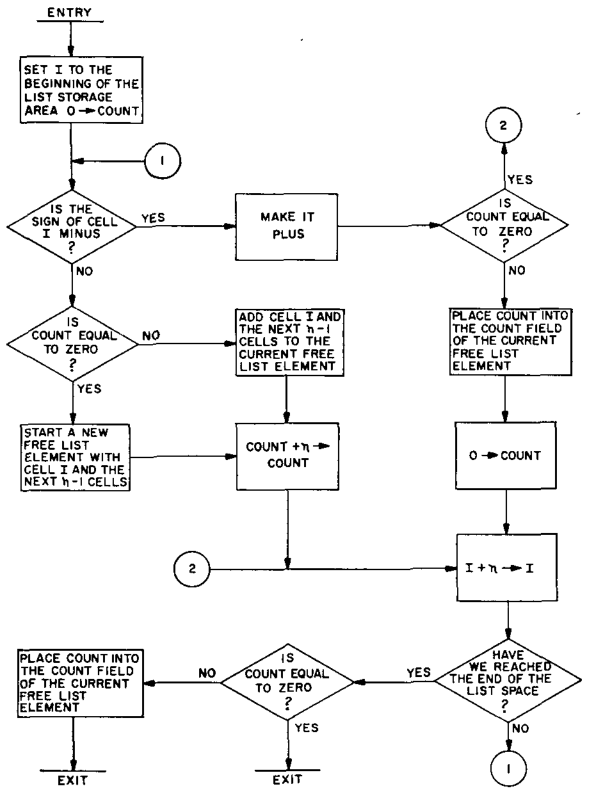
6. Garbage Collection in a Variable Item List

Various list processing systems have been proposed in which each list element consists of a number of consecutive registers [5, 6, 9, 10]. That is, if each element is to consist of n registers and the element is stored at memory location M, then the element is composed of the n consecutive registers $M, M+1, \ldots, M+n-1$. In some systems n is fixed for all list elements ($n = 2$ is a common choice), while in others n may vary from one list element to the next.

If $n, n > 1$, is fixed, the problem of garbage collection is similar to the case considered above where $n = 1$. The scan of Figure 3 is used to mark the first register of each list element minus. Whenever a negative register is found in the sequential scan of the store that follows, it and the next $n-1$ sequential registers are not returned to the free list. If a positive register is found, it and the next $n-1$ are returned. To see this, consider first the way in which the free list is originally formed. If all of the unused storage constitutes a consecutive block [11], then the free list can be constructed so that a new free list element occurs at every nth memory address from the start of the block. Thus it suffices to look at the every nth location in the free storage space; a positive register and the succeeding $n-1$ registers can be returned to the free list.

Consider next, garbage collection when n can vary from one list element to the next. In this case, the HEAD of the first register does not ordinarily contain data, but rather contains the number n of registers that make up the element. It is necessary then to distinguish between the following items that may occupy the HEAD: (1) the number n, as above, of consecutive registers, for $n > 1$, and (2) the data item itself (which may be a pointer to a sublist) if $n = 1$.

This information can be stored in one bit of the prefix field of a 7094 word, leaving one bit for the garbage collector's branch point flag. The scan shown in Figure 3 can again be employed to mark the first registers of those elements attached to lists that have not been discarded. Next, instead of merely returning all non-negative registers to the free list, the sequential scan of storage, which is flowcharted in Figure 4, must be used. This scan does not return a negative register to the free list—nor any of the next $n-1$ consecutive storage registers that make up the rest of the list element. Furthermore, this scan collects the largest block of consecutive discarded registers and forms them into a single free list element.



ENTRY
SET I TO THE BEGINNING OF THE LIST STORAGE AREA 0 → COUNT
1
IS THE SIGN OF CELL I MINUS ?
YES
NO
MAKE IT PLUS
2
YES
IS COUNT EQUAL TO ZERO ?
NO
IS COUNT EQUAL TO ZERO ?
NO
YES
ADD CELL I AND THE NEXT n-1 CELLS TO THE CURRENT FREE LIST ELEMENT
PLACE COUNT INTO THE COUNT FIELD OF THE CURRENT FREE LIST ELEMENT
START A NEW FREE LIST ELEMENT WITH CELL I AND THE NEXT n-1 CELLS
COUNT +n → COUNT
0 → COUNT
2
I+n → I
PLACE COUNT INTO THE COUNT FIELD OF THE CURRENT FREE LIST ELEMENT
NO
IS COUNT EQUAL TO ZERO ?
YES
HAVE WE REACHED THE END OF THE LIST SPACE ?
YES
NO
EXIT
EXIT
1

FIG. 4. Routine to reclaim variable-length items

7. Garbage Collection for Other List Structures

As mentioned above, some list languages permit the HEAD of a list element to contain the address of a full word of data. Since a data word may be a negative number, this presents difficulties during garbage collection. To avoid this difficulty, variable length list elements should be used to achieve full-word storage; this case then reduces to that of Section 6.

In a threaded list system [10], sublists cannot be shared, and garbage collection is particularly simple since the last element always points to the head of a list or sublist. During the forward scan it is not necessary to reverse the pointers, nor is a reverse scan needed since both only serve to effect a return to the head of a (sub)list. A flowchart of the first part of a garbage collection procedure for this type of system is given in Figure 5.

The routine can also be employed, with a slight modification, to collect garbage in a two-way SLIP list [5, 6]. In such a list the first list element, called a Header, has two pointers. One points to the next list element while the second points to the last element. The last element points back to the Header. Garbage collection can be achieved by modifying Figure 5 in the following manner.

(1) When a list is first encountered, it is checked to see if it has already been traversed.

(2) When a branch is descended, the Header address field which points to the last element is changed to point back to the branch element. (This enables a return to be made to the main list).

(3) In traversing the list, the address of the previous list element marked is saved. When the pointer from the last element back to the Header has been followed, this saved address is used to restore the above Header field while the branch element address stored in that field is used to resume scanning and marking the list on which the branch element appeared.

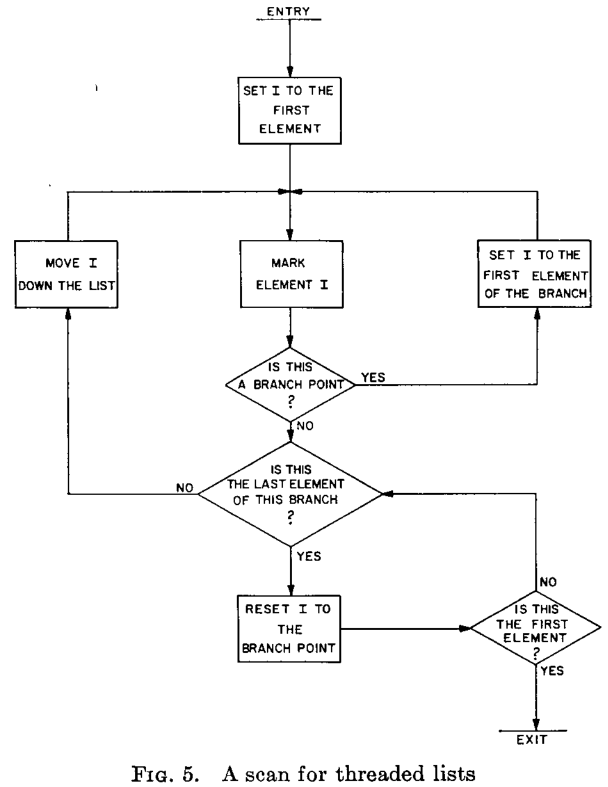

ENTRY
SET I TO THE FIRST ELEMENT
MOVE I DOWN THE LIST
MARK ELEMENT I
SET I TO THE FIRST ELEMENT OF THE BRANCH
IS THIS A BRANCH POINT ?
YES
NO
IS THIS THE LAST ELEMENT OF THIS BRANCH ?
NO
YES
RESET I TO THE BRANCH POINT
IS THIS THE FIRST ELEMENT ?
NO
YES
EXIT

FIG. 5. A scan for threaded lists

8. Conclusion

In this paper, various procedures for garbage collection have been presented. These procedures are useful for the list processing systems and structures previously reported in the literature. An algorithm which will trace *any* one-way list has been presented, and its efficiency discussed. In view of this routine's storage requirements and speed it seems that the method of garbage collection is far more efficient than the use of reference counters in one-way lists. Garbage collection in other types of list structures appears to be implemented by even simpler procedures.

By a slight extension of the elementary functions available to a list processing language, the garbage collection procedure may be described within the language in a nonrecursive way, thus facilitating its transfer from one machine to another. On the basis of these results, the use of garbage collection for reclaiming registers should always be considered when implementing a list processing system.

Acknowledgment. The authors gratefully acknowledge the cooperation of Professor M. V. Wilkes and his colleagues at the Cambridge University Mathematical Laboratory in transferring the WISP system to the IBM 7094. This project was made possible by an allotment of time from the Columbia University Computer Center.

RECEIVED FEBRUARY, 1965; REVISED MAY, 1967

REFERENCES

1. NEWELL, A. (ED.) *Information Processing Language—V Manual*, 2nd ed. Prentice Hall, Englewood N. J., 1964.
2. MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, part I. *Comm. ACM 3* (April, 1960), 184.
3. GERLERNTER, H., ET AL. A FORTRAN-compiled list-processing language. *J. ACM 7* (April 1960), 87.
4. COLLINS, G. E. A method for overlapping and erasure of lists. *Comm. ACM 3* (Dec. 1960), 655.
5. WEIZENBAUM, J. Knotted list structures. *Comm. ACM 5* (Mar. 1962), 161.
6. WEIZENBAUM, J. Symmetric list processor. *Comm. ACM 6* (Sept. 1963), 524.
7. WILKES, M. V. An experiment with a self-compiling compiler for a simple list-processing language. *Annual Review in Automatic Programming, Vol. 4.* Pergamon Press, N.Y. 1964, pp. 1–48.
8. WILKES, M. V. Lists and why they are useful. Proc. ACM 19th Nat. Conf., August 1964, ACM Publ. P-64, F1-1.
9. COMFORT, W. T. Multiword list items. *Comm. ACM 7* (June 1964), 357.
10. EVANS, A., PERLIS, A. J., AND VAN ZOEREN, H. Use of threaded lists in constructing a combined ALGOL and machine-like assembly processor. *Comm. ACM 4* (Jan. 1961), 36.
11. WAITE, W., AND SCHORR, H. A note on the formation of a free list. *Comm. ACM 8* (Aug. 1964), 478.
12. HALSTEAD, M. H. *Machine-Independent Computer Programming.* Spartan Books, Washington D. C., 1962.
13. MCBETH, J. M. On the reference counter method. *Comm. ACM 6* (Sept. 1963), 575.