

Programming Techniques

R. M. McCLURE, Editor

Recovery of Reentrant List Structures in SLIP

JOSEPH WEIZENBAUM

Massachusetts Institute of Technology, Cambridge, Mass.

One consequence of the reference-count-based space-recovery system employed by SLIP is that reentrant list structures are not recovered even when explicitly erased. LISP-like garbage-collection schemes are free of this impediment. They, however, depend on being able to find and mark nodes that are reachable from program variables. By tracing all descendants of such nodes and marking them as well, all cells not reachable from program variables may then be identified and collected. The list-creating function *LIST* of SLIP may be amended to mark those lists for which the programmer wishes to assume responsibility. Given this modification, a LISP-like garbage collector that recovers abandoned reentrant list structures may then be appended to the SLIP system.

KEY WORDS AND PHRASES: list processing, SLIP, garbage-collection
CR CATEGORIES: 4.22, 4.40

The mechanism that administers the fetching of cells from and their restoration to the list of available space in SLIP suffers from the defect that self-referential list structures cannot be automatically garbage collected. It will be recalled that in SLIP every list has a list header which contains, among other data, a reference counter. This counter tallies the number of times a list appears as a sublist of some list. Lists can be created with a reference count of 1 or 0, depending on whether the programmer wishes to take responsibility for erasing the newly-created list or has created it in order to subsequently make it a sublist of some list. The function

LIST (*K*) (1)

creates an empty list and assigns to the variable *K* the name, i.e. pointer to the header, of the newly-created empty list. *K* is then called an *alias* of that list. The programmer is said to "know" lists created with simultaneous name assignment to an alias. Headers of such lists are given an initial reference count of 1. The call

LIST (9) (2)

creates an empty list and leaves its internal name as its

value. The header of such a list has an initial reference count of 0.

The function

ERASEL (*K*) (3)

should be applied only to aliases of known lists. Its function is to decrement the reference count of the list communicated to it and, if the count reaches 0, to erase the list. The erasure itself amounts to restoring the body of the list to available space in bulk—an easy task, since the header contains pointers to both the top and bottom of the list—and finally returning the header cell itself to available space as a single cell. (The erasure of description lists of lists is also simple but will not be discussed here.)

The function *NUCELL* has the responsibility of fetching cells from the top of the available space list one at a time as required. It inspects each cell it is about to deliver. If that cell contains a list name, i.e. if it was most recently a sublist node of a list that has been emptied, then the list there named is subjected to the *ERASEL* procedure. If *NUCELL* finds that there are no more cells to be had, it reports that fact and terminates execution of the program.

An advantage of the SLIP list erasure scheme is that it operates incrementally in response to actual space requirements. Unless the entire list of available space has been gone through at least once, very little time is taken for space recovery. Also, lists are collected in bulk, so to speak, as opposed to cell by cell. The time required to collect a list is thus independent of its length.

To see the difficulty arising from self-referential list structures, consider the following example:

NEWTOP (*A*, *LIST* (*A*))
ERASEL (*A*) (4)

When *A* is created, its header is given a reference count of 1 because it is being created as a known list. When next *A* becomes a sublist of some list, in this case itself, its reference count is increased by 1. The count is 2 by the time the *ERASEL* function is exercised. That function reduced the count to 1 and, seeing that it is not 0, does nothing further, i.e. does not actually erase the list. The space taken by the list *A* is therefore lost to the system. (The reader may wish to convince himself that calling *ERASEL* (*A*) again would only result in chaos! Besides, the programmer shouldn't *have* to know that he got himself into such a condition.) More complex self-referential loops are, for present purposes, equivalent to the above.

The solution to the problem here illustrated depends on the system knowing which lists are accessible from program variables, or are sublists of such lists, and which have survived entirely because they are self-referential structures. The first step is to amend the list creation function *LIST* such that when it creates a *known* list it marks the header of that list as "reachable." (This mark could be the sign bit of the first word of the header pair.) The function *ERASEL* is then modified such that, unless called by *NUCELL*, it removes the reachable mark from the header of the list it is called upon to erase, as well as counting down its reference counter and erasing if necessary. Finally, *NUCELL* is modified such that it doesn't give up when the list of available space is exhausted but calls the Boolean function *GARBAGECOLLECT*. If this function delivers **true** as its value, then it recovered some space and *NUCELL* can go on. Otherwise, *NUCELL* must report failure.

The function *GARBAGECOLLECT* (*freestart*, *length*) is shown below written in ALGOL. *freestart* is the machine address of the first cell of the vector that was made into the list of available space (by *INITAS*), and *length* is the length of that vector. Automatic ALGOL recursion is assumed. The general idea is that one pass identifies and marks as "seen" all headers that are reachable either directly from program variables, i.e. those that are "reachable," or by tracing such reachable lists. In a second pass it empties all lists that have not been "seen" and whose reference counters are greater than 1. The assumption is that such lists have, in fact, been abandoned but retain a nonzero reference count because they are reentrant. The headers of such lists need not be restored by *GARBAGECOLLECT*, since their machine addresses appear as list names somewhere on the structures being emptied. Such headers will therefore subsequently be collected by the operation of *NUCELL*.

The Garbage Collection Algorithm

```
boolean procedure garbagecollect(freestart, length);
integer freestart, length;
begin boolean signal; integer i, pointer;
comment This procedure uses the following SLIP library functions
(in the remainder of this commentary "pointer" is assumed to be a cell containing the machine address of the first word of a SLIP cell pair):
(1) rmarked(pointer) is a Boolean function. Its value is true if and only if the cell pointed to by pointer is a header marked as "reachable."
(2) seen(pointer) is a Boolean function similar to rmarked. Its value is true if and only if the cell pointed to by pointer has been reached in the course of a tracing pass.
(3) unmark(pointer) is a procedure that removes the "seen" mark from the header to which pointer points.
(4) emptylist, leftlink, and header are the old SLIP functions MTLIST, LNK, and NAMTST, respectively, except that the function "header," unlike NAMTST, simply checks the ID of the cell pointed to. The function referencecounter(pointer) assumes pointer is pointing to a list header and retrieves its reference count. It is identical to the standard SLIP function LCNTR.
The parameter freestart communicated to this function is the machine address of the first cell of the vector that forms the list
```

of available space. *length* is the length of that vector. The value of this function is **true** if space has been recovered; otherwise, it is **false**;

```
procedure trace(list);
begin integer sequencer, flag, word, dlist;
comment This procedure marks the header of the list given it as an argument with the seen mark and then proceeds to trace through the entire list structure headed by that header. It recursively applies itself to every sublist (including description lists) so encountered;
comment This procedure uses the standard SLIP library functions
(1) seqdr(list), and
(2) seqr(sequencer, flag), as well as the new functions
(3) markseen(list), which assumes list to be an alias for a list and causes a seen mark to be put on its header;
(4) seen(list), described above; and
(5) dname(list), which is the standard SLIP function that retrieves the name of a description list of a list. Its value is 0 if the relevant list has no description list;
markseen(list);
dlist := dname(list);
if dlist ≠ 0 ∧ ¬ seen(dlist) then trace(dlist);
sequencer := seqdr(list);
read: word := seqr(sequencer, flag);
comment The value of flag will be less than 0 if the word sequenced to is an atom, equal to 0 if it is a list name, i.e. the name of a sublist of list, and greater than 0 if the end of the list being sequenced has been reached;
if flag = 0 ∧ ¬ seen(word) then trace(word);
if flag ≤ 0 then go to read;
end
signal := false;
for i := 0 step 2 until i = length do
comment In this first pass through the freelist vector, all headers that have been rmarked, i.e. that are reachable from program variables, as well as all list headers that are reachable through such headers, are marked as seen;
begin
pointer := freestart + i;
if rmarked(pointer) ∧ ¬ seen(pointer) then trace(pointer)
end
for i := 0 step 2 until i = length do
comment In this second pass through the freelist vector, all headers that are marked seen have that mark removed and all other headers have the lists they head emptied;
begin
pointer := freestart + i;
if seen(pointer) then unmark(pointer) else
begin
if header(pointer) ∧ referencecounter(pointer) ≠ 0 then
begin
emptylist(pointer);
signal := true
end
end
end
garbagecollect := signal
end
```

Discussion

The *TRACE* routine shown here depends on recursion in the host language. A SLIP list-tracing routine requiring no pushdown store can, however, be written by using the space allocated to the *BOTTOM* pointers in list headers for temporary storage during the scanning operation.

The advantage of the scheme here presented is that it

preserves the incremental and bulk garbage-collection mechanism of the original SLIP system. *GARBAGECOLLECT* is called only in an emergency, and then it does recover self-referential list structures. The scheme can be installed in a SLIP system without disturbing the compiler of the language in which SLIP is embedded.

Knuth states [1, p. 420] that "... some systems employ both the reference counter and garbage collection schemes, besides allowing the programmer to explicitly erase nodes." The mechanism presented here is precisely such a scheme for SLIP. Knuth then goes on to say, "This idea is of debatable merit, in view of the inefficiency of garbage collection when memory is nearly packed, and since so few programs come close to filling memory without exceeding the limits shortly thereafter." Clearly, what has to be balanced here is the storage required for the garbage-collection algorithm itself against the number of SLIP cells likely

to be recovered in a given invocation. If, for example, the space required for the algorithm is n words, then the recovery of $n/2$ SLIP cells is obviously not worthwhile. But if, at least in SLIP, reentrant list structures continue to be lost to the system, then such losses become cumulative and will eventually cause termination of a program that might otherwise have run to completion. This writer is the author of a SLIP program that would prove entirely infeasible were it not that reentrant list structures are made recoverable.

RECEIVED APRIL, 1968; REVISED JANUARY, 1969

REFERENCES

1. KNUTH, DONALD E. *Fundamental Algorithms*. Addison-Wesley, Reading, Mass., 1968.
2. MCCARTHY, JOHN. Recursive functions of symbolic expressions and their computation by machine, part 1. *Comm. ACM* 3, 4 (Apr. 1960), 184-194.
3. WEIZENBAUM, J. Symmetric list processor. *Comm. ACM* 6, 9 (Sept. 1963), 524-544.

Randell—cont'd from page 369

ment as part of the formation of the effective address. However no carry is permitted into higher bit positions of the effective address. Thus for example a three quarter page block could be allocated within a page frame, starting in the last quarter of the frame.

Thus the basic difference between this technique and the method incorporated in the partitioned segmenting scheme is that, at the expense of a somewhat more complicated method of allocating and relocating storage, the possibility of a lengthy carry propagation during the formation of an effective address is avoided in the CDC 3300.

Conclusions

The above study has attempted to differentiate between the effects on system performance caused by the way a program and its data are structured and the actions of the dynamic storage allocation system which is used during program execution. Some evidence, obtained from simulation experiments, has been given of the undesirability of rounding up segment sizes to an integral number of page sizes in an attempt to remove storage fragmentation. Finally, descriptions are given of segment allocation and addressing mechanisms for a system of partitioned segmenting, whose investigation was prompted by this experimental evidence.

Acknowledgments. The author is grateful for many useful discussions with L. A. Belady, C. J. Kuehner, M. Lehman, R. W. O'Neill, and F. W. Zurcher.

RECEIVED JANUARY, 1969

REFERENCES

1. BELADY, L. A. A study of replacement algorithms for a virtual-storage computer. *IBM Systems J.* 5, 2 (1966), 78-101.
2. COMEAU, L. W. A study of the effect of user program optimization in a paging system. ACM Symposium on Operating System Principles, Oct. 1-4, 1967, Gatlinburg, Tenn.
3. COMFORT, W. T. A computing system design for user service. Proc. AFIPS 1965 Fall Joint Comput. Conf. Vol. 27, Pt. 1, Spartan Books, New York, pp. 619-628.
4. CORBATO, 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.
5. FALKOFF, A. D. AND IVERSON, K. E. The APL/360 terminal system. ACM Symposium on Interactive Systems for Experimental Applied Mathematics, Washington, D.C., Aug. 26-28, 1967.
6. FINE, G. H., JACKSON, C. W. AND McISAAC, P. V. Dynamic program behavior under paging. Proc. ACM 21st Nat. Conf. 1966, Thompson Book Co., Washington, D.C., pp. 223-228.
7. KNUTH, D. E. *The Art of Computer Programming, Vol. 1—Fundamental Algorithms*. Addison Wesley, Reading, Mass., 1968, p. 448.
8. McCULLOUGH, J. D., SPEIERMAN, K. H., AND ZURCHER, F. W. A design for a multiple user multiprocessing system. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Pt. 1, Spartan Books, New York, pp. 611-617.
9. McKEEMAN, W. M. Language directed computer design. Proc. AFIPS 1967 Fall Joint Comput. Conf., Vol. 31, Thompson Book Co., Washington, D.C., pp. 413-417.
10. RANDELL, B. AND KUEHNER, C. J. Dynamic storage allocation systems. *Comm. ACM* 11, 5 (May 1968), 297-306.
11. TOTSCHKE, R. A. An empirical investigation into the behavior of the SDC time-sharing system. Rep. SP 2191, System Development Corp., Santa Monica, Cal., 1965, AD 622 003.
12. 3300 Computer System Reference Manual. Pub. No. 60157000, Control Data Corp., St. Paul, Minn., 1966.
13. The Descriptor—a definition of the B5000 information processing system. Burroughs Corp., Detroit, Mich., 1961.