Check for updates

Programming Techniques
G. Manacher, S.L. Graham Editors

An Efficient, Incremental, Automatic Garbage Collector

L. Peter Deutsch and Daniel G. Bobrow
Xerox Palo Alto Research Center

This paper describes a new way of solving the storage reclamation problem for a system such as Lisp that allocates storage automatically from a heap, and does not require the programmer to give any indication that particular items are no longer useful or accessible. A reference count scheme for reclaiming non-self-referential structures, and a linearizing, compacting, copying scheme to reorganize all storage at the users discretion are proposed. The algorithms are designed to work well in systems which use multiple levels of storage, and large virtual address space. They depend on the fact that most cells are referenced exactly once, and that reference counts need only be accurate when storage is about to be reclaimed. A transaction file stores changes to reference counts, and a multiple reference table stores the count for items which are referenced more than once.

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

Copyright © 1976, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Authors' address: Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

Introduction

Attention to details of allocation of storage, and explicit deletion of storage no longer necessary, is a burden on a programmer who is trying to focus attention on a problem to be solved. Languages such as Lisp and Snobol, which are designed to facilitate programming of complex symbolic processes, ease this burden by providing automatic storage allocation and reclamation. Such storage management facilities have been designed, in general, to work best in an environment in which all storage can be kept in rapid random access memory. In addition, schemes thus far implemented have a overhead while running which is proportional to the amount of storage in use.

In this paper, we suggest a combination of variations of the two standard methods which we claim provides a more elegant and efficient solution to the storage reclamation problem. Our scheme allows incremental collection of most unused storage, and works well in a system which uses a hierarchy of memory storage devices. Automatic reclamation of storage no longer in use is done by the following two techniques:

(1) *Garbage collection.* When storage fills up, the system traces starting from all data structures directly accessible to the programmer, marking these and all indirectly accessible structures as needing to be retained. Then all space not marked is reclaimed. The disadvantage of garbage collection is that the time it takes to collect unused space is proportional to the amount of space still in use, since that all must be traced. In addition, in an extended physical memory system, the garbage collector must access many pages on secondary store, an inherently slow process. Finally, garbage collection is a completely disruptive process; computation cannot proceed while it is in progress. Aside from the Bobrow proposal [1] for garbage collection while processing continues, which has never been implemented and may actually not work, garbage collection usually requires a consistent static state, with the stack frozen.

(2) *Reference counts.* Each allocated datum keeps track of how many other data reference it. (This includes pointers from variable values as well as from other allocated data.) When the count becomes zero, the datum can be reclaimed. This scheme does not detect self-referential structures which are not accessible from the outside; also, the reference count is often only a few bits wide, and once it reaches its maximum value it can never be decremented. These properties require a garbage collector in addition to the reference counting. Most inaccessible storage is picked up by the reference count going to zero, and thus garbage collections can be much less frequent and ensure maximum utilization of storage. Since the garbage collections are not needed often, the user can be given the choice (responsibility) for choosing a convenient moment to initiate one.

The chief attraction of the reference count scheme

is that it is transaction-oriented: that is, the overhead required for proper storage reclamation (the incrementing and decrementing of the counts) is proportional to the number of transactions which affect the accessibility of data. Only operations which change pointers to structures can affect accessibility. These include updating pointers in record structures, and changing the values of variables which contain pointer values. Since these operations are usually uniformly distributed in a computation, the overhead for storage maintenance is roughly proportional to the total computational effort. This is not true for garbage collection: the overhead for the marking phase is proportional to the total amount of data in the system. This tends to suggest reference counts, rather than tracing, as the basis of any incremental reclamation scheme.

A pure reference count scheme has two significant sources of overhead:

(1) Every allocated datum (list cell, record, etc.) must include space for the reference count. This space overhead may be quite significant in a system like Lisp where the allocation unit is a single word. Packing constraints often render it awkward to store the reference count within the datum itself, which engenders additional computational overhead. There is also a tradeoff between wider counts and more frequent garbage collections necessitated by counts reaching their maximum value.

(2) Every transaction which may affect the accessibility of a datum must alter the reference count. This includes creation of new data, of course, but it also includes resetting of pointer variables, binding and unbinding of function arguments, and all stores into existing allocated data. In many systems the time (or code) overhead required for altering the count on every assignment to a variable is intolerable. In a system in which the data referenced can be in secondary memory when the reference change is made (e.g. Interlisp), the time to update the count will be intolerable, even if special hardware allows simple code for in-core reference changes.

In the remainder of this paper we will discuss a specific scheme for overcoming both of these deficiencies.

Our proposals are specifically directed towards Lisp, although they are applicable to other systems which exhibit similar statistical properties. We derive our motivation from the following behavioral properties of the Lisp programs examined by Clark and Green [4]. These data indicate that when list cells (data items) are created, they tend to be "nailed down" very soon by the creation of another cell which references them. Alternatively, (although less frequently) they are soon "abandoned," with no reference to them remaining anywhere. For kinds of data other than lists (for example, strings, real numbers) it is even more likely that a newly allocated object holds a "temporary" result and will be abandoned in short order. In addition, very few cells (2 percent to 10 percent) are ever referenced by more than one cell. Put another way, allocated space tends to be used in a somewhat stack-like manner. (We conjecture that this is a natural property of computations written in languages which encourage hierarchical decomposition of programs.)

We also take the point of view that core space is at a premium, that disk accesses are expensive, but that computational power is fairly cheap. This is well in keeping with current technological trends (microcomputers and other microprogrammed processors).

1. Basic Method

We begin by borrowing a notion from commercial data processing: the *transaction file*. In an environment where most data is stored on tape, update information must be accumulated over a long period of time, and then sorted to be merged serially with the data. (We assume that our data are stored on disk, not tape, but since we are interested in interactive and even real-time applications, we have proportionately more stringent time constraints on processing time for an individual transaction.) We can identify three kinds of transactions (other than manipulation of variable values, which we do not wish to subject to *any* transaction overhead) that may affect accessibility of allocated data: allocation of a new cell from free space; creation of a pointer to a cell; and destruction of a pointer to a cell. Rather than incur the space overhead for storing reference counts with the data themselves, the immediate time overhead for adjusting them at the time of the transaction, and the paging overhead associated with random access to data, we simply save away the transactions on a (sequential) file. At suitable intervals, we read back the transactions and adjust all the reference counts, which are stored completely separate from the data.

Since the vast majority of data have a reference count of 1, we propose to keep a hash table in which the key is the cell address, a hash link as described by Bobrow [2]. The associated value is the reference count, in which only those cells with a count of 2 or more appear. Since we are not counting variable references, we logically need another table to account for data only referenced from variables on the stack or not at all; that is, all cells for which the number of references from other *cells* is zero. We will refer to these tables as the multireference table or MRT, and the zero count table or ZCT, respectively. Then bringing the tables up to date essentially requires performing the following steps for each entry in the transaction file:

For an "allocate" transaction: make an entry in the ZCT, since there are no pointers to the new datum yet. If an allocate is immediately followed by an operation which "created a pointer" to this new datum (a common case), then the pair of transactions can be ignored.

For a "create pointer" transaction: if the datum

being referenced is in the ZCT, just delete it, since the effective reference count is 1; if the datum is in the MRT, increment the associated count (unless it is at its maximum); otherwise, the datum has a default count of 1, so enter it in the MRT with a count of 2.

For a "destroy pointer" transaction: if the datum being referenced is in the MRT, delete it if the associated count is 2; if the count is at its maximum, leave it; otherwise decrement the count. If there is no entry in the MRT, the datum has a default count of 1, so enter it in the ZCT.

Once the tables are correct, precisely those data are reclaimable which are present in the ZCT and are not referenced by variables. One efficient way to find these items is to create a variable reference table (VRT) which contains in a hash table all those pointers referenced from the stack. Then the ZCT can be scanned, and any cell not referenced from the VRT can be reclaimed. Entries for reclaimed cells can be deleted from the ZCT; entries for cells referenced from variables can not be deleted.

Reclaiming a datum requires not only linking it to a free space list, but also decrementing the reference counts of any data which it may point to. This transaction may, of course, be stored for future processing just like those which occur in the course of normal computation, but since there is a good chance that a sizable structure will be freed all at once, and since the hash tables must be in core during the reclamation process anyway, it seems preferable to do the transaction processing on-the-spot. If this transaction in turn leads to new entries in the ZCT, they must be checked against the VRT to determine whether they too can be reclaimed. If a large structure is being freed, however, it may be desirable to defer some of the processing in order to disperse the disruptive effects of disk references required if the structure occupies many pages.

2. Improvements

Our statistics show that a newly allocated datum is likely to be either "nailed down" or abandoned within a relatively short time. In the former case, there will be an "allocate" and a "create pointer" transaction close together in the file; in the latter, an "allocate" and no more transactions involving that datum at all. We would like both cases to be detected as they occur, to shift more of the reclamation overhead to accompany the ongoing computation, provided we can do this without tying up additional core space.

To detect "allocate"–"create" events in the transaction file, we would like the file itself to be hashed by address. This is not practical, of course, since the file is sequential precisely so that it does not have to occupy scarce random-access memory. However, it will presumably have a reasonable buffer in core, to economize on disk references, and this buffer can be organized as a hash table. In this table, the key is the cell address and the cumulative adjustment of the reference count is the value, with a distinguished value indicating an "allocate" transaction. Then a "create" which finds an "allocate" in the table can simply remove the entry; similarly, a "destroy" which follows a single "create" can delete the entry.

To detect rapidly abandoned data, we can use the same hash table: just before finally writing out the buffer, we can scan the stack, and any "allocate" entry (which implies no subsequent "create" transactions for that datum) whose datum is not referenced from the stack can be reclaimed then and there. This process sounds expensive, but if the stack is all in core and only occupies a few thousand cells, the time is on the order of tens of milliseconds, which is certainly supportable in an interactive system. In fact, this time is comparable to that required to write the buffer on the disk.

3. Linearizing Garbage Collection

The reclamation scheme just described relies on reference counts, and therefore cannot reclaim circular structures not referenced from outside themselves. However, as remarked in the introductory section, all reference count schemes also require an independent trace-accessible-storage garbage collector (which can be run quite infrequently) to deal with this situation. Since it runs so seldom, we think of the principal functions of this independent garbage collector as *compaction* of storage and *reorganization* to produce more linear lists, as opposed to the dynamic *reclamation* of abandoned data which we have been discussing.

The algorithm derived below uses the tracing technique for linearization invented by Minsky [6], and copies the structures to be retained into a new linearized space as was suggested by Fenichel and Yokelson [5]. All of the space copied from is returned to free storage as a block. The principal difference between our scheme and the Fenichel-Yokelson scheme is that their scheme requires every list cell to be large enough to contain a mark and a pointer to the place to which that cell has been moved. In our algorithm, only those cells which are multiply referenced need an associated mark and relocation address. This becomes important when the statistics of lists allow compact encoding of individual cells; in the usual case, the cells may actually be too small to accommodate even one full-size address.

For the purposes of this compaction algorithm, the MRT is copied into a new form which has room in each entry for the relocation address for the multiply referenced cell; that is, a pointer to the cell into which it is moved when found by tracing from one of the user-accessible root pointers. This expanded MRT is also augmented by those data referenced from the stack which are not in the ZCT, i.e. those which are multiply

referenced taking the stack into account. If there is significant locality in the list structure, as might be produced by an appropriate allocation algorithm [3], access to the expanded MRT could be made nearly sequential if it is kept sorted rather than hashed.

In the algorithm below, *atom(a)*, *car(a)*, and *cdr(a)* have their usual Lisp meanings. *Nlistp(x)* is true if *x* is not a list cell, and *collectnlistp(x)* invokes that portion of the garbage collector which collects non-list items. The free variable *LN* is the location of the next available *cons* cell in the space into which we are copying. *Incr(LN)* will increment this quantity, taking account of any nonuniformities of the space. *Docons(LN,a,d)* inserts in the *cons* cell at *LN* the pointers to *a* and *d* in the *car* and *cdr* fields, respectively. *Claim(oldhalfspace)* makes the old halfspace free. The set of root pointers can be sorted by address to minimize paging. For a multiply referenced cell *c*, we will refer to the entry for *c* in the expanded MRT as *mrt*[*c*]. We assume that *mrt*[*c*] = 0 if and only if *c* has not yet been seen in the compaction-collection cycle. If *x* is a cell which is not multiply referenced, then *mrt*[*x*] = −1.

Here is the algorithm:

```
compactor( ) =
prog[[p]
  initialize(LN);
  (for each root pointer p) p := GCcopy(p);
  claim(oldhalfspace);
  ]
GCcopy(p) =
  nlistp(p)→collectnlistp(p);
  mrt[p]>0→mrt[p];          p has previously been copied, return
                            new location
  T→prog[[a,d,n]
    n:=LN; incr(LN);
    (mrt[p]=0→mrt[p]:=n); p not yet copied but is multiply
                            referenced, save new address
    d:=GCcopy(cdr(p));
    a:=GCcopy(car(p));
    docons(n,a,d);          make new cons cell
    return(n);
    ]
  ]
```

4. Incremental Linearization

Linearization of data even within a single page may be desirable if linear storage of lists leads to a particularly compact representation (as discussed in [4] and [7]). Complete compaction of a system with hundreds of thousands of words of data would take on the order of several minutes, since most of the critical data would not fit in core. It appears that incremental compaction, if done sufficiently often, will require relatively few disk references, and take five or ten seconds with relatively small working set.

The basic idea is that immediately after a compaction, one "draws a line" which separates all data structures created before the compaction from those created after. We will call the data created since the last compaction "new data." Incremental linearization will be done only to this new data. This assumes that the reference count scheme will take care of most of the old data released, and full linearization will be done sometime if the rest of storage needs to be reclaimed. The same basic algorithm works, with a few straightforward modifications—specifying a different set of root pointers, segmenting the MRT and ZCT tables, and checking for references from the new data to the old data.

Root pointers. In the global algorithm, Compactor, the set of root pointers is all the permanent or externally accessible places for storage of a pointer, and the stack if one is active at that time. For incremental linearization, the root pointers are just those cells in the old data which refer to the new data, including externally accessible cells which point into the new data space. One way of determining this set of root pointers is by a linear sweep of the entire virtual address space, checking each pointer encountered. This may be reasonable on some hardware configurations where the time to do a linear sweep over the virtual space is relatively low, and the check can be done on the fly. To eliminate much of this sweep, a bit can be kept for each page of old data, and it can be set to 1 if and only if it has been written on since the line was drawn. (This type of check is easy to implement in a microprocessor). A third alternative is to augment the transaction mechanism. For every "create-pointer" transaction which changes a pointer from the old data to the new data, the location of the cell changed could be recorded. This set of locations, sorted with duplicates removed would form the set of root pointers (along with the stack entries).

The MRT and ZCT tables. In compacting the new data, only that portion of the MRT is needed which contains reference counts for the new data area. This implies that the MRT might be segmented by address of the cell referenced. This segmentation has the added advantage that if the entire MRT cannot fit in core during full compaction, the transaction processor can easily make multiple passes over the transaction file, filtering entries on the basis of a simple magnitude test. A similar argument holds for the segmentation of the ZCT by address.

The algorithm for compactor can be modified by replacing references to *mrt*[*p*] by references to *lmrt*[*p*], a local multiple reference table. One other change must be made; where the *mrt* table is checked, a check should be made previously to see if *p* is a pointer into the old data. If so, the pointer itself should be returned as the *lmrt*[*p*].

5. Further Observations

It appears that the reclamation process can be done almost entirely by a processor and local memory sepa-

rate from the processor and memory used for the main computation. The ZCT and MRT need never be referenced by the main processor. The transaction buffers can be delivered to the auxiliary processor and used to update the reference count tables as soon as they arrive. When the main processor reaches a convenient point, it can transmit the current stack to the auxiliary processor; the auxiliary processor can look the pointers up in the ZCT as they arrive, and obtain a list of reclaimable cells immediately upon receiving the last piece of the stack. There still remains the problem of reclaiming cells which become free when the last cell referencing them is reclaimed; however, since no transactions will ever reference a cell once that cell is genuinely inaccessible, the main processor can ship pages to the auxiliary processor at its leisure, and the auxiliary processor lags behind the main processor in the identification of reclaimable cells.

Even without the use of a second processor, it appears feasible to use this method in a real time environment. To do this we must be able to identify the storage needs of those operations which must run in real time. We can then construct a second allocation zone, with its own transaction queue and count tables (perhaps both kept in core), for use by the real time program. For this to work properly, we must guarantee that any transaction which affects the real time zone be placed on the real time queue, and that the reclaimer for the real time zone must run often enough to empty the queue and maintain adequate free storage.

Received November 1974; revised June 1975

References

1. Bobrow, D.G. Storage management in LISP. In *Symbol Manpulation Languages and Techniques*, D.G. Bobrow, Ed., North-Holland Pub. Co., Amsterdam, 1968.
2. Bobrow, D.G. A note on hash linking. *Comm. ACM 18*, 7 (July 1975), 413–415.
3. Bobrow, D.G., and Murphy, D.L. Structure of a LISP system using two-level storage. *Comm. ACM 10*, 3 (March 1967), 155–159.
4. Clark, D., and Green, C.C. An empirical study of list structure in LISP. Stanford U., Stanford, Calif. *Comm. ACM* (to appear).
5. Fenichel, R.R., and Yochelson, J.C. A LISP Garbage-Collector for Virtual-Memory Computer Systems. *Comm. ACM 12*, 11 (Nov. 1969), 611–612.
6. Minsky, M.L. A LISP garbage collector algorithm using secondary serial storage, rev. Memo No. 58, M.I.T. Artificial Intelligence Lab., M.I.T., Cambridge, Mass., 1963.
7. Van der Poel, W.L. A manual of HISP for the PDP-9. U. of Delft, Delft, Netherlands, 1974.

Programming Techniques

G. Manacher, S.L. Graham Editors

Faster Retrieval from Context Trees

Ben Wegbreit
Xerox Palo Alto Research Center

Context trees provide a convenient way of storing data which is to be viewed as a hierarchy of contexts. This note presents an algorithm which improves on previous context tree retrieval algorithms. It is based on the observation that in typical uses context changes are infrequent relative to retrievals, so that data can be cached to speed up retrieval. A retrieval is started from the position of the previous retrieval and auxiliary structures are built up to make the search rapid. Algorithms for addition and deletion of data and for garbage collection are outlined.

Key Words and Phrases: context trees, frame problem, variable bindings, data structures

CR Categories: 3.69, 3.74, 4.10

Copyright © 1976, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Author's address: Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.