Managing Reentrant Structures Using Reference Counts

DANIEL G. BOBROW
Xerox Palo Alto Research Center

Automatic storage management requires that one identify storage unreachable by a user's program and return it to free status. One technique maintains a count of the references from user's programs to each cell, since a count of zero implies the storage is unreachable. Reentrant structures are self-referencing; hence no cell in them will have a count of zero, even though the entire structure is unreachable. A modification of standard reference counting can be used to manaage the deallocation of a large class of frequently used reentrant structures, including two-way and circularly linked lists.

All the cells of a potentially reentrant structure are considered as part of a single group for deallocation purposes. Information associated with each cell specifies its group membership. Internal references (pointers from one cell of the group to another) are not reference counted. External references to any cell of this group are counted as references to the group as a whole. When the external reference count goes to zero, all the cells of the group can be deallocated. This paper describes several ways of specifying group membership, properties of each implementation, and properties of mutable and immutable group membership.

Key Words and Phrases: storage management, garbage collection, reference counting
CR Categories: 4.19

INTRODUCTION

An automatic storage management scheme based on reference counting keeps track of the number of references to any cell and deallocates that cell when the number of references to it drops to zero. This does not deallocate all unreachable storage, however. If a cell A references B which (more or less indirectly) references A, then the *reentrant structure* containing A and B maintains the reference count of each of its cells at one without there being any external references to the structure as a whole. Thus the storage manager will never deallocate any part of this structure, since the reference count of each cell never drops to zero. Trace-and-mark garbage collectors do not have this problem because they compute the accessibility of each cell during the trace, rather than depending on a reference count to provide that information. However, tracing requires time proportional to the amount of storage in use to manage deallocation, rather than to the amount

Author's address: Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Rd., Palo Alto, CA 94303.

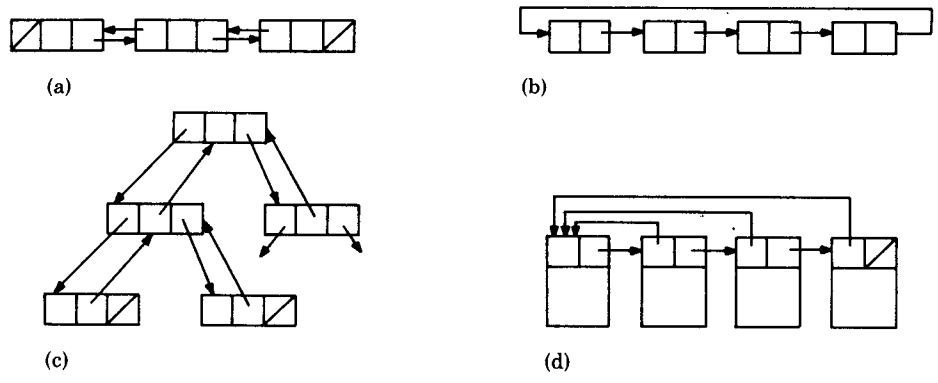

Fig. 1. Some reentrant structures which can be managed. (a) Two-way list. (b) Two-way tree. (c) Circular list. (d) Head-linked list.

of storage to be deallocated. Knuth [4] gives a fuller discussion of the advantages of reference counting over trace-and-mark for storage management.

There are a number of special cases of reentrant structures that are frequently used in programming; some examples are shown in Figure 1. In current reference count storage management systems, such structures could not be automatically deallocated without the user explicitly breaking the cyclic structure. This paper describes how to modify a reference count collector to manage a large class of reentrant structures, including those in Figure 1.

The basic idea is that the set of cells which makes up the reentrant structure is considered as a single group for deallocation purposes. We divide references to any cell in the group into two classes; *internal* references, which are pointers from one element of the group to another in the same group, and *external* references. Internal references are not reference counted. External references to any cell of this group are counted as references to the group as a whole. When the group reference count goes to zero, the entire group can be deallocated.

This scheme requires that programs which store pointers (1) be able to distinguish whether they are creating an internal or an external reference; (2) for an external reference, know where to store the reference count for the group. In each implementation described below, the programmer must declare in advance some information concerning the use of pointers in cells of the structure. None of the implementations take care of cycles that go outside the group and then back in.

ZONE REFERENCE COUNTING

In zone allocation, storage is allocated from a single zone for some purpose. When that purpose has been fulfilled, the programmer deallocates the entire storage zone at once. Our scheme frees the programmer from knowing exactly when the zone is no longer needed. The zone itself is reference counted. Pointers from cells outside the zone to cells within the zone are counted as references to the zone as a whole. The test for pointers within a zone is an address range test or quantum map test. When the zone reference count becomes zero, all cells in the zone are released.

```
store p into f of c
    * p is a pointer; c is a cell, and f is the field of that cell; note
    * that this routine is safe (does not reclaim good stuff) even if it is interrupted
temp ← fetch f of c;
gp ← groupNumber(p);
gc ← groupNumber(c);
gt ← groupNumber(temp);
if gp ≠ gc then incrCount(p);
c[f] ← p;                                        * really store p into field f
if gt = gc then begin
  n ← decrCount(temp);          * decrement the ref count and return new value
  if n = 0 then reclaim(t); end
end store
```

Fig. 2. Sketch of the code for storing a pointer into a cell.

The obvious disadvantage of this scheme is potential wasted space if the size of the group is much smaller than the size of the zone. This can happen because, by definition, storage in the zone can only be used for cells of this one structure. However, zone reference counting has very low overhead and a very simple deallocation mechanism.

IMMUTABLE GROUP LABELING

Cells for many different structures can be allocated out of a single zone. However, this requires that each cell contain its *group number*. In this section we insist this group number be immutable after creation. A *group reference count table*, indexed by group number, stores information common to the group. When a pointer to p is inserted or deleted from a cell c, no reference count operations are done if p is in the same group as c. If p is in a different group, reference counting is done. When the group reference count goes to zero, all members of the group can be reclaimed. Figure 2 shows a sketch of the code for storing a pointer into such a cell.

How can reclamation be done? Some members of the group may be completely disconnected. The allocation zone can be swept to pick up members of released groups. This can be done either on release of any group, or at longer periods to minimize sweep overhead. Alternatively, at the cost of an extra pointer in each cell, all cells of the group can be kept linked through a field of the cell, with a pointer to the head kept in the group reference count table. Then the list can be added to the free list in a single clump.

This scheme allows one to allocate in one zone, with good storage efficiency, many small reentrant structures, since the zone can be shared by a large number of instances. The zone can be extended at any time, as long as the storage manager knows where all pieces of the zone are. It does require a group reference table and space in the cell for the group number. Notice that these cells do not have to be all of the same type, only that their group identity must be found in a known way.

MUTABLE GROUP LABELING

If the programmer declares that certain pointers of a cell are to be internal pointers only, then one can allow change of the group number of a cell at run

time; I call this mutable group labeling. Doing this allows creation of a cell without specifying its group in advance. The group number is filled in the first time an internal pointer to it is created, or the first time an internal pointer is inserted in it to another cyclic cell.

For example, if the programmer declares type *TwoWayList*, with fields *Back* and *Forward* declared internal, then only when a two-way list cell is linked into a list is its group number assigned. Since external references to this cell must be kept even before it is linked in, we use a temporary group number which can be reused when the cell is assigned to a group. At the time of assignment, the external reference counts are transferred from the temporary group to the permanent group.

This notion of merging a newly created cell into a permanent group can be extended to allow the merger of two permanent groups. Suppose there exist two two-way lists A and B. If the user requests that the tail of A be linked to the head of B, there are two reasonable alternatives. The first is to consider this an error. The second is to make A and B belong to the same group.

Merging A and B can be implemented in several ways. All require that the sum of the two group reference counts be ascribed to one group (say A), and that all references to group B afterward be known as references to group A. One could find all cells of group B and change their group label to A (either by tracing the B structure or by sweeping the allocation zone). Alternatively, the group reference count entry for B could be marked as indirect with a pointer to A. This latter indirect merging technique is very similar to that for merging sets, for which a number of good algorithms have been developed and analyzed [3, 5].

Mutable group labeling allows separate structures to be merged. Once groups are merged, they cannot be easily broken apart again. To break a "child" group out from a "parent" group in a mutable labeling scheme, external reference counts must be kept for each individual cell. When the split occurs, it must be possible to find every member of the child subgroup, recompute this group's external reference count, and subtract it from the parent group count.

HEAD LABELED GROUPS

Instead of an index into a table, the group label can be made the address of some particular cell in an instance of a cyclic structure. The group reference counts are forwarded to this "head" cell. Since the group label now has semantic significance, it can be used by the programmer as a pointer within the structure, although the programmer cannot be allowed to change it.

As an example of the use of a head labeled group, consider the head-linked list in Figure 1d. The head link allows a program to find the full list from a pointer to any element in the list. By declaring this structure to be "head labeled" and the link pointer to be internal, the programmer automatically creates a back link field which cannot be changed. This allows the reentrant structure to be deallocated by a reference count storage manager. Note that no more storage is used in the new structure than in the old; the back link just does double duty.

If the reference count forwarding is done recursively, then the two-way tree shown in Figure 1b can be reinterpreted to use the backpointers as forwarding addresses, with a computation cost proportional to the depth of the tree.

RELATED WORK

SLIP [6] uses reference-counted storage control. Only the *head* of a SLIP list can be pointed to by another structure, and the head keeps its reference count. SLIP lists use uncounted internal references between elements of the two-way linked cells of a list. These internal elements can only be referred to by a special dynamic structure called a *reader*. The reader maintains a reference-counted pointer to the head and an uncounted pointer to the internal element.

Fenichel [1] describes general structures that can have uncounted private reentrant pointers, but also requires the system to keep a counted pointer to the head of a structure. Friedman and Wise [2] have done work which is closest in spirit to the present paper. They show how to use reference counting to manage the specialized reentrant structure used in functional bindings. They require the structure to be created all at once, never altered, and pointed to in a standardized way. The scheme described in this paper allows a "naked" pointer to any of the elements of the group. The group can be built incrementally, and the system need not maintain any explicit pointer to the *head* of the reentrant structure.

ACKNOWLEDGMENTS

This paper profited from my discussions with Mark Brown, Peter Deutsch, Martin Kay, Bill McKeeman, and Terry Winograd. Mark Brown pointed out the similarity of the problems of merging deallocation groups and merging sets.

REFERENCES

1. FENICHEL, R.R. List tracing in systems allowing multiple cell-types. *Commun. ACM 14*, 8 (Aug. 1971), 522–526.
2. FRIEDMAN, D.P., AND WISE, D. S. Reference counting can manage the circular environments of mutual recursion. *Inform. Process. Lett. 8*, 1 (1979).
3. HOPCRAFT, J.E., AND ULLMAN, J.D. Set merging algorithms. *SIAM J. Comput. 2* (1973).
4. KNUTH, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1973.
5. TARJAN, R.E. Efficiency of a good but not linear set merging algorithm. *J. ACM 22*, 2 (April, 1975), 215–225.
6. WEIZENBAUM, J. Symmetric list processor. *Commun. ACM 6*, 9 (Sept. 1963), 524–544.

Received November 1979; revised April 1980; accepted April 1980