Garbage Collection

Andrew W. Appel

June 25, 1990

In McCarthy's 1960 Lisp system, linked lists could be freely created by the CONS operator, and accessed by the CAR and CDR operators. The revolutionary innovation of Lisp was that the programmer did not need to tediously reclaim unneeded list cells. This made the manipulation of lists as mathematical values much easier, and as a result, Lisp endures to this day.

More recent language implementations require the garbage collection not just of lists but of function activation records, function closures, executable code, strings, and other data structures. Automatic management of dynamic storage is increasingly regarded as a necessary feature of a modern programming language.

This chapter is a selective survey of garbage collection algorithms; Cohen's survey [11] is a comprehensive and well-organized review of the literature to 1980.

1 Mark and sweep

The first Lisp garbage collector used a depth-first search to mark all nodes accessible from a set of "root" locations (e.g. global variables), and then did a sweep through the address space gathering all accessible cells onto a free list. McCarthy[17] explains it concisely:

> Each register [memory location] that is accessible to the program is accessible because it can be reached from one or more of the base registers by a chain of car and cdr operations. When the contents of a base register are changed, it may happen that the register to which the base register formerly pointed cannot be reached by a car-cdr chain from any base register. Such a register may be considered abandoned by the program because its contents can no longer be found by any possible program; hence its contents are no longer of interest, and so we would like to have it back on the free-storage list. This comes about in the following way.

> Nothing happens until the program runs out of free storage. When a free register is wanted, and there is none left on the free storage list, a reclamation cycle starts. First, the program finds all registers accessible from the base registers and makes their signs negative. This is accomplished by changing the sign of every register than can be reached from it by a car-cdr chain. If the program encounters a register in this process which already has a negative sign, it assumes that this register has already been reached.
>
> After all the accessible registers have had their signs changed, the program goes through the area of memory reserved for the storage of list structures and puts all the registers whose signs were not changed in the previous step back on the free-storage list, and makes the signs of the accessible registers positive again.

McCarthy doesn't say what the marking algorithm is, but we can assume it might be a depth-first search. The problem with depth-first search is that it requires an auxiliary stack, which might require as much storage as all the accessible list cells. A clever technique, discovered independently by Peter Deutsch and by Herbert Schorr and W. M. Waite in 1965 and described in section 2.3.5 of Knuth[15], is known as *pointer reversal*: the stack can be kept in the graph itself, with only one bit per node extra storage. As a chain of pointers is followed from the root by the depth-first search, each already-traversed node is made to point to the previous node. At any point, the current and next nodes are remembered in registers. Then, when it's time to "pop" the stack, previous nodes can be retrieved at will.

2 Copying collection

The problem with mark-and-sweep garbage collection is that each collection touches the entire memory, even if most of it is garbage. *Copying* collection avoids touching the garbage. Memory is divided into two spaces, and list cells are allocated in one space until it fills. Then all the accessible cells are copied into the other space (and the pointers within them are adjusted to point at the new copies). The copied cells don't come close to filling the new space (because there should have been some garbage in the old space). Then the program is resumed, allocating new cells in the remainder of the new space. When the new spaces fills up, the roles of the spaces are reversed and the live cells are copied back.

The accessible cells can be copied in any order. Fenichel and Yochelson[13] first presented copying collection with a depth-first search using pointer-reversal, and

Cheney[10] demonstrated a breadth-first version. Cheney's algorithm has become a basis for many recent garbage collection algorithms, so we will study it in detail.

At the beginning of a garbage collection, we have a set of *root* registers (which might be machine registers or global variables); a *from-space* containing accessible list cells and garbage; and a *to-space* which is initialially empty. We will assume that each list cell occupies two consecutive memory words that are addressible separately. We will also assume that pointers are distinguishable from non-pointers (about which, more later).

We wish to copy the graph of accessible cells from from-space into to-space by a breadth-first search. The classical breadth-first search algorithm requires a queue of unprocessed graph-edges, which could grow quite large. Cheney's algorithm cleverly uses a portion of to-space as the queue: there are two pointers into new space, called *scan* and *next*, which define the head and tail of the queue. Initially, *scan* and *next* point to the beginning of new-space.

In the data structure originally residing in from-space, a given node may have many pointers to it. When the node is copied to to-space, it is important to record the address of the copy, so that all the pointers can be properly updated. We record this information in the node itself, overwriting the first field with a *forwarding pointer*, after first copying all the fields to to-space (see Figure 1).

The fundamental operation of Cheney's algorithm is `forward`ing: converting a pointer to from-space into a pointer to to-space by following a forwarding-pointer (if the cell has already been copied) or by making a copy and installing a forwarding pointer:

```
forward(p) = IF p points into from-space
               THEN IF mem[p] points into to-space
                      THEN RETURN mem[p]
                      ELSE mem[next]   := mem[p]
                           mem[next+1] := mem[p+1]
                           mem[p] := next
                           next := next+2
                           RETURN mem[p]
               ELSE return p
```

If p doesn't point into from-space, it won't be affected by this garbage collection. If `mem[`p`]` already pointed into to-space, it must have been a forwarding pointer; there is no other way for such a pointer to arise inside from-space.

Now, here's Cheney's algorithm:

```
scan := next := beginning of to-space
```

```
FOR each root  p
     p := forward(p)
WHILE scan < next
     mem[scan] := forward(mem[scan])
     scan := scan+1
```

Now let us compare the efficiencies of mark-and-sweep versus copying collection[2]. The mark phase of a mark-and-sweep collection takes time linear in the amount A of accessible data, and the sweep phase takes time linear in the size of memory M. So the cost of one collection is $c_1A + c_2M$ for some constants c_1 and c_2. After a collection, there are $M - A$ cells free before the next collection; we can calculate the amortized cost per cell allocated for mark-and-sweep collection:

$$C_{\mathrm{mark-and-sweep}} = \frac{c_1A + c_2M}{M - A}$$

Similarly, for copying collection, a depth-first or breadth-first copy takes time linear in the number of cells copied. The number of cells allocable before the next collection is only $M/2 - A$, however, because we have divided memory in half. Thus, the amortized cost per cell for copying collection is

$$C_{\mathrm{copying}} = \frac{c_3A}{\frac{M}{2} - A}$$

Since memories are becoming larger and larger, it is interesting to note what happens in the limit of very large memories. The number of accessible cells A is, independent of the memory size; it depends only on the algorithm and its input. When we take the limit as M approaches infinity, we find that the amortized cost of mark-and-sweep collection approaches c_2 per allocation, while the amortized cost of copying collection approaches zero.

Furthermore, copying collection compacts the useful data in memory, thereby reducing fragmentation; and the free memory is contiguous, which simplifies allocation of new cells. These advantages become important if records of different sizes are allocated.

For these and other reasons, copying collection is usually to be preferred. The remaining uses of mark-and-sweep collection in modern systems are usually where it is hard to distinguish pointers from integers[9], so that one cannot safely move cells from one place to another.

3 Reference counts

Instead of having the garbage collector traverse the accessible memory, the allocating program can keep track of which cells are still pointed to. [12] Each time a new pointer to a cell is created, its *reference count* is incremented; when a pointer is destroyed, the cell's reference count is decremented. When the reference count goes to zero, the cell can be reclaimed; this can happen immediately, or there can be a periodic sweep phase that looks for zero-count cells. There are two serious problems with reference counting schemes:

1. Cyclic data structures cannot be reclaimed; an inaccessible cycle of cells will still have nonzero reference counts.

2. Maintaining the reference counts is very costly; every time a pointer is stored to memory, there must be an additional fetch, increment, and store to change the reference count (and if the stored-to destination contains a valid pointer, there must be yet another fetch, decrement, store, and test-for-zero to destroy the old pointer).

These problems can be ameliorated by using mark-and-sweep as a backup algorithm for reference counts; in this case, the size of the reference count field in each object need not be large; when it reaches the maximum value, it is never decremented.[15] The mark-and-sweep collector reclaims all cyclic objects and those unreachable objects with "stuck" reference counts.

4 Incremental collection

A single garbage collection can take a long time, and in real-time or interactive applications it's undesirable to have a long pause in the computation while the collector is invoked. Baker's algorithm[7] does a little bit of garbage collection at a time.

We consider a machine model with a finite set of registers; any value from memory must be fetched into a register before being operated upon. Now, we use Cheney's algorithm, but we let the allocating program resume execution as soon as we have forwarded all the registers. Any cells allocated by the program are put at the end of the to-space, in an area where they won't interfere with *scan* and *next*.

The following invariant will be maintained by the allocating program:

Registers never point to from-space.

This will certainly hold as soon as all the registers are forwarded. But since the unforwarded locations in to-space can contain pointers to from-space, the allocating program has the duty to forward every pointer that it fetches from memory. In the long run (especially in large memories), most pointers fetched from memory will not point into from-space, and thus will return quickly from the *forward* procedure as shown above. However, there must be at least a comparison and branch after each fetch, which takes at least one instruction (unless special hardware is provided, as on a Lisp machine). Since fetches comprise about 30% of all instructions, the overhead of Baker's algorithm is fairly high.

While the allocating program is executing, the garbage collector must also execute, trying to make *scan* catch up to *next* as in Cheney's algorithm. This can be accomplished even in a sequential system by letting the *cons* procedure also forward k pointers to advance *scan* by k. What remains are management details, like adjusting k for best performance and concurrency.

Aside from the overhead of checking the result of every fetch, Baker's algorithm has the same efficiency as Cheney's algorithm, since exactly the same set of cells will be copied from from-space to to-space.

5 Generational Collection

In 1983, Lieberman and Hewitt[16] made the following observations about Lisp programs:

1. Newer cells tend to point to older cells.
2. Newer cells tend to be shorter-lived than older cells.

When one writes (CONS A B) in a program, one constructs a newer cell that points to two older cells. The only way that an older cell can point to a newer cell is if its CAR or CDR is modified after it is created, which is rare.

When a cell has been accessible for a long time, it's typically part of a global data structure that will continue to be necessary. New cells tend to be intermediate results of computations that will soon be useless. In this respect, dynamically allocated list cells are quite unlike radioactive atoms, whose future lifetimes are quite unrelated to their past lifetimes.

Now, we can take advantage of these two observations. Observation 2 implies that we should concentrate our garbage collection effort on the newer cells (which have a higher proportion of garbage), and observation 1 implies that this is possible.

Allocated cells will be kept in several distinct areas G_i of memory, called *generations*. Cells in the same generation are of similar age, and all the records in generation G_i are older than the records in generation G_{i+1}. This implies (by observation 1 above) that for $i < j$, there are (almost) no pointers from G_i into G_j (see Figure 2).

Figure 1: All the records in generation G_2 are newer than the records in G_1; the oldest records are in G_0. The pointer represented by the dashed line is anomalous; it must have been created by an assignment that modified an old object.

Consider the newest generation, G_n. The copying garbage collector may be applied to this generation, using the global variables and runtime stack as a root set. Because (almost) no pointers in the other generations can possibly point to records in G_n, the other generations (almost) need not be considered as roots. The garbage collector may copy the reachable objects of G_n into a new generation $G_n{}'$ without touching the other generations. This is advantageous because G_n is expected (by observation 2 above) to have a higher proportion of garbage than any of the other generations.

In general, the garbage collector can work on generation G_i if it also works on all the newer generations; that is, it may work on any subset $G_i, G_{i+1}, ..., G_n$ of the generations. For greatest efficiency, the garbage collector should usually copy just G_n, but occasionally copy the older generations.

This algorithm depends on the fact that cells contain pointers only to older cells. In the presence of assignments to fields of old cells, this acyclic condition will not hold. Lieberman and Hewitt[16] handle this problem by making such fields point indirectly through an "assignment table." A simpler scheme devised by Ungar[19] keeps a set of addresses of cells that point to newer cells; this set must be maintained with the help of the allocating program—whenever it assigns a pointer into an existing cell, it must add the address of that cell to the modified set. In some implementations, there is just one modified set; at the other extreme, there is a modified set for each pair (i, j), $i < j$, for pointers from G_i into G_j.

Then, when collecting generations $G_k \cdots G_n$, the modified sets of generations $G_0 \cdots G_{k-1}$ must be treated as root pointers.

The management of the memory areas for the generations, modified sets, etc. can get tedious; things can be simplified by having only two generations[4], and performance doesn't suffer much.

In practice, generation garbage collection performs extremely well; though it is typically implemented only for programming languages (like Lisp, Scheme, ML, Smalltalk, etc.) where modification of older objects is relatively rare. In general, it is widely agreed that garbage collection need not take more than 5

6 Virtual memory hardware

The disadvantage of Baker's algorithm—the *only* thing that really causes it to be less efficient than Cheney's "stop-and-copy" algorithm—is the checking of every pointer fetched into registers to make sure it's not a from-space pointer. The disadvantage of generational collection—the *only* thing that could possibly make it slower than Cheney's algorithm—is the checking of every pointer assigned into an existing list cell.

Both of these checks can be approximated very cheaply by the virtual memory hardware available on most computers. The Appel-Ellis-Li algorithm[6] is a concurrent garbage collection algorithm (the garbage collector and the allocating program can run simultaneously, on the same processor or on different processors) with very low overhead and low latency—the allocating program is rarely suspended for more than a few dozen milliseconds. The algorithm is like Baker's algorithm, except that immediately after the registers are forwarded, the to-space is protected (by turning off the read and write accessibility bits in the virtual memory page tables). Then, as each page of to-space between *scan* and *next* is forwarded by the garbage collector, that page is made accessible.

When the allocating program fetches from a protected page of to-space, the collector immediately forwards all the pointers on that page, and makes the page accessible. The allocating program can then resume, and the only delay has been the time to forward one page, which is a few milliseconds. Newly allocated cells are put in an accessible (unprotected) region of to-space, and cause no page faults.

Thus, the chief disadvantage of Baker's algorithm—the need to spend an extra instruction on every fetch to check for a from-space pointer value—has been eliminated; it has been replaced by a cheap (practically free) hardware test that's already being performed by the hardware anyway for other reasons. Since the number of pointers that get forwarded is still the same as in Cheney's algorithm, the only extra overhead will be in the page-fault handler. And by the principle of locality, we would expect that a typical program will not touch many different pages before the collector has a chance to forward all the pages of to-space, and experience with one implementation has found the performance of this algorithm to be quite good.

Similarly, generational garbage collection can make use of virtual memory hardware.[18] Whenever a pointer is stored into an older generation, the location modified must be recorded in an "anomalous references" set. But we can mark all the older generations read-only in the page tables, and the first assignment to an older cell will cause a page fault. We handle the fault by putting the page address in an anomalous-references set, and marking the page read-write. Then, at garbage collection time, the collector must scan the entirety of each page in the set for possible roots of the newest generation(s).

In imperative programming languages like Pascal, Modula, Mesa, etc., where storing pointers into existing cells is very common, it's clear that observation 1 (about generational garbage collection) won't hold. However, there is no good evidence that there is *any* programming language that violates observations 1 and 2 over a large scale, i.e. we might modify observation 1 as follows:

1. A cell usually points to cells that are older than it, or at least not much newer.

Thus, most assignments of pointers will be into the newest generation, where there will be no page fault. If we assume that anomalous references in the older generations exhibit some locality (tend to be rare or clustered), then the number of distinct pages in the modified set will be small, and thus the number of page faults will also be small. With this scheme, assignments into the newest generation or into any recently-modified cell will not incur the cost of a page fault. Thus, this scheme might work well even for imperative languages of the Algol family, where the modified Observation 1 might hold.

7 Representation of data

In most garbage collection schemes (but not all[9]) it is necessary for the collector to be able to distinguish pointers from non-pointers (characters, integers, reals, etc.); and to determine the length and format of each record. A language's compiler and runtime system must agree on the format of runtime data; many schemes have been used successfully [1][14][5].

When the word length of the machine is longer than the size of an address, then each pointer or integer can be tagged with a descriptor. A pointer to a two-word record (a CONS cell, for example) would have a different tag than a pointer to a five-word record, etc. Then the garbage collector can know how much data to copy (or mark, in a mark-and-sweep collector) as it follows each pointer.

When the word length is not much longer than an address (as is increasingly the case), then a 1-bit tag can be used just to distinguish pointers from non-pointers. On modern 32-bit byte-addressible machines, where pointers tend to be multiples of 4 anyway, the low-order bit is typically used for this tag. Then it is necessary to put a descriptor at the beginning of each record explaining its length and format. A refinement of this method is to group records of a commonly-used format together in one range of addresses; then one descriptor suffices for a large group of records, as long as the garbage collector knows about the address-range convention.

In some implementations, it is undesirable to restrict the range of representable integers, so even a 1-bit tag is an intrusion. One solution is to encode in the record descriptor not just the number of fields, but an indication of which ones are pointers. This solution works well for Algol-like languages where the types of records are known at compile-time, but is awkward for languages like Lisp and ML, where the descriptor would have to be constructed on-the-fly for each allocated record.

For statically-typed languages like Pascal or ML, the compiler can provide a map of the type system to the collector.[3] Then, if the collector is told the types of the root pointers, it can follow the type-map with one hand while it follows the data with the other, and no tag bits at all are required! For polymorphic languages like ML it's not quite so easy, however, as some work is required to determine the types of the root pointers.

Finally, in some environments it is impossible to tell whether a word is a pointer or a non-pointer. This may be because of insufficiently strong compile-time type systems combined with a lack of runtime type information (e.g. in the C language), or in mixed-language systems. Mark-and-sweep collection can still be used here [9]; some objects may be spuriously marked because an integer was incorrectly seen as a pointer, but these objects just waste some storage and don't cause incorrect program behaviour. Obviously, copying and compacting collectors are more difficult to use here, since a "potential" pointer cannot be adjusted (when the object it points to is moved) in case it is really an integer. However, copying collection is still possible with sufficient care [8].

References

[1] John Allen.
Anatomy of LISP.
McGraw-Hill, New York, 1978.

[2] Andrew W. Appel.
Garbage collection can be faster than stack allocation.
Information Processing Letters, 25(4):275–279, 1987.

[3] Andrew W. Appel.
Runtime tags aren't necessary.
Lisp and Symbolic Computation, 2:153–162, 1989.

[4] Andrew W. Appel.
Simple generational garbage collection and fast allocation.
Software—Practice/Experience, 19(2):171–183, 1989.

[5] Andrew W. Appel.
A runtime system.

Lisp and Symbolic Computation, 3(to appear), 1990.

[6] Andrew W. Appel, John R. Ellis, and Kai Li.
Real-time concurrent collection on stock multiprocessors.
SIGPLAN Notices (Proc. SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation), 23(7):11–20, 1988.

[7] H. G. Baker.
List processing in real time on a serial computer.
Communications of the ACM, 21(4):280–294, 1978.

[8] Joel F. Bartlett.
Compacting garbage collection with ambiguous roots.
Technical Report 88/2, Digital Western Research Laboratory, Palo Alto, CA, 1988.

[9] Hans-Juergen Boehm and Mark Weiser.
Garbage collection in an uncooperative environment.
Software—Practice/Experience, 18(9):807–820, 1988.

[10] C. J. Cheney.
A nonrecursive list compacting algorithm.
Communications of the ACM, 13(11):677–678, 1970.

[11] Jacques Cohen.
Garbage collection of linked data structures.
Computing Surveys, 13(3):341–367, 1981.

[12] G. E. Collins.
A method for overlapping and erasure of lists.
Communications of the ACM, 3(12):655–657, 1960.

[13] Robert R. Fenichel and Jerome C. Yochelson.
A LISP garbage-collector for virtual-memory computer systems.
Communications of the ACM, 12(11):611–612, 1969.

[14] Ralph E. Griswold and Madge T. Griswold.
The Implementation of the Icon Programming Language.
Princeton University Press, Princeton, NJ, 1986.

[15] Donald E. Knuth.
The Art of Computer Programming, vol. I: Fundamental Algorithms.
Addison Wesley, Reading, Mass., 1973.

[16] Henry Lieberman and Carl Hewitt.
A real-time garbage collector based on the lifetimes of objects.
Communications of the ACM, 23(6):419–429, 1983.

[17] John McCarthy.
Recursive functions of symbolic expressions and their computation by machine - I.
Communications of the ACM, 3(1):184–195, 1960.

[18] Robert A. Shaw.
Improving garbage collector performance in virtual memory.
Technical Report CSL-TR-87-323, Stanford University, 1987.

[19] David M. Ungar.
The Design and Evaluation of a High Performance Smalltalk System.
MIT Press, Cambridge, Mass., 1986.