Programming Techniques

ROBERT MORRIS, Editor

A LISP Garbage-Collector for Virtual-Memory Computer Systems

ROBERT R. FENICHEL AND JEROME C. YOCHELSON
Massachusetts Institute of Technology, Cambridge, Massachusetts

In this paper a garbage-collection algorithm for list-processing systems which operate within very large virtual memories is described. The object of the algorithm is more the compaction of active storage than the discovery of free storage. Because free storage is never really exhausted, the decision to garbage collect is not easily made; therefore, various criteria of this decision are discussed.

KEY WORDS AND PHRASES, LISP: garbage-collector, virtual memory, list-processing, storage-allocation
CR CATEGORIES: 4.19, 4.49

General Remarks

Users of list processing are familiar with garbage-collectors of the sort first described by McCarthy [1]. Systems using collectors of this sort run freely until space is nearly exhausted. Then, all execution stops while a *marking* routine marks every free-storage cell which is reachable by program. Finally, a *gathering* routine scans the free-storage area, collecting the unmarked cells onto a free-storage list, and unmarking the marked cells.

Garbage-collection has always been necessary because the computer's supply of addressable space has always been much less than the total space used during execution of a list-processing program. Garbage-collection makes it possible to reuse the system's limited supply of addressable space.

This work was supported in part by Project MAC, an MIT research program sponsored by the Advanced Research Project Agency, US Department of Defense, under Office of Naval Research Contract No. Nonr-4102 (01). Reproduction in whole or in part is permitted for any purpose of the United States Government.

With the coming of virtual-memory systems [2, 3], the problem of limited addressable space is hardly present. In MULTICS, for example, a LISP system might be made to operate with a potential free-storage list of *billions* of LISP cells.

Such a system may run almost endlessly with no need for garbage-collection. As operation proceeds, however, performance degrades. This is because the active-list-storage becomes spread over a larger and larger region of virtual storage, and it becomes increasingly likely that a given reference to this virtual memory will require a reference to secondary storage.

Bobrow and Murphy [4] faced a substantially similar problem, but the virtual memory of their system was not yet so large as to be effectively infinite. Many of their strategies for pointer enrichment and for data segmentation are appropriate to an infinite memory system, but their garbage-collector is not.

What is needed is a collector whose task is not so much the discovery of free storage as the compaction of active storage. It is especially clear that a routine will not do if its gathering phase must scan all potential storage.

The procedure is shown in detail below. Briefly, it operates by dividing the potential storage space into two *semispaces.* Only one semispace is used for free storage (only one semispace is *current*) at a time. The collection procedure simply copies the active storage into a compact portion of the noncurrent semispace. Then, this second semispace is made current.

Initiating Collection

There is no simple condition under which to initiate a garbage-collection of the sort described here. By hypothesis, one can not wait until storage is nearly exhausted. The primary reason to initiate collection must be a low observed ratio of processor cycles to elapsed time.

Because these collections are never necessary, only increasingly desirable, other factors may influence the decision to initiate collection. In on-line systems, for example, it might be reasonable to favor collection at times when the system would otherwise be blocked, waiting for input. In any type of system, collection should be favored when the pushdown list is short, so that the actual work of collection is likely to be minimal. Any strategy of collection should best be tuned to the statistical behavior of the combined environment presented by users and host system.

The Algorithm

This garbage-collector, because it copies lists structures, shares the chief problem of list-storage copying programs. That is, that if the most obvious strategy[1] is used, then copies of certain lists will be nonisomorphic to the originals. For example, an n-word structure like the one shown in Figure 1 will be expanded to a $(2^n - 1)$-word tree.[2]

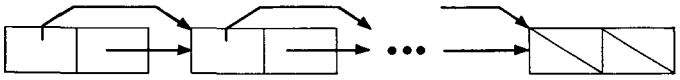

FIG. 1

Because the active storage will be thrown away as soon as the garbage-collector has copied it, the garbage-collector, unlike most copying programs, is privileged to modify the structure being copied. The collector therefore replaces any copied structure by a pointer to the copy. Subsequent discovery of this altered original will prevent recopying.

The top-level structure of the garbage-collector must be a loop which successively examines each independent, program accessible pointer into list storage. At one extreme, there may be only one such pointer—that to the top of the pushdown list. On the other hand, there may be various other pointers (e.g. to an object list, from special temporary storage areas, to list-literals used by compiled routines). The top-level loop is only hinted at below. What is shown in detail is the subroutine COLLECT.

COLLECT takes as its argument a pointer to a list structure in the current semispace, and it returns as its value a pointer to the copy in the other semispace. While the version of COLLECT shown here is recursive, a (complex) nonrecursive COLLECT could be constructed with the algorithm credited to Deutsch, Schorr, and Waite by Knuth [6].

Several functions are taken as primitives:

atom[a], car[a], cdr[a], rplaca[a;b],rplacd[a;b], cons[b; c]	Usual LISP functions. The pointer *a* is taken to refer to the last semispace set by *flipsemispace*.
flipsemispace[ ]	Flips interpretation of pointers to the other semispace.
flipconsspace[ ]	Alters operation of *cons* so that *list cells* are generated in the other semispace. The *n*th list cell taken there will simply be the *n*th cell in the semispace. There is, in other words, no explicit free-storage list.
collectatom[a]	Like COLLECT, but for atoms. Probably uses COLLECT internally on property lists, depending upon implementation of atoms (see [4]).
nrplaca[a;b],nrplacd[a;b]	Like *rplaca* and *rplacd*, but the pointer *a* is taken to refer to the other semispace.

Here is the garbage-collector:

```
garbagecollector[ ] = prog[[p];
  flipconsspace[ ];
  (for each root pointer p) p := collect[p];
  flipsemispace[ ]]
collect[p] = [
  atom[p] → collectatom[p];
  eq[car[p]; ALREADYCOPIED] → cdr[p];
  T → prog[[a;d;q];
    a := car[p];
    d := cdr[p];
    q := cons[NIL; NIL];
    rplaca[p; ALREADYCOPIED];
    rplacd[p;q];
    nrplacd[q; collect[d]];
    nrplaca[q; collect[a]];
    return[q]]]
```

RECEIVED DECEMBER, 1968; REVISED MAY, 1969

REFERENCES

1. MCCARTHY, JOHN. Recursive functions of symbolic expressions and their computation by Machine-I. *Comm. ACM 3*, 4 (Apr. 1960), 184–195.
2. CORBATÓ, F. J., AND VYSSOTSKY, V. A. Introduction and overview of the MULTICS system. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Pt. 1, Spartan Books, New York, pp. 185–196.
3. COMFORT, WEBB T. A computing system design for user service. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Pt. 2, Thompson Book Co., Washington, D.C., pp. 619–626.
4. BOBROW, DANIEL G., AND MURPHY, DANIEL L. Structure of a LISP system using two-level storage. *Comm. ACM 10*, 3 (Mar. 1967), 155–159.
5. EDWARDS, DANIEL J. Secondary storage in LISP. Paper, First Internat. LISP Conf., 1964.
6. KNUTH, DONALD E. *The Art of Computer Programming, I.* Addison-Wesley, Reading, Mass., 1968, p. 417.
7. YOCHELSON, JEROME C. Multics LISP. Unpublished, MIT, Cambridge, Mass., 1967.

[1] That is copy[x] =[atom[x] → x; T → cons[copy[car[x]]; copy [cdr[x]]]].

[2] Structures of this sort have been called BLAM lists by Edwards [5], presumably onomatopoeically from the problem which they present to copying-programs.

CORRIGENDUM (COMMUNICATIONS SYSTEMS): For the interest of those who file or republish abstracts and others who may have been puzzled, attention is directed to an unfortunate error in the paper "A Modular Computer Sharing System" by Herbert B. Baskin, Elsa B. Horowitz, Robert D. Tennison, and Larry E. Rittenhouse [*ACM Comm. 12*, 10 (Oct. 1969), 555–559]. The phrase in the fifth line of the abstract should read "a memory/processor pair."