A Real-Time Garbage Collector Based on the Lifetimes of Objects

HENRY LIEBERMAN AND CARL HEWITT *MIT Artificial Intelligence Laboratory*

Henry Lieberman is a Research Scientist at the MIT Artificial Intelligence Laboratory. His research interests include artificial intelligence, programming environments, computer systems for education and beginning users, and computer graphics. Arpa Network address: HENRY@MIT-AI or HENRY@MIT-ML.

Carl Hewitt is Professor of Computer Science at MIT. His research interests include message-passing semantics, open systems, organization theory, artificial intelligence, and scientific communities. Arpa Network address: CARL@MIT-AI.

ABSTRACT: In previous heap storage systems, the cost of creating objects and garbage collection is independent of the lifetime of the object. Since objects with short lifetimes account for a large portion of storage use, it is worth optimizing a garbage collector to reclaim storage for these objects more quickly. The garbage collector should spend proportionately less effort reclaiming objects with longer lifetimes. We present a garbage collection algorithm that (1) makes storage for short-lived objects cheaper than storage for long-lived objects, (2) that operates in real-time--object creation and access times are bounded, (3) increases locality of reference, for better virtual memory performance, (4) works well with multiple processors and a large address space.

1. INTRODUCTION

> *One day a student came to Moon and said, "I understand how to make a better garbage collector. We must keep a reference count of the pointers to each cons." Moon patiently told the student the following story:*
>
> *"One day a student came to Moon and said, `I understand how to make a better garbage collector . . . '"*
>
> *Danny Hillis*

In Lisp and similar systems with dynamically allocated storage, the lifetimes of objects can vary widely. Some objects are used to store relatively permanent data and remain in the system for long periods of time. Others are used by programs to store temporary results: these objects are created, used for a short while, then are no longer needed. These short-lived objects account for a large proportion of storage use.

The traditional garbage collection algorithms [1] have the defect that storage for objects with short lifetimes is just as costly as storage for objects with long lifetimes. When an object becomes inaccessible, the time needed to recover it is independent of the lifetime of the object. Our observations of large Lisp programs indicate that there is much to be gained in performance by optimizing the special case of recovering storage for short-lived objects.

In this paper, we propose a new garbage collection algorithm that takes account of the lifetimes of objects to improve efficiency. To use an analogy, our scheme can be thought of as "renting" memory space, where the storage management cost for an object is proportional to the time during which the object is used. Traditional methods are more like "buying" memory space, since the cost for an object is paid once and is always the same, regardless of how much the object is used. When large numbers of objects are used, although each object may be used only for a short period of time, the renting strategy will cost less overall than the buying strategy. Our garbage collector should also turn out to be more efficient on long-lived objects, since the garbage collector will spend less effort continually considering them as candidates for reclamation.

We were led to work on the garbage collection problem because of the performance needs of applications in artificial intelligence (Al). The performance of the new generation of object-oriented, message-passing systems, which we believe to be the best vehicle for Al applications [15, 17, 20], will rely increasingly on the efficiency of storage for short-lived objects. Programs that do a lot of internal "thinking" will need lots of short-lived objects as "thinking material" before they commit themselves to decisions. These programs will need to construct hypothetical worlds which may eventually be thrown away after their purpose has been served.

Some systems use *reference counts* instead of garbage collection, primarily because a reference count system can reclaim short-lived objects more quickly. Short-lived objects are reusable as soon as they become inaccessible, that is, when their reference counts reach zero. However, reference count systems have formidable problems of their own. Reference counting cannot reclaim circular structures, as our introductory story points out. Circular structures are becoming an increasingly important programming technique in sophisticated Al applications. Making sure reference counts are always updated when necessary and kept consistent is sometimes tricky. Maintaining the reference counts often consumes a considerable percentage of the total processor time. If a large proportion of the objects which are created are eventually lost, garbage collectors which *trace the accessible objects* will be preferred to reference counts, which *trace the inaccessible objects*. Some have also proposed more complicated systems which combine reference counts with garbage collection [10, 24].

Our garbage collector incorporates a simple extension to a garbage collection algorithm devised by Baker [4]. Baker's garbage collector performs garbage collection in *real time*— the elementary object creation and access operations take time which is bounded by a constant, regardless of the size of the memory. We would also like a garbage collection algorithm that will work well on machines with a very large address space [6].

We believe these properties will be essential in making garbage collection practical on the next generation of computers. The suggestions described in this paper are currently being explored for implementation on the Lisp Machine [13, 25], a high performance personal computer currently in use at MIT, and on the Apiary [16], a proposed multiprocessor machine for object-oriented programming.

2. A REVIEW OF BAKER'S ALGORITHM

Baker proposes the address space be divided into *fromspace* and *tospace*. Objects are created (by operations like Lisp's CONS) from successive memory locations in tospace. The garbage collection process traces accessible objects, incrementally *evacuating* objects, moving them from fromspace to tospace. When no more accessible objects remain in fromspace, its memory can be reused. An operation called a *flip* occurs, where the tospace becomes the fromspace and vice versa.

When an object is evacuated from fromspace to tospace, an *invisible pointer* (or *forwarding pointer*) is left in the fromspace memory cell pointing at its new location in tospace. To make an analogy with mail, an invisible pointer is like a *forwarding address*. When a person moves, the post office sends mail destined for the old address to the new address instead. In addition, the sender should be informed that mail should be sent to the new address from then on.

When a fromspace cell containing an invisible pointer is referenced, the link to tospace is followed and the tospace object is returned. Furthermore, the original reference is altered to point to the tospace object. On a microcoded machine, this occurs in microcode and is completely transparent to the user's program.

The operations that access components of an object (like CAR and CDR in Lisp) check the address to make sure the address is in tospace. Any object located in fromspace is evacuated to tospace, and the reference updated.

When an object is first evacuated to tospace, one of its components can point back to fromspace. We would like to remove all pointers back to fromspace so that fromspace's memory can be recycled. Whenever a pointer from tospace to fromspace is found, we can remove the pointer by evacuating the fromspace object, moving it to tospace, and updating the tospace pointer to the newly evacuated object in tospace. This process is called *scavenging*.

Tospace is divided into two areas: the *creation* area where newly created objects appear, and the *evacuation* area, which contains objects evacuated from fromspace. (In Baker's scheme, the creation area was allocated from the highest location in tospace downward and the evacuation area was allocated from the bottom upward.)

Scavenging is a process that linearly scans the evacuation area of tospace; if a component of an object points to fromspace, the fromspace object is evacuated to tospace (appended to the evacuation area). Like the mark phase of traditional garbage collectors, scavenging touches all accessible objects. It does so in breadth-first order and does not require a stack.

The *scavenger* process can be interleaved with object creation, evacuating a few fromspace objects to tospace every time an object is created. Since only a small amount of work must be done whenever an object is created or parts of an object are accessed, the garbage collection operates in real time. (See Figure 1.) (A more detailed description of the Baker algorithm appears in Appendix l.)

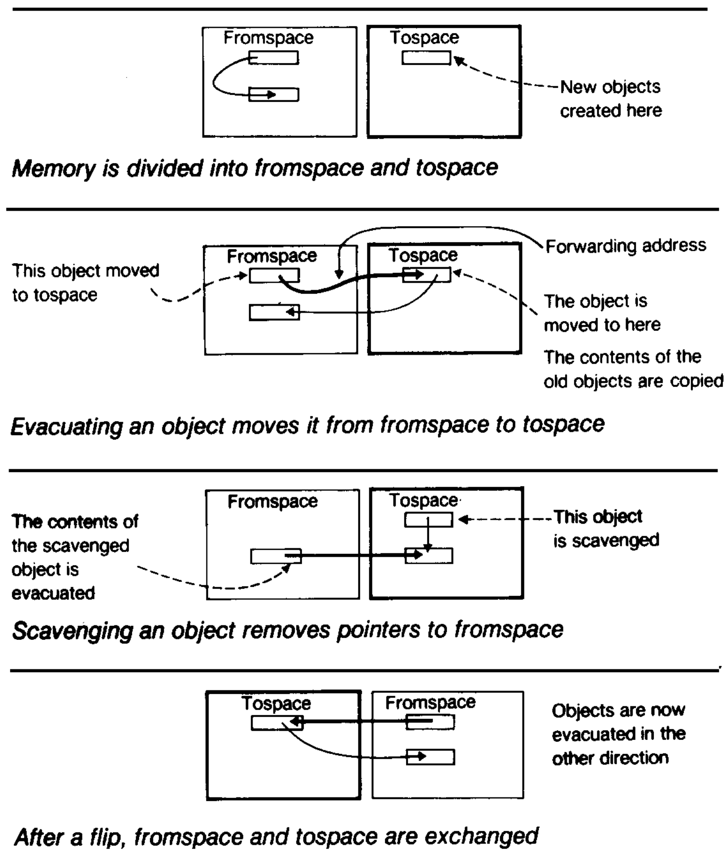

FIGURE 1. Henry Baker's Real-Time Garbage Collector.

3. SMALL REGIONS OF MEMORY CAN REPLACE BAKER'S SPACES

We now present a description of our alternative to Baker's algorithm. (A more detailed, step-by-step description of this procedure appears in Appendix II.) We will start with the key concepts behind the algorithm and then discuss special-case modifications, optimizations, and alternative implementations.

For the moment, let us pretend that all references to objects reside in the heap memory. We will consider other sources of object references later. We will retain some of the essential aspects of Baker's algorithm. Garbage collecting a space will involve moving all the accessible objects out of the space, *evacuating* them to another space, then scavenging to remove all pointers pointing into the old space so the memory for the space can be recycled.

Our scheme involves two major improvements to Baker's algorithm. Baker divides the address space into two halves, fromspace and tospace (cutting down the effectively usable address space by a factor of 2). In our scheme, *the address space is allocated in small regions.*

A region is a small set of pages of memory (not necessarily contiguous). We will not commit ourselves to a particular size for regions, but regions should be small compared to the address space. Of course, allocating address space in regions opens up the possibility that we will waste some space because partially filled regions will occupy memory. However, it should be possible to choose the region size large enough to minimize the effects of fragmentation of regions. The machine should be able to quickly tell, for a given page, to what region it belongs.

We will use these fine divisions of the address space to *vary the rate of garbage collection for each region*, according to the age of the region. Recently created regions will contain high percentages of garbage and will be garbage collected frequently. Older regions will contain relatively permanent data and will be garbage collected very seldom.

New objects are created from storage allocated in *creation* regions. At any time, there is a *current* creation region, in which operations like CONS can create new objects. When the current creation region is filled, a new one is allocated.

We introduce a mechanism to keep track of how recent each region is, so we can distinguish between data likely to be relatively temporary or more permanent. Regions are organized into *generations*. The system keeps track of a *current generation number*; when a creation region is born, it is given the current generation number. The current generation number is periodically incremented.

The process of garbage collecting a particular region is initiated by *condemning* the region. We will call objects *obsolete* if they reside in a region that has been condemned. Condemning a region announces our intention to move all the accessible objects out of the region so that we can recycle the memory for that region. When we condemn a region, we create new regions to hold the objects evacuated out of a condemned region. Each of these *evacuation regions* inherits the same generation number as the condemned region but is assigned a *version number* one higher. The version number of a region counts how many times regions of that generation have been condemned.

Objects are evacuated in the same way as in the original Baker algorithm. We allocate space for a new object in the evacuation region and copy the contents of the old object into the new space. An invisible pointer is left in the old memory cell pointing to the new object. If we encounter any reference to a cell containing an invisible pointer, the reference is updated to point to the new object. (See Figure 2.)

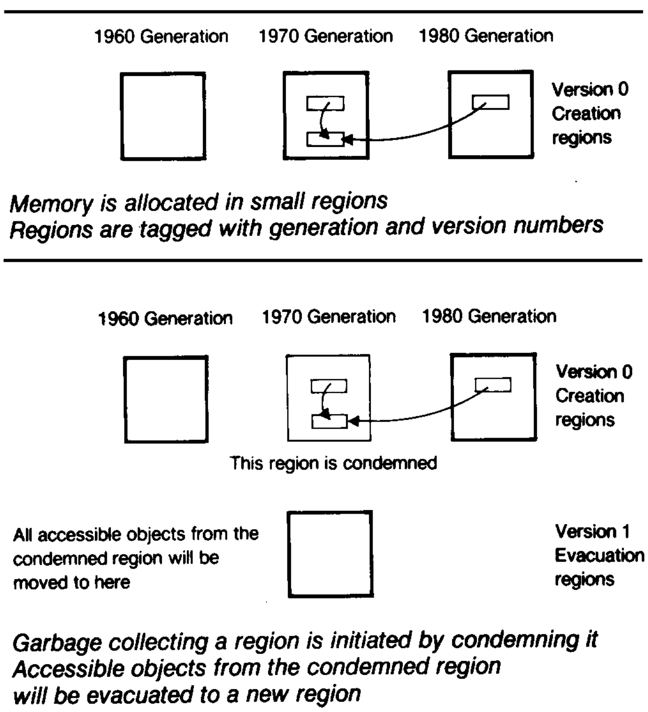

FIGURE 2. Our Real-Time Garbage Collector.

The correspondence between our algorithm and Baker's is that obsolete areas of memory play the role of fromspace; everything else in memory is like Baker's tospace. Condemning a region is like Baker's flip operation on a much smaller scale.

4. SCAVENGING IS REDUCED BY GROUPING POINTERS FROM OLDER TO NEWER OBJECTS

In order to release memory for a condemned region, we have to make sure that no pointers from outside the condemned region point to it. This is done, as in Baker's algorithm, by scavenging, that is, by linearly scanning all regions that might contain a pointer to an obsolete object, evacuating any obsolete object, and updating the reference.

Let us examine the reasons for performing scavenging. A primary reason is to be able to *reuse the address space*. (Note that *reusing real memory* is not an issue in virtual memory systems, since paging manages the use of real memory.) If the address space is small, it may be necessary to reuse addresses which previously held objects which became inaccessible to avoid exhausting the address space. Another reason for scavenging is to *compact the address space*. In systems with large address spaces, the page tables themselves may be subject to paging, so performance can be improved by compacting the address space. Additional reasons for scavenging are concerned with the disk. It may be necessary to reuse space on the disk or compacting the storage on the disk may result in reduced disk access time. (See Figure 3.)

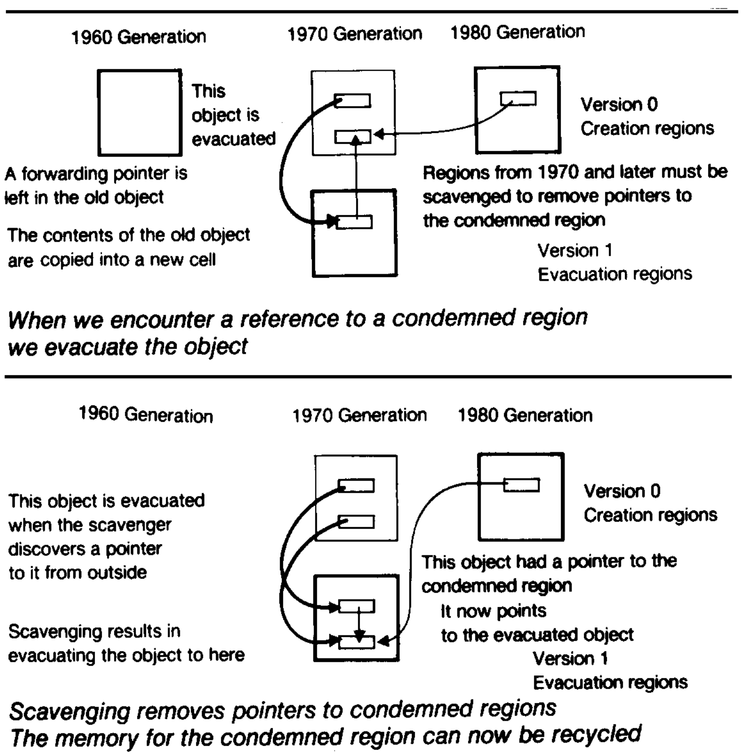

FIGURE 3. Evacuating and Scavenging

Scavenging is potentially a lot of work, and since our algorithm is designed to condemn regions at a much faster rate than Baker's algorithm does flips, the efficiency of scavenging is more crucial for our system. We will attempt to hold down the scavenging time by enforcing restrictions on where pointers may point, so that we will have a better chance of knowing where to look to find all references to a condemned region. These restrictions will cut down the amount of storage that has to be scanned to find and update obsolete references.

We intend to exploit some empirically observed properties of heap storage. Most pointers point *backward in time*, that is, objects tend to point to objects that were created earlier. This is because object creation operations like CONS can only create backward pointers, since the components of the object must exist before the object itself is created. Pointers which point *forward in time* can only arise as a result of a destructive operation like RPLACA which can assign a newer pointer as a component of an older object. Since we intend to condemn regions in recent generations more frequently than regions in older generations, we will try to engineer a scheme that reduces scavenging for newer generations at the expense of making scavenging more costly for older generations.

The idea is to *allow objects to point backward any number of generations*, *but to keep track of forward pointers*. By restricting pointers from older generations to newer generations, we can arrange that references to a region will come from either the same generation or from younger generations. Thus, when a region is condemned, we need not scavenge regions in any of the older generations. This will mean it will be much faster to reclaim regions in recent generations, since there will be comparatively little storage that needs to be scavenged.

What happens when an attempt is made to create a pointer from an older generation to a younger generation? Instead of pointing directly from the older object to the newer object, we require that the older object point *indirectly* through another cell held in an *entry table*. We now associate with each region containing objects another region called its entry table, which contains the indirect cells for all pointers to objects in that region from older generations. All pointers directly into an object-containing region from older generations must lie in the entry table. Of course, when the user's program references a pointer which points to an entry table, the link to the younger object is automatically followed, so this extra indirection is transparent to the user's program.

When a region *R* is condemned, only newer generations must be scavenged to find and update pointers into the condemned region. Instead of scavenging the older generations, the entry table associated with *R* is scavenged instead, since its purpose is to collect all pointers from older generations. Since pointers from older to younger generations are only produced by destructive operations like RPLACA, these operations must check to see if they might cause an older object to point to a younger object. We expect these pointers to be relatively rare compared to object creation operations, so the size of entry tables should be relatively small compared to the size of object regions. This is in keeping with our philosophy of making object creation cheap even if it requires a little more overhead on object modification. (See Figure 4.)

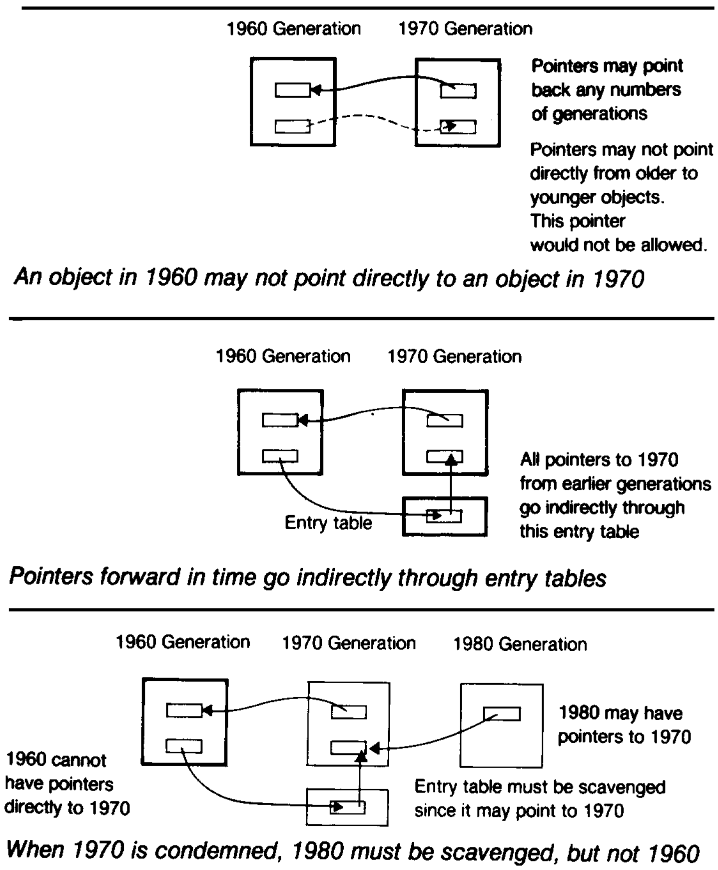

FIGURE 4. Entry Tables for Forward Pointers Reduce Scavenging.

What about storage reclamation for the entry tables themselves? How do we recover storage in the entry table when a pointer from an older to a younger object becomes inaccessible? Since we expect there to be a relatively small number of forward pointers, efficiency of storage management for entry tables is not as critical an issue as it is for objects. There are several alternatives, and here we present a suggestion of Lucassen's [21]: if we record the name of the region of the originating object with each entry in the entry table, we have a means of detecting inaccessible pointers in the entry table. When the system completes garbage collection and scavenging for a region, it is known that all objects in the region are inaccessible, and the system records the region in a list. When looking at entry tables, any cell created for an object in an inaccessible region is known to be inaccessible. This requires that region names are unique, which is not hard to assure, and also that entries are not shared, since every forward pointer gets its own entry.

The reader should be sure to understand that it is not necessary to wait for scavenging to be completed for one condemned region before another region can be condemned. Condemning a region starts a *wave* of scavenging, scanning all memory more recent than the condemned region. The wave stops when the scan reaches the most recent region, and memory for the condemned region is released. Many such waves can be present in the system at any time, without interfering with each other. Each wave of scavenging just needs to keep a pointer saying where it currently is working, and the pointer is advanced each time more scavenging is performed.

There is some flexibility about the order in which scavenging is performed. We would probably recommend always scavenging the oldest objects first. Paging during scavenging might be reduced by adopting a suggestion of Greenblatt's [13], or a similar one by Knuth [19], which would always prefer scavenging a resident page to one which is out on the disk.

We should point out that the idea of restricting pointers which point forward in time to go through entry tables is independent of the particular method used to accomplish garbage collection for each generation. It would be possible to substitute a more standard *mark and sweep* algorithm for the Baker-style copying garbage collection that we advocate.

5. OLDER OBJECTS ARE GARBAGE COLLECTED MORE SLOWLY THAN YOUNGER OBJECTS

The performance of our garbage collector is improved by varying the rate at which regions in a generation are condemned according to the age of the objects. A good heuristic is to assume that if objects have been around for a long time, they are relatively permanent and will continue to be accessible. This makes it reasonable to use the generation number and version number of a region as a guideline to decide when to condemn a region.

As the objects in a region get older, the operation of garbage collecting the region by making the region obsolete and evacuating all its accessible pointers will happen less frequently. This will save time that would have been wasted moving permanent objects around, at the cost of increasing the time it takes to reclaim those objects in the region which do become inaccessible. For regions containing mostly objects with long lifetimes, this tradeoff will be worthwhile. Young regions will contain a high percentage of garbage, so it is advantageous to reclaim inaccessible objects in these regions as soon as possible.

Recovering storage for old inaccessible objects is costly, since all the more recent memory must be scavenged. Since garbage collection is so expensive for old objects, we should do it infrequently, so the cost can be amortized over a long time period. Recovering storage for new inaccessible objects is cheap, since very little storage has to be scavenged.

Another consideration for deciding when to condemn regions is that it is necessary to be able to reclaim circular structures that cross generation boundaries. Some provision for these cycles must be made, otherwise our entry tables, which are analogous to reference counts in keeping track of references to a region, would inherit the same inability to deal with circular structures. Because of locality of reference, we expect the number of such cross-generation circular structures to be small. The easiest solution is to synchronize the condemnation operations, to assure that condemning a region implies condemning all regions younger than the condemned region. This need not be done every time the region is condemned, since it incurs additional expense; it needs to be done only from time to time to assure the circular structures are eventually reclaimed. Many regions will contain no forward pointers, so it might be worth marking these as such to avoid extra condemnations solely to recover circular structures.

An additional optimization that might be worthwhile for very old objects is to *coalesce* several adjacent generations. Since the number of objects in a generation decays with time, old generations may contain few objects. It would reduce scavenging time to look for pointers to any generation of a group rather than to just one generation, since scavenging for old generations requires going through many generations. This would reduce the paging time necessary to bring in all the pages between a very old generation and the present generation. Coalescing generations also tends to decrease the number of forward pointers, since combining generations collapses the two ends of the pointer into a single generation. This should also reduce the number of cross-generation circular structures.

6. WEAK POINTERS ARE TREATED LIKE FORWARD POINTERS

Normally, having a pointer to an object is an indication that the object is needed by some active program, and the garbage collector is only allowed to recover an object if no pointers exist to it. A few Lisp systems allow another kind of pointer, called a *weak pointer*, which does not protect the object pointed to from garbage collection. Why are weak pointers useful? Sometimes it is desirable to keep track of all currently available objects of a certain type in a list, so the user can ask, "What are all the objects I currently have?" But even if the user's program forgets about a certain object, the global list of all objects still points to the object, preventing it from being garbage collected. Or, representing part/whole relationships may require parts to have *backpointers* to a containing object, which should not necessarily protect that object from garbage collection.

Weak pointers are not followed by tracing in garbage collection schemes. In our scheme, objects connected by weak pointers would not be subject to evacuation. Implementing weak pointers poses a problem since we do not want to leave dangling references. When the object pointed to by a weak pointer is recycled, the weak pointer should be set to null. Thus weak pointers have to be controlled, and we can use the same mechanism to restrict weak pointers as we do for forward pointers. Weak pointers are constrained to point indirectly through entry tables in the same manner as forward pointers. When a region is condemned, it becomes easier to find all the weak pointers into the condemned region. When an object is recycled, the pointer in the entry table is modified. We assume the number of weak pointers is relatively small compared to ordinary, *strong* pointers.

7. VALUE CELLS AND STACKS MAY NEED SPECIAL CONSIDERATION

In presenting our garbage collection algorithms, we acted as if *all* pointers to objects were resident in the object memory itself. However, most present day Lisp implementations also involve internal *stacks*, which store control state information and variables. In *shallow binding* implementations of Lisp such as MacLisp and Lisp Machine Lisp, each atomic symbol representing a variable has a *value cell* associated with it to hold its current value. We must consider object references that reside in these places as well as those stored in object memory. (Alternatively, *deep binding* or *lexical binding* implementations of Lisp store values in data objects called *environments* and are not subject to this problem.)

The stack and value cells must be scavenged for pointers to obsolete objects before the memory for a condemned region may be recovered. No modification to our algorithm is essential, if we agree that value cells and stacks are to be treated as objects, even though they are not user-accessible objects in many implementations. Conceptually, we will consider the stack to be part of the "oldest" generation, since it is always present in the system. Value cells should be part of the oldest generation too, regardless of when they are actually created, since they are usually "permanent." When a reference to an object is created from the stack or from a value cell to an object, this will create a forward pointer, which must go through the entry table of the object. Thus, when the entry table is scavenged, all stack slots and value cells pointing to objects in its region will be scavenged.

Since in many Lisp systems, the performance of PUSH and POP operations on the stack is critical, it may be necessary to optimize these operations. Since objects stored on the stack are likely to be very temporary and modifications occur at a high rate, we might like some way of avoiding creating entry table pointers for each stack reference. A solution is to always consider the stack as part of the *youngest* generation instead of the oldest, so that no entry table pointers are kept for it. The stack must then be scavenged for each condemned region.

Two tricks make it possible to save some work in scavenging the stack. First, if more than one region is condemned at a time, it might pay to scavenge the stack for several condemned regions simultaneously rather than each individually. Second, keeping track of the top of the stack as it is pushed and popped may result in having to scavenge fewer objects. We observe that after a region is condemned, no new references may be created directly to objects in the condemned region, since our algorithm provides for evacuation of the object in that case. Popping the stack can remove references to a condemned region, but pushing objects on the stack can never result in new references to a condemned region. The number of references to a condemned region can only decrease due to pushing and popping after the region is condemned. Therefore, scavenging can always stop at the point where the scavenger meets either the current top of the stack or the top of the stack at the time the region was condemned, whichever is lower. (See Figure 5.)

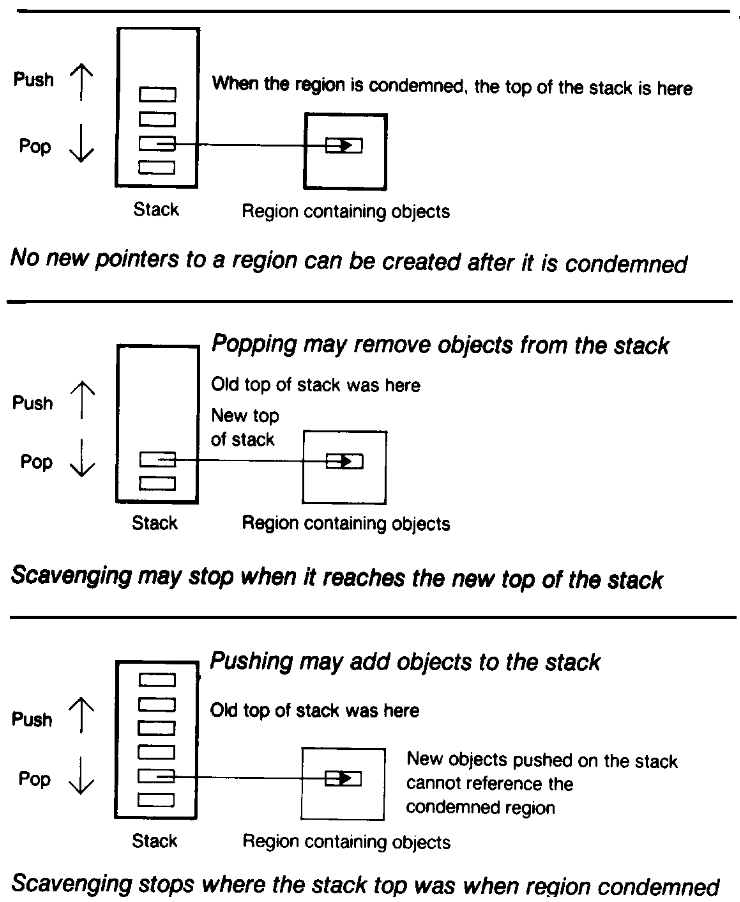

FIGURE 5. Scavenging the Stack

A suggestion which might help performance is to notice that the lifetime of short-lived objects is *approximately* (though not exactly!) correlated with pushing and popping the stack. This suggests that a good time to expect a lot of garbage is when returning from functions. This might lead to a policy of condemning regions after a certain number of stack pops.

Using linear stacks for temporary storage is a popular technique mainly because it has the property that we seek for our garbage collector: temporary storage is reclaimed quickly after becoming inaccessible. When Lisp calls a function, the arguments are pushed on a stack and automatically popped off when the function returns. The storage used for the arguments on the stack is immediately reusable as soon as the function returns. However, sticking to a strict stack discipline has its well-known problems, leading to the traditional *funarg problem* of Lisp [23]. Object-oriented languages do not follow a stack discipline, and we would like temporary storage in these languages to be efficient.

There is currently a sharp discrepancy between cheap stack storage and expensive heap storage. It should be the case that holding on to an object only slightly longer is only slightly more expensive. We would like to reduce reliance stacks, yet retain reasonable efficiency. Our hope is that we can reduce the cost of garbage collection in the case of temporary storage so that it is competitive with using a stack for temporary storage.

8. HOW GOOD IS THE PERFORMANCE OF OUR GARBAGE COLLECTOR?

Judging garbage collection algorithms is tricky. These algorithms are heavily dependent on the empirical properties of data used by programs, and their performance depends upon whether certain kinds of operations are cheap or expensive in the underlying machine. We believe our algorithm has the potential for good performance, considering tradeoffs appropriate for the machines that will be prevalent in the next couple of years and the needs of large-scale Al software.

The primary reason we expect good performance from our garbage collector is that it takes into consideration the lifetimes of objects. Our garbage collector should be more efficient than traditional alternatives for objects with short lifetimes, since it recovers the storage for these objects quickly after the object becomes inaccessible. Our garbage collector should be more efficient for objects with long lifetimes since the garbage collector wastes less time repeatedly examining objects that remain accessible for long periods.

One way to think about the efficiency of garbage collectors is to ask the question, "How much work does the garbage collector have to do *per memory cell reclaimed*?" Since the purpose of garbage collection is to recycle memory, more efficient garbage collectors should do as little work as possible to collect the garbage. We assert that considering the lifetime of objects results in reducing the amount of work necessary per inaccessible object recovered.

All the work our garbage collector does occurs in either the evacuation or scavenging phases. How does the amount of work for each phase compare with conventional alternatives? Since regions in younger generations are condemned more frequently than regions in older generations, most of the scavenging time is spent in younger generations. We make the plausible assumption that the proportion of garbage is higher in younger generations than older generations. Thus most of the scavenging time is spent where there is the highest proportion of inaccessible objects, which tends to minimize the amount of scavenging that needs to be performed per inaccessible object.

Our algorithm tends to require fewer evacuations per object for older objects at the expense of more evacuations per object for younger objects, as compared with the standard Baker algorithm. Our algorithm is at least no worse in this respect than Baker's since the rate of condemnation can always be adjusted so that the average number of evacuations per object is comparable.

Baker [5] considers another criterion for the efficiency of garbage collection: the *density* of accessible objects. A garbage collector is good if it maintains a high proportion of accessible objects to inaccessible objects in the address space, especially in primary memory for virtual memory systems. A problem with our algorithm is that it introduces fragmentation, since partially filled regions will waste some space, lowering the average density of accessible objects. However, just like the fragmentation problem in paging systems, the region size should be chosen so that fragmentation is not a significant source of inefficiency.

To maintain a high density of accessible objects, it is necessary to remove inaccessible objects as soon as possible. If we grant the hypothesis that most of the garbage occurs in younger generations, then most of the garbage will be removed quickly, since the rate of garbage collection is faster for younger generations.

9. WHAT ASPECTS OF PROGRAM BEHAVIOR AFFECT EFFICIENCY OF GARBAGE COLLECTION?

A next step in trying to determine whether our garbage collection scheme would be feasible involves observing the behavior of currently existing large-scale Lisp programs. The few simple kinds of measurements we describe below would help greatly in predicting the performance of our proposals. To our knowledge, no currently existing Lisp system is instrumented in such a way that the kinds of measurements we suggest are easily obtainable. We would strongly encourage readers to try to collect such data for their systems.

Rate of Object Creation. How fast are objects created?

Average Lifetime of Objects. How fast do objects become inaccessible? If there is a high proportion of short-lived objects, our proposals become advantageous.

Proportion of Forward versus Backward Pointers. How often do pointers point to objects that are younger than themselves versus pointing to older objects? Forward pointers can only be created by object modification operations or by creation of circular structures, not by creation of noncircular objects. *Delayed evaluation* (also called *suspended* or *lazy* evaluation) also results in creating forward pointers, but these can be implemented using invisible pointers, which are eventually removed in the course of garbage collection. The proportion of forward pointers will depend to some extent on the programming style adopted. A high percentage of pointers pointing to older objects bodes well for our scheme.

The Average "Length" of Pointers. How much locality is there in the program? Do pointers often point to nearby objects or to objects far away? Our proposal would fare well with programs which naturally have a high degree of locality of reference.

We believe it is plausible to expect that empirical observations would bear out our assumptions about program behavior and justify the design choices in our garbage collector. Certainly the trends are in the direction of programs with increased locality and toward programs that rely on object creation rather than modification.

Our future research plans include constructing a simulator that will allow us to test the behavior of real programs and to pick sensible values for parameters such as region size for a wide variety of conditions. Precise determination of how well our garbage collector will perform on real programs and comparison with more conventional alternatives must await actual implementation and measurement.

10. USERS CAN PREDICT THE LIFETIMES OF OBJECTS TO HELP THE GARBAGE COLLECTOR

Often a sophisticated user is in a position to know whether a particular object is likely to be relatively temporary or more permanent. The system should be able to take advantage of such knowledge to improve the performance of the program. It might be advantageous to supply the user with several different flavors of object creation operations, so that the system can choose the best allocation strategy appropriate for that kind of object. An operation could be supplied which creates objects directly in some older generation, rather than in the current generation. Of course, this decision will have no effect upon the semantics of the program; it will only affect the efficiency of garbage collection.

Adjusting the *region size* can control the efficiency of using short term versus long term memory. Short-lived objects should be allocated in small regions, so the storage for the object will be recovered very soon after it is abandoned. On the other hand, long-lived objects should be allocated from larger regions. This saves the system the trouble of having to frequently evacuate the object from generation to generation at the cost of having to wait longer before the storage can be recovered. Larger regions also reduce the expense of interregion pointers.

Since we expect that most storage is used for short-lived objects, we recommend that objects be created in short term memory by default. System primitives, like Lisp's PUTPROP, that expect to create relatively permanent objects can use longer term versions of CONS.

Being able to take advantage of a priori knowledge of the lifetimes of objects may become important for some kinds of systems. Trends are developing toward systems which create many structures which are known to be permanent at the time they are created. Several recently developed languages for artificial intelligence research produce some types of data which never become inaccessible.

Current implementations of new pattern-directed invocation languages like AMORD [9] or ETHER [18] do not have any operations that completely *remove* or *let go* of assertions in the database. Once an assertion is made, it remains forever, though belief in the assertion may be renounced by further processing. Description languages such as KRL [7] or OMEGA [2] currently have this characteristic as well. (However, future versions of ETHER and OMEGA are developing a notion of *viewpoints*, which may allow some knowledge to become inaccessible and be reclaimed.) These languages have not yet been applied to sufficiently large problems so that reclamation becomes an important issue in present day implementations.

Databases for business applications also may have the property that records are virtually permanent once created. Improvements in computer technology will make keeping data for long periods feasible, through storage hierarchies that make older data progressively harder to access, but never impossible. Very large address spaces may obviate the need for reuse of the address space. We may reasonably expect computers in the next generation that may be able to run for weeks to years without needing to reuse address space [26]. Write-once media such as video disks may be used for secondary storage, so that reusing or compacting secondary storage space becomes less of an issue.

Under circumstances such as these, knowing that data is permanent helps the garbage collector avoid performing too much work scavenging, for example, trying to find inaccessible objects where there are none. The need for garbage collection is not totally eliminated in these systems, however, as processing of individual database entries, indexing and retrieval may require creating short-lived objects.

11. OUR GARBAGE COLLECTOR IS SUITABLE FOR PARALLEL PROCESSOR MACHINES

Since processors are continually getting cheaper, multiprocessor machines will soon appear. The incentive for using multiple processor machines is especially important for Al applications. Our garbage collection scheme has been designed to be suitable for implementation on multiple processor machines.

On a multiprocessor system in which several processors share common memory, an attractive way to exploit parallelism is to allocate processors to be *scavengers*, performing the scavenging task concurrently with *worker* processors, which run user programs. Care must be taken to avoid timing errors and contention for shared resources. The major potential trouble spot with our scheme occurs when objects are being evacuated. Objects can be evacuated either by a worker who references an obsolete object or by a scavenger. The danger here is that one processor may attempt to evacuate an obsolete object, creating a new object, and before the invisible pointer to the new object is installed, another processor may try to evacuate the same object. Evacuation operations on the memory must have sufficient synchronization to prevent this.

We prefer a multiprocessor architecture such as the Apiary [16] in which each worker processor has its own memory, not shared by other processors. We will briefly describe how our algorithm can be extended to operate on such a machine.

Each worker maintains its own storage, allocates its own regions, condemns them periodically, evacuating and scavenging exactly as described for the single processor case. A consideration arises when a worker must reference an object which lives on another worker. Such an object may reside in a condemned region and need evacuation. Another consideration is that when a region is condemned, pointers to that region from other machines must be scavenged.

On the Apiary, each worker maintains two tables to manage pointers that reference objects residing on other workers. The first table is an *exit table* for references *to other machines*. When an object on another machine is referenced, a message is sent out over the network to fetch the object, so the user's program objects on other machines do not have to be treated differently than objects on one's own machine. We arrange that a worker receiving a request for an object first checks to see whether the object is obsolete, and if so, evacuates it, returning the evacuated object. This assures that workers will never reference condemned regions on other machines.

Each worker also has an *interest table*, which keeps track of references to objects on that worker *from other machines*. When a region on a worker is condemned, the interest table must be scavenged since it may reference the condemned region. Here our solution to the problem of forward pointers comes in handy. We can require all pointers from other machines to go indirectly through the entry table in the same manner as we required for forward pointers. This reduces the amount of work during scavenging, and the extra overhead on intermachine pointers (which we assume to be relatively rare compared to intramachine pointers) should not be significant.

12. CHEAPER SHORT TERM MEMORY MAY IMPROVE PROGRAMMING STYLE

It is our hope that making the use of short-lived objects cheaper will lead to improvements in program clarity. Often, complications in program structure are motivated by the need to avoid creating short-lived objects for intermediate results.

Here is an example of how the cost of short-lived objects can affect design decisions in programming. Consider the problem of writing *a matrix multiplication* routine in Lisp to operate on matrices represented as lists of rows, each row represented as a list of numbers.

$$\begin{pmatrix}1 & 2\\3 & 4\end{pmatrix} \times \begin{pmatrix}5 & 6\\7 & 8\end{pmatrix} = \begin{pmatrix}19 & 22\\43 & 58\end{pmatrix}$$

This example would be represented as

(MATRIX-MULTIPLY '((1 2) (3 4)) '((5 6) (7 8)))
evaluates to ((19 22) (43 50))

Let us imagine that as part of our mathematics library we already have a function which takes the *dot product* of vectors and a function which produces the *transpose* of a matrix:

(DOT-PRODUCT '(1 2) '(5 7)) *evaluates to* 19
(TRANSPOSE '((5 6) (7 8))) *evaluates to* ((5 7) (6 8)))

The usual procedure for multiplying a matrix is to compute the elements of the product by multiplying elements of the rows of the first matrix by elements of the columns of the second matrix. Using the transpose procedure, we can turn the columns of the second matrix into rows, so that they "line up" with the rows of the first matrix, then use the dot product function to multiply corresponding rows. This solution is elegantly expressed as follows:

```
Define MATRIX-MULTIPLY of a LEFT-MATRIX and a RIGHT-
MATRIX:
  Let COLUMNS be the TRANSPOSE of RIGHT-MATRIX.
  Create a list whose elements are:
    For each ROW in the LEFT-MATRIX,
      Create a list whose elements are:
        For each COLUMN in the COLUMNS matrix,
        the DOT-PRODUCT of the ROW and the COLUMN.
```

(The actual Lisp code corresponding to the descriptions of algorithms in this section appears in Appendix III.) This solution has a potential efficiency problem: the TRANSPOSE function creates a new list that is thrown away after the matrices are multiplied.

In a conventional Lisp system, using lists like this is expensive, since the lists are created and only used for a short time before being subject to garbage collection. This leads programmers to try to optimize out the creation of intermediate list structure. Instead of doing a "two-pass" procedure over the matrix, one to transpose, another to multiply, we can use instead a more complicated "one-pass" procedure. Instead of creating a new list whose elements are in a convenient order, the one-pass procedure extracts the appropriate elements from the columns of the matrix when needed. Especially if multiplications of small matrices are frequent, the following version might be considerably faster in a conventional Lisp system:

```
Define MATRIX-MULTIPLY-WITHOUT-TRANSPOSING
of LEFT-MATRIX and RIGHT-MATRIX:
  Create a list whose elements are:
    For each ROW in the LEFT-MATRIX,
      Create a list whose elements are:
        For COLUMN-INDEX from 0 to the number of columns of
        RIGHT-MATRIX:
          The DOT-PRODUCT-COLUMN of
            the ROW,
            the RIGHT-MATRIX, and
            the COLUMN-INDEX.
```

This now forces us to write a new DOT-PRODUCT routine which can extract the elements of the second vector from the columns of the matrix. This duplicates some of the knowledge we already had in the DOT-PRODUCT function.

```
Define DOT-PRODUCT-COLUMN of a ROW, a MATRIX, and a
COLUMN-INDEX;
  If the ROW is empty, return 0.
  Otherwise, return the sum of:
    the product of
      The FIRST element of the row,
      and the element indexed by COLUMN-INDEX of the FIRST
      element of the MATRIX.
  and the DOT-PRODUCT-COLUMN of
        the REST of the ROW,
        the REST of the MATRIX,
        and the COLUMN-INDEX.
```

Instead of being able to modularly build a solution using the TRANSPOSE and DOT-PRODUCT functions we already had, we were forced to write new lower level routines. The need to avoid using short-lived objects encourages more complex and obscure programming techniques.

This example is an illustration of a general situation where an *N*-pass procedure will use data objects to store the output of intermediate passes. There is a temptation to substitute a one-pass procedure to avoid using storage for intermediate results, but this procedure has to be more complicated and specialized, because the code inside the loop must do a little piece of all of the passes.

Another approach for reducing inefficiency due to creating objects to store intermediate results is *program transformation* [8, 14]. The hope is that a smart compiler could replace a program which uses temporary storage with another equivalent version that did not, much as in our two versions of matrix multiplication. We consider research in program transformation techniques extremely valuable, but program transformation is no substitute for an efficient garbage collector.

Besides introducing difficulty in debugging (the system must be able to relate bugs in the transformed version of the program to bugs in the original), programs may use short-lived objects in a dynamic way which might thwart static compilation. The lifetime of objects is often short, but unpredictable, and we would like our system to deal with this kind of object efficiently. The simplest class of short-lived objects are those created by a procedure for its own use and abandoned when the procedure returns. This is the kind of procedure that is most amenable to optimization by program transformation systems. There is another kind of procedure, represented by the matrix multiplication example, where a procedure creates an object that is returned and then used temporarily by some caller of the procedure. These are much more difficult to compile out, especially in compilers that allow separate compilation of procedures.

The optimization may also depend on the outcome of runtime events, making it impossible for any static optimizer to perform the optimization. A third class involves using a short-lived object as part of some data structure and later modifying it, making the object inaccessible within a short time. Some uses of these objects are not correlated with the procedure calling stack at all, and program transformation systems will have little success with these.

For example, a user may keep a *directory*, which contains objects like files which the user may choose to *delete* at any time. The exact moment at which a file is deleted is completely unpredictable by a program transformation system, and therefore a garbage collector is necessary.

Our aim is to make the use of short-lived objects more efficient, so that the creation of short-lived objects is not much worse than allocating temporary results on a stack. If programmers are not severely penalized in terms of efficiency for choosing cleaner programming styles, we hope that they will continue to improve their programming styles.

APPENDIX I. THE BAKER REAL-TIME GARBAGE COLLECTION ALGORITHM

We present a summary of Henry Baker's original algorithm.

The CREATE operation creates objects, like Lisp's CONS. ACCESS retrieves a component of an object, like Lisp's CAR and CDR, or accesses a subscripted element of an array. MODIFY performs assignments to components of objects, like Lisp's RPLACA and RPLACD, or storing into a subscripted element of an array.

The address space is divided into two semispaces, *fromspace* and *tospace*. Object creation happens in tospace and the semispaces are exchanged in a *flip* when tospace fills.

```
Define CREATE an object, given an INITIAL-CONTENTS:
  If the INITIAL-CONTENTS is in FROMSPACE,
    EVACUATE the INITIAL-CONTENTS to TOSPACE.
  Allocate memory space at the location CREATE-OBJECTS-FROM-
  HERE.
  Fill the memory with its INITIAL-CONTENTS.
  Advance the CREATE-OBJECTS-FROM-HERE pointer past the new
  object.
  Call the SCAVENGER to perform a bit of the work to reclaim
  memory.
  Return the pointer to the new object.

Define ACCESS an ELEMENT of an OBJECT:
  If the CONTENTS of the cell containing the ELEMENT is in
  TOSPACE, return it.
  If it's in FROMSPACE,
    Check to see if the cell contains an INVISIBLE-POINTER
      [which means it has already been evacuated.]
      If so, change the ELEMENT to
        where the INVISIBLE-POINTER points
      and return that forwarded object.
    If it's in FROMSPACE and there's no INVISIBLE-POINTER,
      EVACUATE it from FROMSPACE to TOSPACE.
      Update the ELEMENT of the OBJECT to point to the new
      object in TOSPACE,
      and return the new object in TOSPACE.
```

When an object in fromspace is accessed, it is EVACUATEd, moving it to tospace. An INVISIBLE POINTER is left behind so references to it will still work.

```
Define EVACUATE an OLD-OBJECT:
  If the OLD-OBJECT is in FROMSPACE,
    Copy the OLD-OBJECT into the EVACUATION area of
    TOSPACE,
      creating a NEW-OBJECT.
    Leave an INVISIBLE-POINTER in the old cell to the NEW-OB-
    JECT.
    Return the NEW-OBJECT.
  If the OLD-OBJECT is in TOSPACE, just return it.
```

The SCAVENGER makes sure all objects in tospace also have their components in tospace. There is a variable SCAVENGER-SLICE that controls how much work in reclaiming storage is performed every time an object is created. There is a pointer SCAVENGE-HERE that points to the next object to be scavenged.

```
Define the SCAVENGER:
  Repeat the following until either
      No more UNSCAVENGED objects remain in the
      EVACUATION area of TOSPACE, or the loop has been repeated
      SCAVENGER-SLICE times:
    SCAVENGE the object at the location SCAVENGE-HERE by
      EVACUATING each element which lies in FROMSPACE,
      moving that element to TOSPACE,
      and changing the element to point to the TOSPACE object.
    Advance the SCAVENGE-HERE pointer.
  When no more UNSCAVENGED objects are left,
    do a FLIP, exchanging FROMSPACE and TOSPACE.

Define the procedure to MODIFY an ELEMENT of an OBJECT to have
a NEW-CONTENTS:
  If the NEW-CONTENTS is in FROMSPACE,
  EVACUATE the NEW-CONTENTS, yielding an object in
  TOSPACE.
  Store the TOSPACE object NEW-CONTENTS into the cell
    containing the ELEMENT of the OBJECT.
```

APPENDIX II. OUR REAL-TIME GARBAGE COLLECTOR

Creation and access are similar to Baker's, except that instead of fromspace and tospace, memory is allocated in *regions: creation regions* to create objects, *evacuation* regions to move objects from older to newer regions. Instead of Baker's flips, regions are *condemned*, which causes the accessible objects to begin to move out of the region. Condemning a region starts scavenging to remove pointers to the region. When scavenging is complete, the memory for the region can be recycled.

The description that appears here has been somewhat simplified for expository purposes. We have omitted code for handling the stack, value cells, storage management for entry tables, and multiprocessing. These considerations have been discussed in the body of the paper, and modifications to the algorithms below to incorporate them are straightforward.

Define CREATE an object, given an INITIAL-CONTENTS:
If INITIAL-CONTENTS is OBSOLETE [its region was CONDEMNED],
EVACUATE the INITIAL-CONTENTS.
If there's no more room in the current CREATION region,
Make a new CREATION region from which to create objects,
Inheriting GENERATION and VERSION numbers from the previous one.
Call the SCAVENGER coroutine to incrementally try to reclaim memory.
If the POPULATION of the current generation is high enough,
Start a new GENERATION by
Incrementing the CURRENT-GENERATION-NUMBER.
Allocate memory space at the location CREATE-OBJECTS-FROM-HERE.
Fill the memory for the object with its INITIAL-CONTENTS.
Advance the CREATE-OBJECTS-FROM-HERE pointer past the new object.
Return the pointer to the new object.

Define the SCAVENGER:
Each time an object is created,
do SCAVENGER-SLICE steps of the following procedure:
Look for a region which is scheduled to be CONDEMNED.
A region is CONDEMNED when it is considered likely to contain garbage:
Young regions are CONDEMNED frequently, older ones more seldom.
SCAVENGE for all pointers into the CONDEMNED region.
Abandon the memory for the CONDEMNED region
when the scavenger has removed all pointers to it.

Define CONDEMN a REGION:
Mark the REGION as being CONDEMNED.
Allocate a new EVACUATION-REGION whose
GENERATION number is taken from the CONDEMNED region, and
whose VERSION number is one higher than the CONDEMNED region.

Define ACCESS an ELEMENT of an OBJECT:
If the CONTENTS of the cell containing the ELEMENT isn't OBSOLETE,
then return it.
If the CONTENTS is OBSOLETE [resides in a CONDEMNED region],
Check to see if the cell contains an INVISIBLE-POINTER
[which means it has already been evacuated.]
If so, change the ELEMENT to
where the INVISIBLE-POINTER points
and return that forwarded object.
If it's OBSOLETE, and there's no INVISIBLE-POINTER,
EVACUATE it.
Update the ELEMENT of the OBJECT to point to the EVACUATED object,
and return that new object.

Define EVACUATE an OLD-OBJECT:
If the OLD-OBJECT is OBSOLETE [in a CONDEMNED region]
Copy the object to an EVACUATION region
of the same GENERATION as the region containing the OLD-OBJECT
and whose VERSION number is one higher,
creating a NEW-OBJECT.
Store the contents of OLD-OBJECT into the NEW-OBJECT.
Leave an INVISIBLE-POINTER in the old cell to the NEW-OBJECT,
and return the NEW-OBJECT.

The scavenger removes pointers to obsolete objects by evacuating such objects. As soon as the scavenger is finished removing all such pointers, the memory for the region can be reclaimed.

Define SCAVENGE for pointers to a CONDEMNED-REGION:
  Repeat the following for each region
  in the GENERATION of the CONDEMNED-REGION,
  and the ENTRY TABLE for the CONDEMNED-REGION,
  and all GENERATIONS more recent than the GENERATION of the
  CONDEMNED-REGION:
    For each OBJECT in each REGION:
    SCAVENGE the OBJECT,
      looking for pointers to the CONDEMNED-REGION.

Define SCAVENGE an OBJECT, which may point to a CONDEMNED-REGION:
  Check to see if any element of the OBJECT points to the
  CONDEMNED-REGION.
    If it does, EVACUATE the element of the OBJECT and modify
    the element of the OBJECT to point to the evacuated object.

Define the procedure to MODIFY an ELEMENT of an OBJECT
  to have a NEW-CONTENTS:
  If the NEW-CONTENTS is OBSOLETE,
  EVACUATE the NEW-CONTENTS.
    Is the GENERATION of the NEW-CONTENTS
        younger than the GENERATION of the OBJECT?
      No, store the object NEW-CONTENTS into the cell containing
        the ELEMENT of the OBJECT.
      If it is, create a new ENTRY for the OBJECT in the ENTRY
      TABLE of the NEW-CONTENTS.
        Modify the cell containing the ELEMENT of the OBJECT to
          point to the ENTRY.
        Modify the ENTRY to point to the NEW-CONTENTS.
      Otherwise, store the NEW-CONTENTS in the cell containing
        the ELEMENT of the OBJECT directly.
    If the OLD-CONTENTS of the cell pointed to a younger object,
    Remove its entry in the ENTRY TABLE.

APPENDIX III. LISP CODE FOR THE MATRIX MULTIPLICATION EXAMPLE

First, the solution which *transposes* the right matrix.

```
(DEFUN DOT-PRODUCT (LEFT-VECTOR RIGHT-VECTOR)
  (COND ((OR (NULL LEFT-VECTOR) (NULL RIGHT-VECTOR)) 0)
        ((+ (* (CAR LEFT-VECTOR) (CAR RIGHT-VECTOR))
            (DOT-PRODUCT
            (CDR LEFT-VECTOR)
            (CDR RIGHT-VECTOR))))))

(DEFUN TRANSPOSE (MATRIX)
  (COND ((NULL (CAR MATRIX)) NIL)
        ((CONS (MAPCAR 'CAR MATRIX)
        (TRANSPOSE (MAPCAR 'CDR
        MATRIX))))))

(DEFUN MATRIX-MULTIPLY (LEFT-MATRIX RIGHT-MATRIX)
  (LET ((COLUMNS (TRANSPOSE RIGHT-MATRIX)))
       (MAPCAR
       '(LAMBDA (ROW)
                (MAPCAR
                '(LAMBDA (COLUMN)
                         (DOT-PRODUCT ROW
                         COLUMN))
                 COLUMNS))
                         LEFT-MATRIX)))
```

The following solution, which avoids transposing the matrix, replaces MATRIX-MULTIPLY-WITHOUT-TRANSPOSING for MATRIX-MULTIPLY and DOT-PRODUCT-COLUMN for DOT-PRODUCT:

```
(DEFUN MATRIX-MULTIPLY-WITHOUT-TRANSPOSING
(LEFT-MATRIX RIGHT-MATRIX)
  (MAPCAR
  '(LAMBDA (ROW
            (LET ((COLUMN-INDEX 0)
                 (MAPCAR
                  '(LAMBDA (COLUMN
                           (PROG1 (DOT-PRODUCT-COLUMN
                                  ROW
                                  RIGHT-MATRIX
                                  COLUMN-INDEX)
                                  (SETQ COLUMN-INDEX
                                  (+ COLUMN-INDEX 1.))))
                  (CAR RIGHT-MATRIX))))
  LEFT-MATRIX))

(DEFUN DOT-PRODUCT-COLUMN
(ROW MATRIX COLUMN-INDEX)
  (COND ((NULL ROW) 0)
        ((+ (* (CAR ROW)
              (NTH COLUMN-INDEX (CAR MATRIX)))
           (DOT-PRODUCT-COLUMN
              (CDR ROW)
              (CDR MATRIX)
              COLUMN-INDEX)))))
```

Acknowledgment. We would like to thank David Moon, who is implementing garbage collection for the Lisp Machine, for discussions concerning the ideas presented here and for finding bugs in our earlier proposals.

Tom Knight and Gerry Sussman were among the first to become concerned about the feasibility of the Baker algorithm because of locality problems and the lengthy interval between flips. Their concern helped motivate our work. We would like to thank Richard Stallman for help with the value cell problem and for suggesting several plausible alternatives to specific aspects of our proposals. Comments on an earlier draft from Donald Knuth and from an anonymous reviewer substantially improved our algorithm and its presentation in this paper. We also thank Danny Hillis for his koan about garbage collection at the beginning of this paper.

We would like to thank Hal Abelson, Russell Atkinson, Giuseppe Attardi, Henry Baker, Alan Bawden, Peter Deutsch, Richard Greenblatt, Danny Hillis, Jack Holloway, Dan Ingalls, Ted Kaehler, Kenneth Kahn, Tom Knight, William Kornfeld, Marc LeBrun, John Lucassen, Dexter Pratt, Dave Robson, JonL White and David Wise for their helpful comments on this paper, and Priscilla Cobb for proofreading help.

REFERENCES

1. Allen, J. *Anatomy of Lisp*. McGraw-Hill, New York, 1979.
2. Attardi, G., and Hewitt, C. Knowledge embedding in the description system OMEGA. Presented at the American Association for Artificial Intelligence Conf., Stanford Univ., Stanford, Calif., 1980.
3. Baker, H. Actor systems for real time computation. Tech. Rept. TR-197, MIT Lab. for Computer Science, Cambridge, Mass., 1978.
4. Baker, H.G. List processing in real time on a serial computer. *Commun. ACM 21*, 4 (April 1978) 280-294.
5. Baker, H. The paging behavior of the Cheney list copying algorithm. Tech. Note 1, Symbolics, Inc., Cambridge, Mass., 1980.
6. Bishop, P. Computer systems with a very large address space and garbage collection. Tech. Rept. TR-l78 MIT Lab. for Computer Science, Cambridge, Mass., May 1977.
7. Bobrow, D., and Winograd, T. An overview of KRL: A language for knowledge representation. *Cognitive Science 1*, (1977).
8. Burstall, R.M., and Darlington, J.L. A transformation system for developing recursive programs. *J. ACM 24*, 1 (Jan. 1977), 24-77.
9. deKleer, J., Doyle, J., Rich, C., Steele, G., and Sussman, G. AMORD--A deductive procedure system. Memo 435. MIT Artificial Intelligence Lab., Cambridge, Mass., Jan. 1978.
10. Deutsch, L.P., and Bobrow, D.G. An efficient, incremental, automatic garbage collector. *Commun. ACM 19*, 9 (Sept. 1976). 522-526.
11. Dijkstra, E., Lamport, L. et al. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM 21*, 11 (Nov. 1978), 966-975.
12. Friedman, D., and Wise, D. Garbage collecting a heap which includes a scatter table. *Inf. Process. Lett. 5*, 6 (Dec. 1976).

13. Greenblatt, R., Knight, T., Holloway, J., and Moon, D. A Lisp Machine. Presented at the Workshop on Computer Architecture for Non-Numeric Processing, Pacific Grove, Calif., March 1980.

14. Guibas, L., and Wyatt, D. Compilation and delayed evaluation in APL. Presented at the 5th ACM Conf. Principles of Programming Languages, 1978.
15. Hewitt, C. Viewing control structures as patterns of passing messages. In P. Winston and R. Brown (Eds.), *Artificial Intelligence: An MIT Perspective*, MIT Press, Cambridge, Mass., 1979.
16. Hewitt, C. The Apiary network architecture for knowledgeable systems. In Proc. 1980 Lisp Conf., Stanford Univ., Stanford, Calif., 1980.
17. Ingalls, D. The smalltalk-76 programming system: Design and implementation. Presented at the 5th ACM Conf. Principles of Programming Languages, 1978.
18. Kornfeld, W. Ether—A parallel problem solving system. Presented at the 6th Joint Conf. Artificial Intelligence, Tokyo, Japan, Aug. 1979.
19. Knuth, D. Garbage collection in real time. Class handout for course CS144C. Stanford Univ., Stanford, Calif., Spring 1981.
20. Lieberman, H. A preview of act 1. AI Memo 625, MIT Artificial Intelligence Lab., Cambridge, Mass., 1980.
21. Lucassen, J.M. Improvements to the Lieberman-Hewitt garbage collector. Term Paper for MIT course 6.845, May 1981.
22. Moon, D. MacLisp Reference Manual. MIT Lab. for Computer Science. Cambridge, Mass., 1980.
23. Moses, J. The function of Function in Lisp. Memo, *ACM SIGSAM Bull.*, July, 1970.
24. Snyder, A. An object-oriented machine architecture. Tech. Rept. TR-209, MIT Lab for Computer Science, Cambridge, Mass., 1979.
25. Weinreb, D., and Moon, D. Lisp Machine Manual. MIT Artificial Intelligence Lab., Cambridge. Mass., 1978.
26. White, J. Memory management in a gigantic Lisp environment or GC considered harmful. In Proc. 1980 Lisp Conf., Stanford, Calif.

CR Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors); D.1 [**Programming Techniques**]; D.4.2 [**Operating Systems**]: Storage Management; E.1 [**Data Structures**]; I.2.4: [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods; I.2.5**:** [**Artificial Intelligence**]: Programming Languages
General Terms: Algorithms, Languages, Performance.
Additional Keywords and Phrases: real-time garbage collection, Lisp, object-oriented programming, reference counting, virtual memory, parallel processing

Received 4/80. revised 10/81; accepted 5/82

This research is supported in part by the Office of Naval Research under our contract N00014-75-C-0522, in part by the Advanced Projects Research Agency under ARPA contract N00014-80-C-0505 and in part by a grant from the System Development Foundation.

Authors' Present Address: Henry Lieberman and Carl Hewitt, MIT Artificial Intelligence Laboratory, 545 Technology Square, Room 765. Cambridge, MA 02139.