Brief survey of garbage collection algorithms

David R. Chase

1987; reformatted October 2000

1 Introduction

In this paper (chapter) I describe several classes of garbage collection algorithms, and provide a few examples (both real and hypothetical) of each. This collection is by no means complete; Cohen [Coh81] presents an exhaustive collection, but I am more interested in describing a few representative collectors in detail and providing examples for a later paper (chapter) on garbage collection and optimization. The collectors are also chosen for comparison and contrast; the various algorithms are quite different, but often depend upon similar assumptions about program behavior for good performance. Empirical studies support many of these assumptions.

2 Garbage Collection

The garbage collection problem: Given a finite memory and a program without explicit instructions for "freeing" (making available for re-use) memory that the program uses, discover portions of memory that are no longer referenced by the program and make them available for its re-use as necessary. LISP is the most famous example of a language that requires garbage collection, but it is also used in Cedar [Rov85], Simula [Arn72], Smalltalk [Kra83], SETL, SNOBOL and almost all functional languages[1]. Garbage collection may appear in relational database systems, file systems [Bro85, RT78], and

[1] The exception to this rule is "string reduction", in which values are not shared with pointers. In this instance it is always possible to know when an object may be recycled. When objects are shared by reference, however, allowing the programmer to specify when objects may be recycled removes any guarantee of functional semantics.

multi-processor systems [GP81]. In this paper I will focus on garbage collection of storage in a program's address space.

As a more abstract form of the problem (taken from Dijkstra *et al* [DLM+78]), consider a directed graph with a fixed number of nodes. Each node has a left out-edge and a right out-edge, though either one or both of these edges may be missing. In this graph there is a fixed set of "root nodes." A node is "reachable" if it lies on a directed path from a root node; any node that is not reachable is said to be a "garbage node." The subgraph consisting of the reachable nodes and their interconnections is called the "data structure." The data structure subgraph can be modified by the following operations:

1. Redirect an existing outgoing edge of a node towards an already reachable node.

2. Redirect an existing outgoing edge of a node towards a not yet reachable node with no outgoing edges.

3. Add—where there is no outgoing edge—an edge pointing to an already reachable node.

4. Add—where there is no outgoing edge—an edge pointing to a not yet reachable node with no outgoing edges.

5. Remove an outgoing edge of a reachable node.

Operations 1, 2, and 5 *may* disconnect a node from the data structure, causing it to become garbage. Operations 2 and 4 consume garbage nodes by making them reachable. It is the task of the garbage collector to discover garbage nodes and make them available to the process altering the graph—the "mutator." Garbage nodes discovered by the collector are known as "free" nodes (to distinguish them from garbage nodes that the collector has not discovered).

In classical garbage collection the graph is represented in such a way that discovering the successors of a node is a primitive action, and the nodes may be addressed without reference to the rest of the graph. Finding predecessor nodes is not a primitive action; this is what makes garbage collection difficult. In addition to finding unreachable nodes, garbage collectors are sometimes given the responsibility of compacting storage to reduce fragmentation and improve locality, and can be used to transform data structures into more

compact representations [Bak78, BC79]. Garbage collection in file systems, databases, and distributed systems can be much more difficult; the synchronization implicit in the single process case disappears and cheap primitive operations are often neither cheap nor "primitive". Sometimes it is practical to place restrictions on the graphs that can be built in order to simplify garbage collection. In the sections that are described below there will be some examples of collection of restricted graphs.

2.1 Reference counted garbage collection

In a reference counting garbage collector, each node in the graph also contains a count of the number of predecessors of that node. The five operations on the graph are augmented to maintain these counts. Clearly, any non-root node with a count of zero is not reachable and may be recycled.

Reference counting has been proposed for distributed systems [DR81] because a node's reference count summarizes (locally) global information about the storage graph. In addition, reference counting garbage collection does not need tight synchronization; reordering is allowed, provided that dereference operations do not get moved ahead of reference operations. However, a *naive* reference-counting garbage collector will not reclaim all garbage nodes, because cycles in garbage graphs will prevent the counts of nodes in the graph from falling to zero. (Below are summaries of several non-naive reference counting papers.) It is also necessary to set aside enough space in each node to store a number as large as the number of nodes in the graph, or else the count may overflow. Doing this typically increases the size of a node by 50%, but not doing this requires code to check for overflow at each reference count adjustment. Nodes with overflowed reference counts will never be collected, but in practice the amount of storage lost in this way is not significant. Reference counting has a reputation for being very slow [Bak78, BS83] when compared to some of the other methods presented here, but it is still being studied because of its apparent suitability to concurrent and distributed garbage collection.

2.2 Mark-and-sweep garbage collection

A (classical) mark-and-sweep garbage collector discovers garbage by interrupting the mutator, searching from the roots to discover and "mark" all reachable nodes, and then sweeping sequentially through the nodes to collect

all the unmarked (and hence unreachable) nodes. This method collects all garbage and requires only an additional bit of storage per node for marking purposes, but has cost per collection proportional to the size of the entire graph and does not compact the graph. This algorithm is easy to understand, but introduces several problems. After a collection, free and active blocks are interspersed throughout the program's address space. This external fragmentation can increase paging in a virtual memory environment and complicate allocation of objects. In the worst case, allocation will be impossible because no free block of memory is large enough to satisfy a request for memory. If all objects are the same size, however, external fragmentation is not a problem. Naive marking algorithms present another problem; the simplest algorithms are recursive, but little memory is available to hold a recursion stack when garbage collection is required. Knuth describes several algorithms that have been developed to traverse graphs when auxiliary memory is limited [Knu68, p.417]. The cost of a garbage collection is a more serious problem; increasing the size of the graph (amount of memory available) reduces the frequency of garbage collection, but increases the cost of a sweep phase in proportion to the amount of free memory. In large interactive or real time systems this is completely unacceptable [REFERENCE!!].

2.3 Locality and compaction

In a computer with virtual memory both mark-and-sweep and reference-counting garbage collection tend to exhibit poor locality of reference and may run slowly because of excessive page faulting. This is so for mark and sweep collection because it fills an address space with objects before performing a collection; after a collection, objects are sparsely scattered throughout the address space.

Reference counting does not scatter objects throughout an address space because it frees objects as soon as they become unreachable, but it still suffers from poor locality of reference. The reference count field stored with each object increases the object's size, thus reducing the density of objects in memory by some fraction. Each time a pointer to an object is created or destroyed, the object's reference count must be adjusted. Though it scatters objects less, reference counting touches objects so often that it has poor locality.

The locality of mark-and-sweep collection can be improved if the objects in memory are periodically "compacted"; that is, if the objects are arranged

so that there is no free memory separating them. This is not so effective for reference counted objects because the primary causes of non-locality, references to update counts and the "fluffing" effect of the count fields, are not affected by compaction. The locality of reference counting is improved somewhat with storage compacting algorithms that "linearize" lists by arranging objects in the storage graph into some depth-first visit order; in this case, pointers and their targets are usually moved much closer together, so count adjustments are more frequently local. Of course, linearization also improves the locality of mark-and-sweep collection. Empirical studies of LISP list structure by Clark and Green [CG77] showed that on the average over 98 percent of list pointer **cdr**s point to the next cell after linearization along **cdr** edges.

Locality is also improved by the use of special encodings that reduce the size of objects. **cdr**-coding [BC79] permits the efficient use of only two bits for the **cdr** field of a **cons** cell; this compression places more cells on a single page and thus increases locality. When combined with a linearizing storage compactor, **cdr**-coding is expected to cut memory consumption nearly in half, both improving locality and reducing the rate of garbage collection[2] Figure 1 shows an example of **cdr**-coding. In the example, the 'C' (for *cons*) tag means that the two word **cons** cell contains a pointer to its **car** and a pointer to its **cdr**. An 'N' (for *next*) tag means that the first word of the **cons** cell contains a pointer to its **car** and the location of the **cdr** is the immediately following word; the pointer is implicit. A tag of 'E' (*end*) means that the **cdr** is the value (pointer to) NIL, while an 'F' (*forwarding*) tag means that the first word (normally the **car** pointer) points instead to a forwarded (relocated) **cons** cell. In the example, all three lists are equal to the LISP list "*(a b)*".

Reference counting's locality may be improved by separating counts from objects. This increases locality by concentrating count adjustments into a smaller section of memory. Transaction-based reference count systems [DB76] can postpone count updating, allowing the count tables to be paged out until the transactions are processed. Delayed collection, however, will result in less efficient use of memory and less locality because of external fragmentation.

[2] The effect on garbage collection frequency can be quite minor if the active graph fills only a small part of available memory.

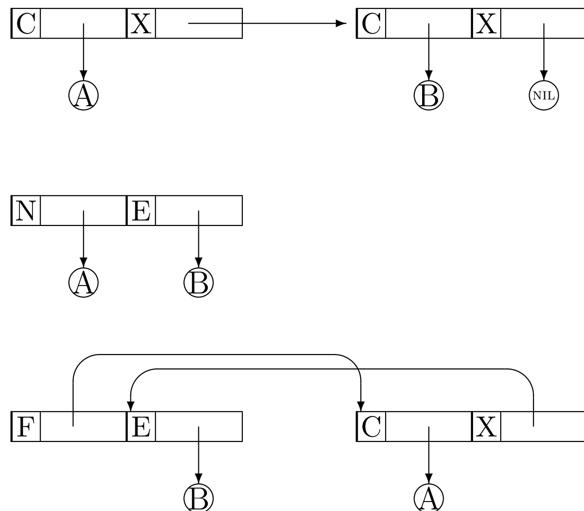

Figure 1: Examples of **cdr**-coding

2.4 Copy and compact garbage collection [Che70] [FY69] [Bak78]

Copy and compact garbage collection takes a different approach from reference-counted or mark and sweep garbage collection. Reference counting and mark-and-sweep work to generate or discover unused nodes, and attach them to a free list; copy-and-compact collectors work to discover active nodes, and copy them into "fresh" memory.

The garbage collector maintains two equally-sized areas of memory called the *fromspace* and the *tospace*. Each of these spaces corresponds to a graph in the abstract model. The fromspace contains only free nodes that will be used in the next garbage collection; the tospace contains the free space, reachable nodes, and garbage nodes. When the tospace runs out of nodes, the roles of tospace and fromspace are reversed (this is called a *flip*) and all the reachable nodes in the new fromspace are copied into the new tospace (the old fromspace). When this process is complete all the nodes in the fromspace are abandoned. The free nodes remaining in the tospace are arranged in ascending order, so the nodes copied into it are compacted into one end of the tospace and subsequent allocations are very quick.

Copy algorithms for early versions of copy-and-compact were either recursive or based on the Deutsch-Schorr-Waite marking algorithm [FY69], but

Cheney [Che70] discovered an elegant and efficient copying algorithm that is used today. Cheney's method for copying uses pointers S, B, and T ("scan",

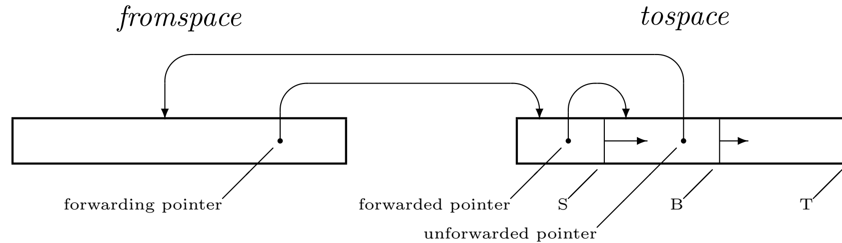

Figure 2: Cheney's copying algorithm

"bottom" and "top") into the tospace; B marks the bottom of the free area (nodes below B are allocated and reachable), T marks the top of the free area (nodes between B and T are free), and S divides the reachable nodes into those whose out-edges point into the tospace (below S) and those whose edges point into the fromspace (between S and B). At the start of a garbage collection, the root nodes are copied into the tospace, and S points to the beginning of the tospace. The collector then advances S until it is equal to B, moving nodes into the tospace (and incrementing B) in the process. Eventually (if the garbage collection is to succeed) S will reach B and all out-edges will point into tospace. Each node must be large enough to contain its forwarding pointer (overwriting the old contents), and an additional bit is needed to indicate whether or not the node contains a forwarding pointer.

This method has two principal advantages; the cost of a garbage collection is proportional to the amount of reachable memory, so that (unlike mark-and-sweep) increasing the amount of memory to reduce the frequency of garbage collection will not increase the cost of an individual garbage collection, and memory is automatically compacted after each collection. A variation of this algorithm uses an additional scan pointer to automatically linearize lists. The second adbantage is that allocation (ignoring the amortized cost of garbage collection) is very rapid because the free area is compact and may be managed with a pointer to each of its bounds.

This algorithm has a few disadvantages; it guarantees that one-half of the memory available will not be used, the copying process is somewhat slower than the equivalent marking phase of a mark-and-sweep collector, and it must be always be possible to tell whether or not a value is a pointer. These

problems are not necessarily serious. Virtual address space may be wasted as long as the paging rates are not too high, and this algorithm makes **cdr**-coding in LISP very attractive. Use of indirect pointers as "handles" or in an "object table" simplifies the copying phase by removing forwarding, and also permits garbage collection with imprecise information because no program values need to be changed to reflect objects' new locations after a collection. This does, however, require an extra indirection for every object reference, but the trade-off was considered acceptable in at least one system [BS83].

2.5 Serial real time compacting garbage collection [Bak78]

Henry Baker's garbage collector is based on the copy-and-compact allocator described above, but the garbage collector's operation is interleaved with program execution. This is done in such a way that it is never necessary to indefinitely postpone the program's execution to collect garbage. Furthermore (or in other words) the time to execute each primitive list operation (**cons**, **car**, **cdr**, **rplaca**, **rplacd**, **atom**, **eq**) and its associated incremental garbage collection is bounded by a small constant; thus the term "real-time."

Since garbage collection occurs incrementally, both spaces will contain active nodes. However, the primitive operations are modified to guarantee that the program only sees nodes in tospace. The operations that need to be changed to preserve this illusion are **car** and **cdr**; **cons** allocates nodes in the tospace, thus satisfying the requirement. The result of **rplaca** (or **rplacd**) is one its arguments; if those are in tospace, then so is the result. **atom** and **eq**, of course, do not return pointers. The pointers returned by **car** and **cdr** are checked to see that they are in the tospace; if they are not, then the nodes in fromspace to which they point are moved into tospace and their new addresses returned instead.

This alone is not enough to guarantee that garbage is collected, because an arbitrary number of **cons**es may take place without any intervening **car**s or **cdr**s. Therefore, each time a **cons** is performed the scan pointer is advanced by a few (say k) nodes. **cons** nodes are allocated from the other end of the space to prevent the garbage collector from (uselessly) scanning nodes whose edges already point into the tospace.

The parameter k determines a bound on what fraction of the tospace may be reachable. Just after a flip (after the other space has filled up) the tospace contains only the root nodes and one **cons** node (located at the low end of tospace); the roots are necessary to discover all reachable nodes, and the **cons**

node is the one that caused the garbage collection (it cannot be located at the high end of the tospace, because its children are not necessarily in the tospace yet). After some number of **cons** operations, say M, this tospace fills up. At this point no nodes in the tospace may contain pointers into the fromspace, because the fromspace is about to be recycled. At each **cons** operation k nodes were scanned, so a total of Mk nodes are guaranteed to contain pointers into the tospace. That is, the old fromspace may contain at most Mk reachable nodes if it is to be recycled. The spaces contain $M+MK$ nodes each (new **cons** nodes plus surviving nodes), so the fraction that may be reachable is $M/(M+Mk)$, or $1/(1+k)$.

Baker also describes extensions to his algorithm that handle array-like objects, a separate user stack, and that produce a linearized version of the graph.

2.6 Lifetime-dependent garbage collection [LH83]

Lieberman and Hewitt describe a garbage collector based on Baker's incremental garbage collector. They introduce several modifications based on the observed and believed behavior of LISP programs. The most important property is that recently allocated objects have short lifetimes. The other property is that most pointers are directed backward in time; that is, objects point to older objects more often than they point to newer objects.

This (design for a) garbage collector is similar to Baker's collector, but the address space is divided into *many* allocation regions instead of just two. At any time, one of these regions is "active", corresponding to Baker's tospace region. Storage for new nodes is allocated out of the current active region. Regions are reclaimed through a process of *condemning* and *scavenging*. Condemning a region is a "declaration of intent" to reuse that region; this cannot occur until scavenging has *evacuated* all active nodes from the region. To scavenge a condemned region, all other regions that might contain pointers into it are scanned; when such a pointer is found, the pointer's target is evacuted from the condemned region to the active region and a forwarding address is stored in the target's old location.

So far, this doesn't seem like a very good garbage collection technique; it won't be very fast if the entire graph must be scanned to reclaim one small region, and it won't be very effective if it can only collect garbage subgraphs that lie entirely within a region (if a garbage path starts in region A, passes into region B, and then on to region C, nodes on the path that lie in C

won't be collected until A, B, and C have been collected *in that order*). By exploiting properties of typical programs, however, Lieberman and Hewitt produce a very effective garbage collector.

Most pointers are directed from young objects to older objects, and the youngest objects are expected to be active for the shortest time. This collector assigns generation and version numbers to the regions to keep track of the ages of objects that they contain and the number of collections that objects within them have survived. Pointers from old regions into younger regions are treated specially. To refer to an object in a younger region, a pointer must point indirectly through the younger region's *entry table*. This makes it easy to find all objects in the younger region referenced from older regions, and makes it easy to relocate objects without updating pointers in older regions. Since only a few pointers from old objects to young objects are expected this should not consume too much space or time. To scavenge a condemned region, it is only necessary to scan the entry table (for pointers from older regions) and younger regions. This means that reclamation of a young region is faster than reclamation of an old region, and (compared to garbage collection in general) quite fast. This is good, because young regions are expected to contain a higher proportion of garbage than old regions and will be reclaimed often. The generation numbers may also be used to arrange collections in a more productive order; collecting younger regions first should eliminate many pointers into older regions, allowing more productive collection of those regions.

Collection of cyclic subgraphs spanning regions is still difficult. They are collected in one of two ways; over time, regions may be combined as they grow smaller, and the cycles will then be contained in one region and become collectable. It is also possible to reclaim several regions at once, and this will collect garbage cycles contained in those regions. This seems expensive, but it is no worse than one flip of Baker's collector, and should occur much less frequently. Reclamation and inheritance of entry tables also presents a problem; when a region is reclaimed, its entry table cannot be recycled, and it is not possible to immediately detect unused pointers in the entry table–if it were, the entry table would be unnecessary. As objects are relocated the entry table's pointers are changed to point to their new locations. Presumably the new region will inherit the old region's entry table and continue to use it. Lieberman and Hewitt suggest that reclamation of entry tables is less critical than reclamation of regions because the entry tables should be much smaller than the regions.

Lieberman and Hewitt briefly describe several ways for recycling slots in entries tables and entry tables themselves. Entry tables should be small relative active memory, so they may be collected more sluggishly and at somewhat higher cost.

Note that this method of partitioning memory up into regions may be combined with other styles of garbage collection; storage in regions may be reclaimed through the use of mark-and-sweep collection, for example. This method also bears some resemblance to proposals for distributed garbage collectors, in which each region corresponds to a single node's memory and the entry tables correspond to tables of off-node references. Although it contains several loose ends, this is a very interesting paper.

2.7 Garbage collection in the Symbolics 3600 [Moo84] [Moo85]

The garbage collector in the Symbolics 3600 is similar to both the Baker incremental collector and the Liebermann-Hewitt lifetime-dependent garbage collector. Objects are grouped according to their "age", but other aspects of the collector have been simplified to allow an affordable hardware assist. The result appears to be an excellent compromise.

The collection algorithm is based on Baker's incremental copy-and-compact collector. During a collection (interleaved with normal program execution) three spaces are used; *oldspace*, *copyspace*, and *newspace*. Instead of allocating memory from both ends of a tospace, objects evacuated from oldspace are copied to copyspace, and objects created by the program are allocated from newspace[3]. The copying algorithm is based on Cheney's non-recursive algorithm, but has been modified to be "approximately depth-first". The new algorithm maintains an additional scan pointer into the partially filled page to which objects from oldspace will be copied. Before advancing the normal scan pointer, the garbage collector examines objects on the last page of copyspace for references to objects in oldspace. These objects will be forwarded to the last (same) page in copyspace, leading to improved locality.

Tagged memory allows the hardware to distinguish pointers from other values. This simplifies the garbage collector's task because it need not use type maps to trace through data structures and in fact allows it to completely

[3] I see no fundamental difference between these and Baker's two spaces, but I will use Moon's terminology.

ignore object boundaries. This simplifies the code generated to run on the 3600 because it need not maintain object's types for the garbage collector. Memory is divided into fairly large *quanta* to allow the hardware to easily determine the target space of a pointer; all objects in the same quantum are in the same space (though a space may include many quanta).

Tagged memory also permits an important hardware assist; in the Baker collector, **car**, **cdr**, and other object dissecting operations must check their results to be sure that they do not return pointers to fromspace. Doing this in software is too slow to be practical; even implementing this in microcode will significantly degrade the machine's performance. The solution used is a hardware "barrier" on the memory bus. Whenever the barrier hardware sees a pointer to oldspace in a word read from main memory it initiates a *transport trap*, and software forwards the object and updates the pointer as necessary (and retries the trapped instruction)[4].

The garbage collector on the 3600 classifies objects according to their expected lifetimes. *Static* objects are treated as permanent unless a garbage collection is explicitly requested. *Ephemeral* objects are recently created, and are expected to quickly become garbage. The garbage collector concentrates its efforts on these objects. *Dynamic* objects are expected to have intermediate lifetime, and are collected less aggressively than ephemeral objects.

As seen above in the Lieberman-Hewitt paper, frequent collection of small areas (the ephemeral objects, for example) is not practical unless it is possible to avoid scanning all of memory for pointers into those areas. Their collector maintains indirection tables for pointers from older areas; the 3600 uses a simpler approach and maintains a bitmap called the GCPT (Garbage Collector Page Tags) identifying pages in physical memory that might contain pointers to ephemeral objects. To recycle an area full of ephemeral objects the collector scans those pages for references and treats any found there as root pointers. Again, this simpler scheme allows a hardware implementation. The barrier hardware monitors all writes to main memory, setting a page's bit in the GCPT whenever the address of an ephemeral object is written into it (again, this is determined by the quantum containing the address). References to ephemeral objects from swapped-out pages are maintained by software in a separate sparse table called the ESRT (Ephemeral Space Reference Table). To simplify the hardware implementing the GCPT, pages

[4] I am somewhat amused to see a LISP-oriented architecture turning the "Von Neumann Bottleneck" to its own purposes.

remain in the GCPT unless the garbage collector discovers (in the course of an ephemeral garbage collection) that there are no pointers to ephemeral objects on that page. The cost of scanning a page that has been swapped to disk is much higher, however, so before pages are swapped to disk they are scanned for references to ephemeral objects. This is not terribly costly, because a page need only be scanned if its GCPT bit is set. Moon's paper describes other situations in which pages need not be added to the ESRT.

The actual system is somewhat more complicated than this; ephemeral objects are divided into *levels* so that "less ephemeral" objects (those that survive a few collections) are collected less frequently. It seems as if synchronization of garbage collection and access to memory should be very complex; this is avoided by having very little synchronization, and occassionally scanning a page several times searching for references.

The division of collectable objects into ephemeral and dynamic classes significantly improves the performance of the system. In the two benchmarks cited by Moon the collection of ephemeral objects cuts the elapsed time approximately in half and reduces page faulting by factors of 7 and 50.

The 3600 also employs **cdr**-coding to compress the representation of linear lists. Given expected LISP list structure and approximately depth-first copying, this should significantly reduce memory usage and improve locality. This seems incompatible with the claim that tagged pointers allow the collector to ignore object structure, but perhaps **cons** cells are treated specially.

I feel that these papers are an impressive argument for hardware assist in a LISP implementation. Little extra hardware is required (Moon estimates 10% for 4 tag bits and 2 or 3% for the hardware barrier), but that which is added obviates software maintenance of tag bits and keeps fundamental operations (**car**, **cdr** and **eq**) simple.

2.8 Generation scavenging [Ung84]

David Ungar describes a garbage collector used in the Berkeley Smalltalk that (to quote)

- limits pause times to a fraction of a second,
- requires no hardware support,
- meshes well with virtual memory, and

- uses less than 2% of the CPU time in one Smalltalk system. This is less than a third the time of the next best algorithm.

Generation scavenging achieves this high performance by using the idea of segregating objects into "generations" proposed by Lieberman and Hewitt, but performs many small garbage collections instead of one large garbage collection or continuously reclaiming storage. Continuous reclamation guarantees "real-time" performance (bounded response time), but imposes a large constant overhead. Large garbage collections are too disruptive. The small garbage collections are performed frequently, but are so fast that the response is acceptable for interactive use[5].

Memory is divided into four areas: the NewSpace, PastSurvivorSpace, FutureSurvivorSpace, and OldSpace. NewSpace holds newly created objects. PastSurviviorSpace contains objects that have survived previous scavenges. FutureSurvivorSpace is empty during program execution. During scavenging objects are copied from NewSpace and PastSurvivorSpace into FutureSurvivorSpace, and the two Survivor spaces are swapped. Objects that survive a number of scavenge operations are "tenured" and moved into OldSpace, where they become effectively permanent. Stores into OldSpace of references to the NewSpace or SurvivorSpace cause the referencing objects to be placed into the *remembered set*. Along with machine registers, the *remembered set* is used as the root of a scavenging operation. During scavenging objects are removed from the *remembered set* when they are discovered to no longer point to one of the new object spaces. Typically sizes for the various spaces are:

NewSpace 140 kilobytes of main memory

Survivor spaces 28 kilobytes each of main memory

Old space 940 kilobytes of demand paged memory

In the benchmarks given in the paper, scavenging operations occurred about once every 16 seconds, and required an average of 160 milliseconds to complete. The maximium pause was 330 milliseconds, the minimum 90 milliseconds. The Old space is collected once every 3 to 8 hours by a mark-and-sweep collection that takes 5 minutes.

[5]Several users of this system have verified this claim.

This algorithm does require that stores into old objects place a reference to the old object into the *remembered set*, but that is apparently the only cooperation required by the program or the hardware. Support for this checking is included in the architecture of SOAR [Pat83].

2.9 Collection of cyclic structures with reference counts in Scheme [FW79]

Friedman and Wise note that in SCHEME circular reference structures may only be created in certain situations, and that this allows the use of reference counts to reclaim these circular reference structures.

In SCHEME (a dialect of LISP), it is not possible for users to create circularly linked storage in memory because there are no destructive **rplaca** and **rplacd** operators. However, the implementation of recursive functions implicitly creates circular structures in memory. Their interpreter represents an environment (a map from names to objects) as a linked list of **cons** nodes. The **car** of each node is the name and the **cdr** is the object. Function objects are represented with two **cons** nodes; the first contains a tag (**funarg** or β**funarg**) and a pointer to the second, which contains a pointer to the definition of the function and the environment in which it is interpreted. To implement (mutually) recursive functions, SCHEME provides an operator **labels** that takes as input (implicitly) the environment and (explicitly) a number of function names and definitions. Executing a **labels** construct extends the environment with bindings for the functions, but the environment associated with each function is the extended environment, not the original environment. This is the only way to create cyclic reference paths in SCHEME. (**car** environment yields a binding, **cdr** binding yields a tagged object, **cdr** object yields a closure, **cdr** closure yields the environment again.) To identify the cyclic references, the closures created by **labels** are tagged with β**funarg** instead of **funarg**. This is illustrated by figure 3, in which 'G' is the original environment and 'E' is its extension by a pair of mutually recursive functions.

To allow garbage collection of these cycles with simple reference counting, Friedman and Wise impose restrictions to guarantee that when the root of a cycle is discarded, all members of the cycle are also discarded. To guarantee that the cycle dies when the head dies it is necessary that at all times all nodes in the cycle (except the head) are referenced only by other nodes in

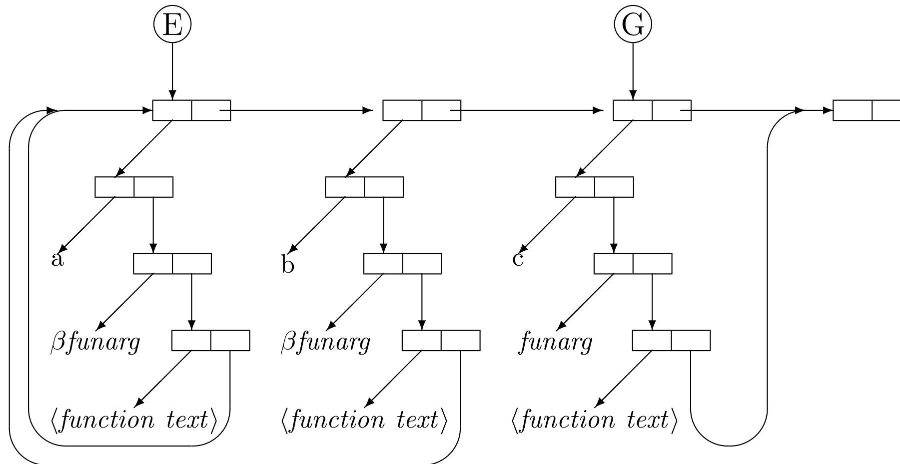

Figure 3: Circular environments in Scheme

the cycle. This is true initially, because **labels** creates a circular structure all at once; no part of the structure existed before executing **labels**, so it cannot be shared. This property is maintained whenever non-head portions of the cycle are used by copying the portions instead of referencing them. Since the closures are the only parts of the cycle that may be referenced individually, it is sufficient to check a closure's tag before referencing it. If the closure is part of a cycle, then the root of the closure and the root of the function and environment pair must be copied. The environment points to the head of the cycle, so no more copying is necessary and no non-head part of the cycle is shared. Finally, cyclic edges must be treated specially during storage reclamation because they will point to freed storage when they are traversed. Again, the tags on the closures allow cyclic edges to be identified and handled correctly.

2.10 Garbage collection in a combinator machine [SCN84]

Stoye et. al. describe several novel methods used in SKIM II, a microcoded processor for evaluation of functional languages. Their garbage collector combines traditional garbage collection with one-bit (hardware-assisted) ref-

erence counting to reduce the overall cost of memory management. The reference bit is stored with pointers to objects instead of with the objects. This is possible because the bit has only two states, shared and not-shared. Before a pointer is copied its bit (and the copied bit) are set to indicate sharing; since there is only one copy of a pointer to an unshared object, it is not necessary to search for other pointers in order to maintain their reference counts. This has a very beneficial effect on locality of reference; conventional reference counting must inspect and update objects in the heap, while this method can inspect and update reference counts without any additional memory references.

Storage is reclaimed whenever a pointer to an unshared object is discarded; the storage to which it points is returned to a free list for reuse. Additional gains are realized by creating special-purpose versions of the primitive instructions for each possible combination of shared and unshared arguments, allowing direct reuse of memory with fewer references to the free list. The results are spectacular—an average of 70 percent of the discarded cells are immediately reclaimed through the use of reference counts.

2.11 Garbage collection in Interlisp [DB76]

In this paper Deutsch and Bobrow describe a garbage collector for Interlisp based on reference counting. Their goal was to avoid long pauses caused by conventional garbage collectors, while also avoiding the high overhead associated with reference counting.

Reference counting is incremental; reclamation of storage is easily interleaved with other computation[6]. However, reference counting incurs significant space and time costs. Each object must have an associated reference field, and every time a pointer to that object is created or destroyed the reference count must be adjusted. This uses additional time and code space, and can cause an otherwise unnecessary page fault if the object and its reference count are not in a program's working set.

Several modifications make reference counting practical. First, reference count adjustments are treated as transactions and stored in a "file". References from variables (activation records) are not counted, and reference counts are not stored with the objects. Instead, reference counts are stored

[6]The time to free a reference-counted object is not bounded, because its destruction may recursively free other objects, as noted by Baker [Bak78]. This problem is solved (in theory) by the use of a transaction file.

by means of hash links [Bob75]. Addresses of objects with no references (except possibly from program variables) are stored in the zero count table (ZCT); addresses and counts for objects with more than one reference are stored in the multi-reference table (MRT); addresses of objects referenced by variables are stored in the variable reference table (VRT). Any object appearing in neither the ZCT nor the MRT has an implicit reference count of one. In a LISP system most objects are expected to have only one reference [CG77], so this is expected to save space and improve locality of reference.

As the program runs the tables become out of date, but a record of the adjustments to (in-heap) reference counts is stored in the transaction file. In this system, computation is briefly interrupted to collect unused storage. All references appearing on the stack are entered into a new VRT. The MRT and ZCT tables are paged in, and the adjustments in the transaction file are processed. When the tables are up to date, all objects whose addresses appear in the ZCT but not in the VRT may be reclaimed.

The various tables are also used to help the auxiliary garbage collector (based on Fenichel and Yochelson's copy-and-compact collector) that reclaims circular storage and compacts memory. The MRT is augmented with a field to store forwarding pointers and multiple references from variables (that otherwise would create entries in the VRT). By storing the forwarding pointers in the MRT, smaller objects may be allocated (which otherwise must be large enough to hold a flag and a forwarding pointer). When an object is copied its forwarding address is recorded only if its original address appears in the MRT; if its address is not in the MRT, then there is only one reference to the object and it can be updated when the object is copied. The single-reference situation is expected to occur fairly often, so this shortcut is probably a win.

A concurrent variation of this algorithm is also described. A co-processor maintains the ZCT and MRT as transactions occur. To collect garbage, a snapshot of the stack is delivered to the co-processor, which removes variable references from a copy of the ZCT. All addresses remaining in this copied table after processing the stack are unreachable, and may be recycled.

This paper is interesting because it views reference count adjustments as transactions and because it uses hash links to store reference count information. It is also a very good illustration of the way a garbage collector may be tailored to a program's expected behavior and adapted to deal with real-world problems (such as paging).

2.12 Garbage collection in Cedar Mesa [Rov85]

This garbage collector is similar to Deutsch and Bobrow's Interlisp garbage collector described above, though it has been modified in several important ways. The multi-reference table is not used; instead, reference counts are stored in objects and updated immediately. References from the stack (activation records) are still uncounted to save time. The implementation of the variable reference table (now called the "found on stack table", or FOSTable) has been refined to save time and table space, at the expense of allowing some retention of inaccessible objects. The auxiliary copying collector has been replaced with a trace-and-sweep collector.

Each collectable object has a header, which contains information about the object's type, size, reference count, and four flags "maybeOnStack", "rcOverflowed", "onZCT", and "finalizationEnabled". The type field allows runtime type-checking and provides an index into a table of type descriptors. Included in a type descriptor is a map locating all collectable pointers (REFS) within an object of that type. Objects, not pointers, are tagged. Reference counts are adjusted as pointers are created and destroyed; if a count falls to zero then the object's address is placed in the ZCT. An object's maybeOnStack flag is set whenever (1) the collector is active and (2) an assignment decrements its reference count. This allows concurrent garbage collection and program execution. When a reference count overflows rcOverflowed is set. Whenever an object is placed in the ZCT onZCT is set to prevent the object from being entered in the table a second time.

Garbage collection is triggered by a number of events; the ZCT may become too large, or the amount of virtual memory available may become too small, or (more usually) the amount of memory allocated since the last collection will exceed some threshold. In any case, to reclaim memory the garbage collector takes a "snapshot" of the stack to build the FOSTable. The computing process resumes, and the collector (concurrently) frees any objects that appear in the ZCT, do not appear in the FOSTable, and have maybeOnStack false.

The Found On Stack Table is implemented as a hash table. To speed up membership tests, each entry in the table contains the logical OR of all REFs hashing to that table address. This is fast, but conservative, since it may incorrectly indicate that a REF appears on the stack. A different hash function is used at each garbage collection to reduce increased retention (constipation!). When building the FOSTable, any value on the stack that

might be a pointer (i.e, contains a valid heap address) is treated as a pointer; again, thorough garbage collection is sacrificed for speed.

The Zero Count Table is implemented as a queue of blocks. Objects are added to the table at the *write pointer*; the collector scans the table from the *read pointer*. Each address scanned by the collector is either freed (if it satisfies the conditions given above) or moved to the end of the ZCT. When the collector finishes with a block, it places the block at the end of the queue. A garbage collection finishes when the read pointer catches up to the write pointer.

The maybeOnStack flag permits concurrent reclamation and mutation. Without it, the following incorrect reclamation (taken from Rovner [Rov85]) can occur: at the start of a collection, X has one reference from the accessible object Y and another from the inaccessible object Z, giving a count of 2. No references to X are on the stack at the beginning of collection, so it is not in the FOSTable. The program moves the reference to X from Y to the stack and NILs out Y's reference (decrementing X's reference count). The collector discovers that Z is inaccessible and decrements X's count to zero, placing X in the ZCT and subsequently (incorrectly) reclaiming it. The conditions for setting the maybeOnStack flag prevent this; if the collector is active and an assignment (NILing out Y's reference to X) decrements an object's reference count, then its maybeOnStack flag is set and it will not be collected. The object is also placed in the ZCT when this occurs so that the collector can find and reset all objects whose maybeOnStack flags are set. The flag may also be reset if the object's reference count is incremented.

The trace-and-sweep collector is straightforward and conventional,and will not be described here.

This paper is interesting to me because it describes an instance of reference-counting in the "real world", and because it illustrates many interesting tricks and compromises used to get acceptable performance. One important aspect of memory management in the Cedar environment is mentioned only in passing; assistance to the collector from the programmer. This also occurs in other systems, but it is more important in Cedar because the primary garbage collector is unable to reclaim cyclic structures. The programmer can assist the garbage collector by replacing pointer values with NIL when they are no longer needed; this can be used to break cycles and reduce the size of the FOSTable, increasing the yield of a given garbage collection. This is not a common activity, but in certain situations it is very important.

2.13 Collection of cyclic structures with reference counts in a combinator machine [Bro85]

Combinator graph reduction [Tur79] generates a large amount of garbage in very predictable ways. Only a few primitives can copy nodes in a graph, and only one can create a cyclic reference where none existed before. Brownbridge proposes a reference-counting technique that also reclaims circular structures by taking advantage of properties of combinator graph reduction.

Pointers are tagged with a bit to determine if they are cyclic ("weak") or acyclic ("strong"), and each object maintains one reference count for the number of weak in-pointers and one for the number of strong in-pointers. Normally (in the absence of cycles) reference counting garbage collection occurs in the usual way. Whenever a node's strong reference count falls to zero all of its children are deferenced and its storage is reclaimed. When there are cycles (when the weak reference count is not zero), the algorithm is more complicated, as will be described below.

Strong and weak pointers are defined by two rules:

1. The strong pointers together with the objects in use form a directed acyclic graph in which every object is reachable from a distinguished root object.

2. Pointers which are not strong are weak.

Notice that a given graph may have more than one assignment of strength to its edges that is consistent with the above rules. Cyclic reference counting works because the creation of weak pointers is easily detected in a combinator machine, and there is an algorithm that correctly collects garbage and maintains a valid pointer strength assignment. The two reference counts are called *WRefC* (weak reference count) and *SRefC* (strong reference count).

Combinator graph reduction creates weak pointers only through application of the Y combinator or by copying an existing weak pointer. The primitive graph operations are NEW, COPY, and DELETE, described below. Note that not all graphs generated by these operations obey rules 1 and 2, but that any combinator graph representing a functional program will.

NEW(R) Create a strong pointer from R to some free object U, setting $SRefC(U) = 1$ and $WRefC(U) = 0$.

COPY($R, \langle S, T\rangle, c$) Create a pointer from R to T. If 'c' is true then assume that $\langle R, T\rangle$ will be cyclic and add one to $WRefC(T)$ and mark $\langle R, T\rangle$ as weak; otherwise, add one to $SRefC(T)$ and mark $\langle R, T\rangle$ as strong. Correct operation depends upon supplying the correct value of 'c', according to rules 1 and 2. For a combinator graph reducer, 'c' is true when applying Y or when $\langle S, T\rangle$ is a weak pointer.

DELETE($\langle R, S\rangle$) There are several cases for deletion:

1. The pointer $\langle R, S\rangle$ is weak.
   Decrement $WrefC(S)$.
   By rule 1, S is still reachable by strong pointers.
2. The pointer $\langle R, S\rangle$ is strong and $WRefC(S) = 0$.
   Decrement $SRefC(S)$. If it is then zero, DELETE all outgoing edges from S and make S available for reuse.
3. The pointer $\langle R, S\rangle$ is strong, $WRefC(S) > 0$, $SRefC(S) > 1$.
   Decrement $SRefC(S)$.
4. The pointer $\langle R, S\rangle$ is strong, $WRefC(S) > 0$, $SRefC(S) = 1$.
   The last strong pointer to S disappears. Recursively search objects pointed to by S, attempting to find an external pointer and adjusting strength of pointers to obey rules 1 and 2.
   (a) Set $SRefC(S) = 0$ and remove $\langle R, S\rangle$.
   (b) Convert weak pointers to S into strong pointers (there is a trick that makes this a simple operation) and swap $WRefC$ and $SRefC$.
   (c) For T in $CHILDREN(S)$, $SUICIDE(S, \langle S, T\rangle)$.
   (d) If $SrefC(S)$ is zero, DELETE all outgoing edges from S and make S available for reuse.

The *SUICIDE* operation recursively visits nodes in the subgraph, reassigning strength to edges to obey rules one and two. If (after this reassignment) S has a strong in-edge, then S is still reachable from the root. Otherwise, S is freed and edges to its children deleted.

$SUICIDE(start, \langle R, S\rangle)$
`if` $\langle R, S\rangle$ *is strong* `then`
`if` $S = start$ `then` *make* $\langle R, S\rangle$ *weak*
`else if` $SRefC(S) > 1$ `then` *make* $\langle R, S\rangle$ *weak*
`else for` $T \in CHILDREN(S)$ `do` $SUICIDE(start, \langle S, T\rangle)$

Strength and weakness of pointers is determined by comparing a bit stored in the pointer with a bit stored in the pointee. If they agree, then the pointer is strong; otherwise, the pointer is weak. This is the "trick" that makes converting weak pointers to strong pointers a simple operation; the strength of all pointers to an object is inverted by complementing the bit in the object.

One drawback to this algorithm is that one must always know whether or not an added edge creates a cycle. This is no problem on the combinator machines for which Brownbridge intends his collector. It may also be practical for LISP, because creation of cyclic structure only occurs with the execution of RPLACA or RPLACD; perhaps these operations are used rarely enough that checking for creation of cycles will not be too expensive.

3 Measurements of storage allocating behavior

The garbage collection algorithms described above either exploit or require certain behavior from their client programs. Of these, the most important are:

1. Recently allocated objects usually have short lifetimes.
2. Most objects are referenced by only one pointer.
3. LISP lists are frequently linear in structure.
4. Many LISP **cdr**s contain the value NIL.
5. Cycles are relatively rare in garbage graphs.
6. Object sizes are not uniformly distributed.

These assumptions are supported both by empirical studies and by the improved performance of garbage collectors that exploit them.

In particular, Clark's work [Cla79, CG77, BC79] explores the locality of list structure in LISP. He finds that most lists have linear structure in the **cdr** direction, and that **cdr** linearization persists for some time. In the programs that Clark examines, **rplaca** and **rplacd** occur more than half as often as **cons** but usually replace an object pointer with one to NIL or a pointer to NIL with a pointer to an object.

Work by Batson and Brundage [BB77] and by Nielsen [Nie77] supports the assumption that objects' sizes are not uniformly distributed. In a study of memory allocator performance in computer simulation, Nielsen found that the best algorithms maintained a separate free list for each object size requested; that is, if a program requests a particular amount of memory once, it is likely to request it again. Batson and Brundage performed a study of "segment lifetimes" for Algol 60 programs, and found that segments usually had very short lifetimes, and were typically quite small. They present cumulative distributions for segment sizes that show abrupt jumps; that is, a few segment sizes occur very frequently.

4 Conclusions

Though this compilation includes only a small sample of the many garbage collectors in existence, it demonstrates several things. Garbage collectors rarely solve (efficiently) the general problem; most depend upon well-known properties of garbage generation and program behavior. Others impose restrictions on program behavior to guarantee correct and efficient performance. Many variables affect the cost of garbage collection. The use of virtual memory increases the importance of locality of reference many-fold, while it eases the hard constraint on address space imposed by its absence. Hardware and microcode assists to the garbage collector permit algorithms that would be prohibitively expensive if implemented by software, but are of little use when unavailable. They also make bad (and good) design decisions permanent. Different languages make different demands on garbage collectors, either because of hard rules or because of popular programming styles. Garbage collection even varies significantly from program to program. Given this, it is clear that there is no "best garbage collection algorithm" for all problems.

Other factors not mentioned above also affect the quality of garbage collection. Various optimizations may reduce the amount of garbage, especially short-lived garbage, that a program generates. This is clearly demonstrated by combinator compilation techniques for the G-machine [Kie85]; by avoiding the construction of application trees, many of the objects that would be reclaimed by (for example) one-bit reference counting in Stoye's collector above are never allocated. Many LISP compilers perform some analysis to avoid

heap allocation of activation records [Ste77, BGS82][7], reducing the number of short-lived objects. However, the LISP compiler for the Symbolics 3600 performs these optimizations [Moo85] and still reaps a substantial benefit from its special treatment of newly created objects.

Programmers can also help or hinder a garbage collector. Object references can be explicitly removed by replacing them with pointers to NIL. In a reference counted system this may break a cycle; in a more general system this is still useful if the last reference to an object is explicitly removed before a garbage collection. Programmers may affect garbage collection in other ways. Bob Shaw [Sha86] has noticed that objects seem to "stratify", with lifetimes apparently corresponding to lifetimes of modules. If this hypothesis is true and a collector is designed to take advantage of this additional property of programs, then programming in a modular style should lead to more effective garbage collection. Programmers will then use that style for efficiency purposes, causing objects to become even more stratified. This illustrates the possiblity for self-fulfilling prophecies in garbage collector design, and points out the potential influence of changing programming styles (in addition to the influence of changing hardware, languages, collection algorithms, and compilers).

References

[Arn72] S. Arnborg. Storage administration in a virtual memory SIMULA system. *BIT*, 12:125–141, 1972.

[Bak78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[BB77] A. P. Batson and R. E. Brundage. Segment sizes and lifetimes in ALGOL 60 programs. *Communications of the ACM*, 20(1):36–44, January 1977.

[BC79] Daniel G. Bobrow and Douglas W. Clark. Compact encodings of list structure. *ACM Transactions on Programming Languages and Systems*, 1(2):266–286, October 1979.

[7] and many many others.

[BGS82] Rodney A. Brooks, Richard P. Gabriel, and Guy L. Steele Jr. An optimizing compiler for lexically scoped LISP. In *SIGPLAN Symposium on Compiler Construction*, pages 261–275, 1982.

[Bob75] D. G. Bobrow. A note on hash linking. *Communications of the ACM*, 18(7):413–415, July 1975.

[Bro85] D. R. Brownbridge. Cyclic reference counting for combinator machines. In *Functional Programming Languages and Computer Architecture*, pages 273–288, 1985.

[BS83] Stoney Ballard and Stephen Shirron. The design and implementation of VAX/Smalltalk-80. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, pages 127–150. Addison-Wesley, 1983.

[CG77] D. W. Clark and C. C. Green. An empirical study of list structure in LISP. *Communications of the ACM*, 20(2):78–87, February 1977.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[Cla79] Douglas W. Clark. Measurements of dynamic list structure use in Lisp. *IEEE Transactions on Software Engineering*, 5(1):51–59, January 1979.

[Coh81] Jacques Cohen. Garbage collection of linked data structures. *Computing Surveys*, 13(3):341–367, September 1981.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[DR81] John Darlington and Mike Reeve. Alice: A multi-processor reduction machine for the parallel evaluation of applicative languages.

In *Functional Programming Languages and Computer Architecture*, pages 65–75, 1981.

[FW79] D. P. Friedman and D.S. Wise. Reference counting can manage the circular invironments[sic] of mutual recursion. *Information Processing Letters*, 8(1):41–44, 1979.

[FY69] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[GP81] Dale H. Grit and Rex L. Page. Deleting irrelevant tasks in an expression-oriented multiprocessor system. *ACM Transactions on Programming Languages and Systems*, 3(1):49–59, January 1981.

[Kie85] Richard Kieburtz. The G Machine: A fast, graph-reduction evaluator. In *Functional Programming Languages and Computer Architecture*, pages 400–413. Springer-Verlag, 1985.

[Knu68] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1968.

[Kra83] Glenn Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1983.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[Moo84] David Moon. Garbage collection in a large Lisp system. In *SIGPLAN Symposium on LISP and Functional Programming*, pages 235–246, 1984.

[Moo85] David A. Moon. Architecture of the Symbolics 3600. In *International Symposium on Computer Architecture*, pages 76–83, 1985.

[Nie77] Norman R. Nielsen. Dynamic memory allocation in computer simulation. *Communications of the ACM*, 20(11):864–873, November 1977.

[Pat83] David A. Patterson. Smalltalk on a risc: Architectural investigations. Technical Report CS292R, Computer Science Division, University of California, Berkeley, April 1983.

[Rov85] Paul Rovner. On adding garbage collection and runtime types to a strongly-typed, statically checked, concurrent language. Technical Report CSL-84-7, Xerox Palo Alto Research Center, 1985.

[RT78] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Bell System Technical Journal*, 57(6):1905–1929, 1978.

[SCN84] W. R. Stoye, T. J. W. Clarke, and A. C. Norman. Some practical methods for rapid combinator reduction. In *SIGPLAN Symposium on LISP and Functional Programming*, pages 159–166, 1984.

[Sha86] Bob Shaw. conversation at Stanford University, April 1986.

[Ste77] Guy Lewis Steele Jr. Compiler optimization based on viewing LAMBDA as RENAME plus GOTO. Master's thesis, Massachusetts Institute of Technology, 1977.

[Tur79] David Turner. A new implementation technique for applicative languages. *Software, Practice and Experience*, 9, 1979.

[Ung84] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, 1984.