AD-A084 819 MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTE--ETC F/6 9/2
A REAL TIME GARBAGE COLLECTOR THAT CAN RECOVER TEMPORARY STORAG--ETC(U)
APR 80 H LIEBERMAN, C HEWITT N00014-75-C-0522
UNCLASSIFIED AI-M-569 NL

1 OF 1
AD
A084819

END
DATE
FILMED
6-80
DTIC

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER A.I. Memo 569	2. GOVT ACCESSION NO. AD-A084 819	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Real Time Garbage Collector That Can Recover Temporary Storage Quickly.		5. TYPE OF REPORT & PERIOD COVERED memorandum
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Henry Lieberman, Carl Hewitt		8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0522, N00014-75-C-0643
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE April 1980 13. NUMBER OF PAGES 32
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

LEVEL

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

DTIC SELECTE MAY 27 1980 D

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

garbage collection
temporary storage
compaging storage
reference counting
LISP
object-oriented programming
stacks
virtual memory

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In previous heap storage systems, the cost of creating objects and garbage collection is independent of the lifetime of the object. Since temporary objects acount for a large portion of storage use, it's worth optimizing a garbage collector to reclaim temporary storage faster. We present a garbage collection algorithm which: Makes short term storage cheaper. Operates in realtime - object creation and access times are bounded. Works well with multiple processors and a large address space.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ADA084819

DDC FILE COPY

Massachusetts Institute of Technology
Artificial Intelligence Laboratory

A.I. Memo No. 569 April, 1980

A Real Time Garbage Collector That Can Recover Temporary Storage Quickly

Henry Lieberman
Carl Hewitt

Abstract

In previous heap storage systems, the cost of creating objects and garbage collection is independent of the lifetime of the object. Since temporary objects account for a large portion of storage use, it's worth optimizing a garbage collector to reclaim temporary storage faster. We present a garbage collection algorithm which:

Makes short term storage cheaper than long term storage.
Operates in real time - object creation and access times are bounded.
Works well with multiple processors and a large address space.

Acknowledgements: This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Office of Naval Research under Office of Naval Research contract N00014-75-C-0522, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

80 5 22 0 69

A Real Time Garbage Collector That Can Recover Temporary Storage Quickly

Henry Lieberman and Carl Hewitt

Artificial Intelligence Laboratory
and Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

In previous heap storage systems, the cost of creating objects and garbage collection is independent of the lifetime of the object. Since temporary objects account for a large portion of storage use, it's worth optimizing a garbage collector to reclaim temporary storage faster. We present a garbage collection algorithm which:

Makes short term storage cheaper than long term storage.
Operates in real time - object creation and access times are bounded.
Works well with multiple processors and a large address space.

1. Temporary storage should be cheaper than long term storage

In Lisp, and similar systems with dynamically allocated storage, the most common use of memory is as *temporary storage*. Most objects are created, used for a while, then thrown away. The traditional garbage collection algorithms [13] have the defect that temporary storage is just as costly as more permanent storage. It takes the same time to recover the storage for an object that becomes inaccessible, regardless of the lifetime of the object. Empirical studies of Lisp programs indicate that there is much to be gained in performance by optimizing the special case of recovering temporary storage. The performance of the new generation of object oriented, message passing systems [3], [4], [5] will rely increasingly on the efficiency of temporary storage.

Some systems use *reference counts* instead of garbage collection, primarily because a reference count system can reclaim temporary storage more quickly. A reference count system has the property that temporary storage is re-usable as soon as it becomes inaccessible, when its reference count reaches zero. However, reference count systems have formidable problems of their own. They cannot reclaim circular structures, which are becoming increasingly important in sophisticated AI

programming. 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. Some have also proposed more complicated systems which combine reference counts with garbage collection [11], [12].

In this paper we propose a simple extension to a garbage collection algorithm which can recover temporary storage quickly. Our idea builds on the algorithm of Henry Baker [1] which 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 which will work well on machines with a very large address space [14]. We believe these properties will be essential to make garbage collection practical on the next generation of computers. The suggestions described in this paper are currently being implemented on the MIT Lisp Machine [9], [10].

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 re-used. An operation called a *flip* occurs, where the tospace becomes the fromspace and vice versa.

When a object is evacuated from fromspace to tospace, a *forwarding pointer* or (*invisible pointer*) is left in the fromspace memory cell pointing at its new location in tospace. Whenever the fromspace cell is referenced, the forwarding pointer is followed and the reference is changed to point to the tospace object.

The operations which 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 a object is first evacuated to tospace, one of its components can point back to fromspace. We'd 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 which linearly scans the evacuation area of tospace and 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.

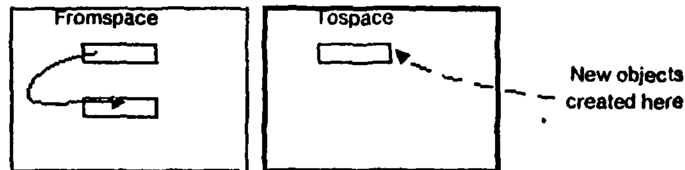
Figure [1]

(See next page)

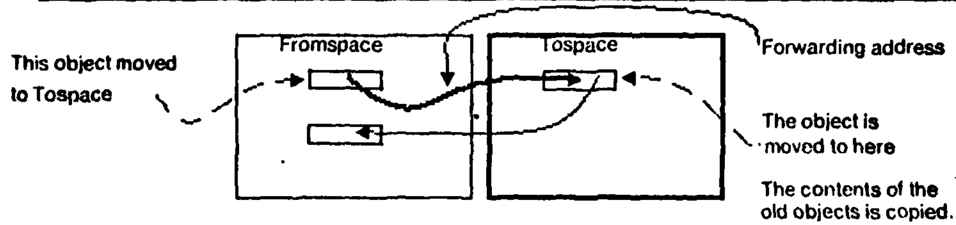
Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or special
A

Figure 1

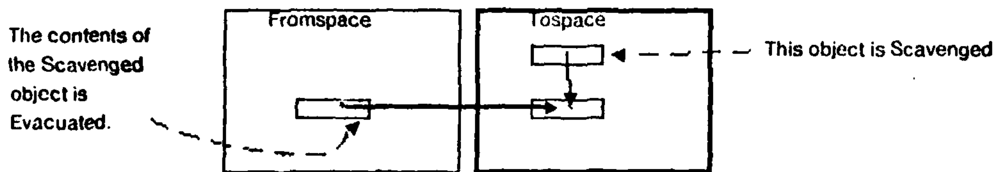
Henry Baker's Real Time Garbage Collector



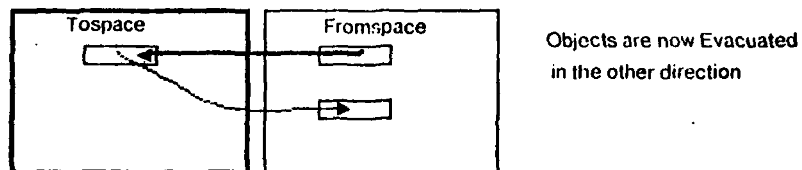
Memory is divided into Fromspace and Tospace



Evacuating an object moves it from Fromspace to Tospace



Scavenging an object removes pointers to Fromspace



After a Flip, Fromspace and Tospace are exchanged

(For reference, a more detailed description of the Baker algorithm appears in an appendix.)

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 the procedure appears in an appendix. For the moment, let's 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 will involve two major improvements to Baker's. Baker divides the address space into two halves, fromspace and tospace (cutting down the effectively usable address space by a factor of two). 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 won't commit ourselves to a particular size for regions, but regions should be small compared to the address space. The machine should be able to quickly tell, for a given page, what region it belongs to.

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's 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 temporary or more permanent. Regions are organized into *generations.* The system keeps track of a *current generation number* and 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'll call objects *obsolete* if they reside in a region that's 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.

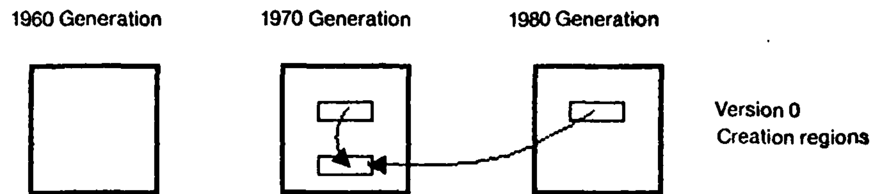
Objects are evacuated in the same way as in the original Baker algorithm. We allocate space for a new object in the evacuation region, and copy the contents of the old object into the new space. A forwarding pointer is left in the old memory cell pointing to the new object. If we encounter any reference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

Figure [2]

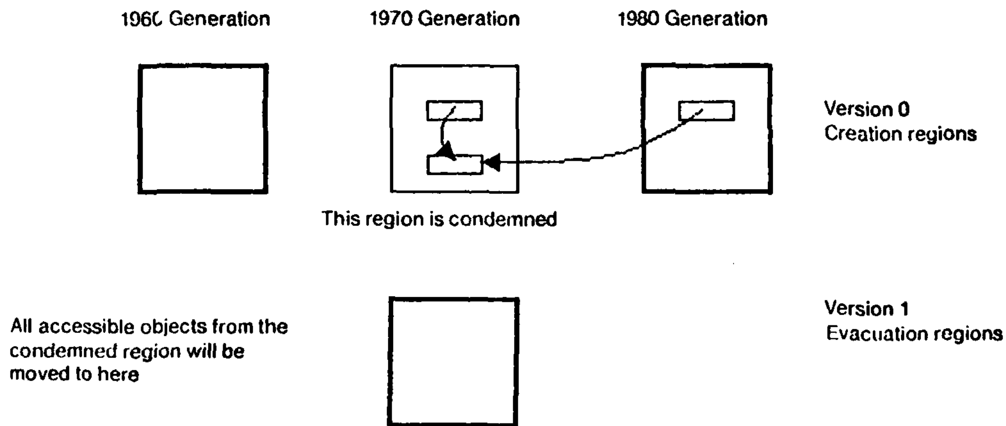
(See next page)

Our Real Time Garbage Collector

Figure 2



Memory is allocated in small regions
Regions are tagged with generation and version numbers



Garbage collecting a region is initiated by condemning it
Accessible objects from the condemned region
will be evacuated to a new region

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 time is reduced by restrictions on where pointers can point

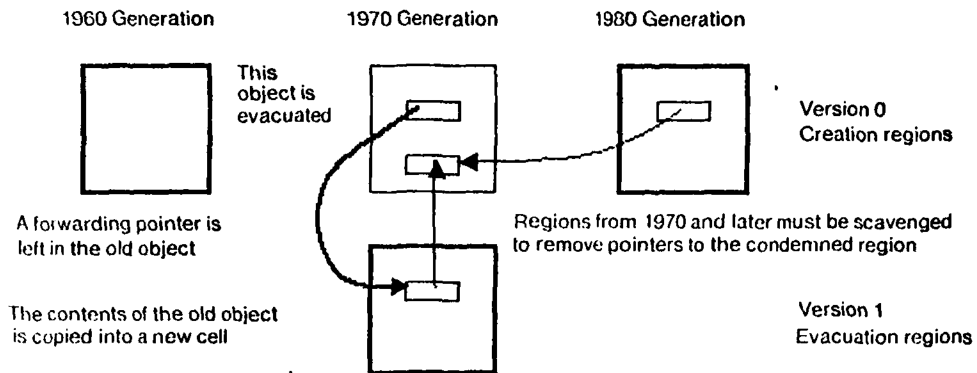
In order to release memory for a condemned region, we have to make sure that no pointers from outside the condemned region point to it. This is done, as in Baker's algorithm, by *scavenging*, linearly scanning all regions which might contain a pointer to an obsolete object, evacuating any obsolete object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

Figure [3]

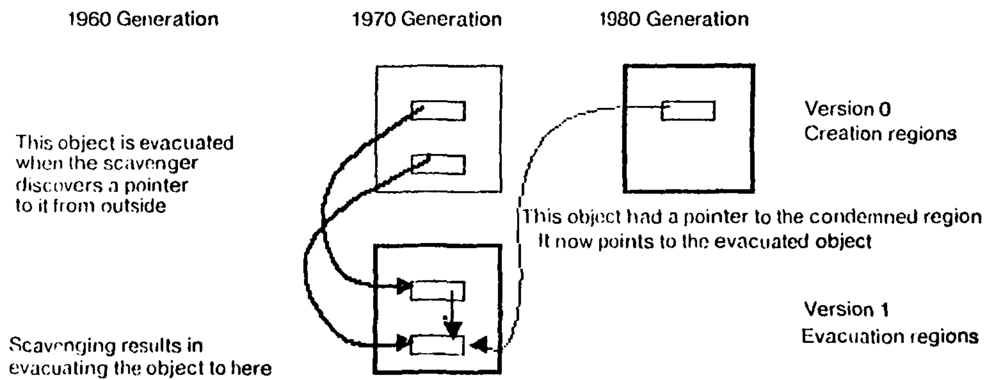
(See next page)

Figure 3

Evacuation and Scavenging



When we encounter a reference to a condemned region we evacuate the object



Scavenging removes pointers to condemned regions
The memory for the condemned region can now be recycled

Scavenging is potentially a lot of work, and since our algorithm is designed to condemn regions at a much faster rate than Baker 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 which 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 which 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 older generations, we will try to engineer a scheme which 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 forward only one generation.* This property will be maintained as an *invariant* throughout the operation of the system. By restricting pointers from older generations to newer generations, we are assured that references to a region will come from either the same generation, the previous generation, or from younger generations. Thus, when a region is condemned, we need not scavenge any generations older than the immediately previous generation. 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? If an old object wants to point to a member of the younger generation, we will require that the old object must become part of the younger generation! Operations like RPLACA must check to see if they might cause an older object to point to a younger object. If this is the case, *the older object is evacuated*, moving the old object into the same generation as the younger object. Then the older object can point to the younger object without creating a pointer forward across a generation gap.

When we evacuate the old object, we must leave a forwarding pointer from the old object's former memory location to its new home. We can't allow these forwarding

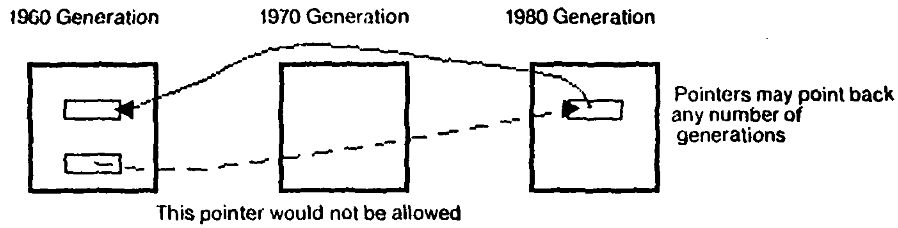
pointers to extend across more than one generation, since that would violate our invariant. So, instead of evacuating the object to its new home all at once, the evacuation is performed *one generation at a time*. First, the object is evacuated into an evacuation region of the immediately following generation, then to the next, and so on until the destination generation is reached. This creates a chain of forwarding pointers between the generations. Following the forwarding pointers when the old object is referenced will take some time, but the forwarding pointers will disappear as the regions containing them are eventually condemned, and we expect forward pointers to be relatively rare.

Figure [4]

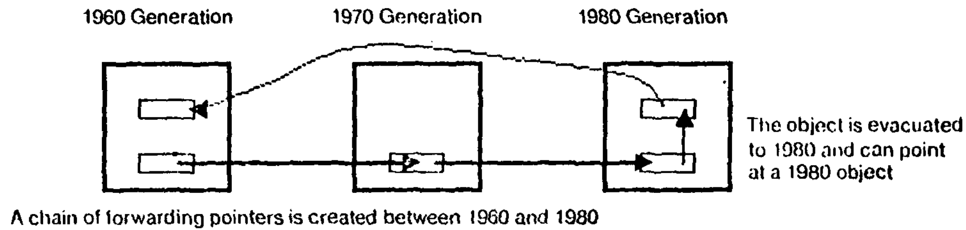
(See next page)

Pointers may point forward only one generation

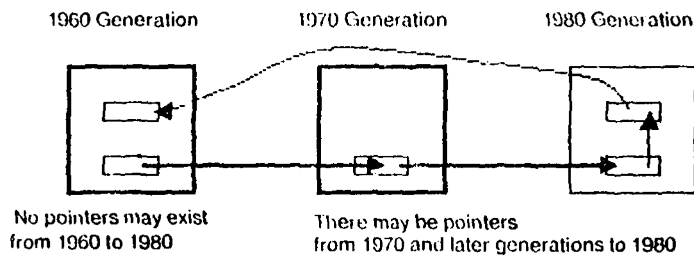
Figure 4



An object in 1960 may not point directly to an object in 1980



The old object is evacuated in steps to the new generation



When 1980 is condemned, 1970 is scavenged, but not 1960

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, 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 to point forward only one generation at a time 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, that 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 it.

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 which 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. So our scheme can be thought of as *renting memory*, where the cost is proportional to the time it is used, as opposed to traditional methods which are like *buying memory*, since the cost is constant.

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 thru many generations. This would reduce paging time necessary to bring in all the pages between a very old generation and the present generation.

6. Scavenging may not be necessary at all!

The most radical solution to the scavenging problem is to avoid scavenging entirely! This might actually be a serious proposal. There are several reasons which may make this both feasible and desirable.

Let's examine the reasons for performing scavenging in the first place. A primary reason for scavenging is to be able to *re-use the address space*. (Note that *re-using 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 re-use addresses that 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 re-use space on the disk, or compacting the storage on the disk may result in reduced disk access time.

It is possible that the evolution of the next generation of computer systems may change the characteristics of systems so as to reduce or eliminate the need for scavenging. Of course, prediction of these trends is highly speculative, but we would like call attention to some of the possibilities. Very large address spaces may obviate the need for re-use 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

re-use address space [7]. Write-once media such as video disks may be used for secondary storage, so that re-using or compacting secondary storage space does not become an issue. Under circumstances such as these, the cost of scavenging may exceed the benefits obtained.

Furthermore, several recently developed languages for artificial intelligence research have the property that they would not benefit from scavenging at all. The languages have the property that no objects in the language can ever be reclaimed! Current implementations of new pattern directed invocation languages like AMORD or ETHER do not have any operations which completely *remove* or *let go* of assertions in the data base. Once an assertion is there, it remains forever, though belief in the assertion may be renounced by further processing. Description languages such as KRL or OMEGA currently suffer from this problem 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. Future machines which use these languages for large projects would want to disable scavenging, since the scavenger would operate to no avail. Some means of getting to any piece of knowledge in the system would always remain.

7. Stacks need special consideration

In presenting our garbage collection algorithms above, 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*, *registers*, and free (*special*) variables which may also reference objects. We must consider object references which reside in these places as well as those stored in object memory.

The stack, registers, and value cells must be scavenged for pointers to obsolete objects before the memory for a condemned region may be recovered. Conceptually, we will consider the stack, registers and value cells to always be a part of the current generation, regardless of when they were actually created.

The stack can be scavenged incrementally starting from the bottom, doing a little bit of the job on each object creation operation. Between object creation operations, the stack may change. The scavenger always remembers where it left off, and when it resumes scanning, it checks the pointer to the top of the stack to see if the stack

has been popped past the place where it remembered. If so, the job is done, since we're only interested in *currently* accessible objects.

In a system which uses more than one stack to implement coroutines or multiprocessing, such as [9] we recommend that each stack have its own set of current regions for creating objects, to improve paging behaviour. Another suggestion which might help performance is to notice that the lifetime of temporary storage is *approximately* (though not exactly!) correlated with pushing and popping the stack. This suggests that a good time to expect there will be 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 it becomes 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 re-usable 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 [15]. Object oriented languages do not follow a stack discipline, and we would like temporary storage in these languages to be efficient.

There's 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 on 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. Value cells need special consideration

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. Value cells are troublesome to our scheme because the memory for a region cannot be recycled until all the references to it which may reside in value cells are removed. (Alternative *deep binding* or *lexical binding* implementations of Lisp store values in data objects called *environments*, and are not subject to this problem.)

We would like to avoid having to scavenge all the value cells. There are several ways this can be accomplished. One possibility we recommend is to keep a data structure which holds, for each generation number, a list of all the value cells that refer to it. Having this data structure makes it easy to update all the references from value cells. The scavenger just asks for all value cells which point to that generation, and evacuates them exactly as for references from the heap.

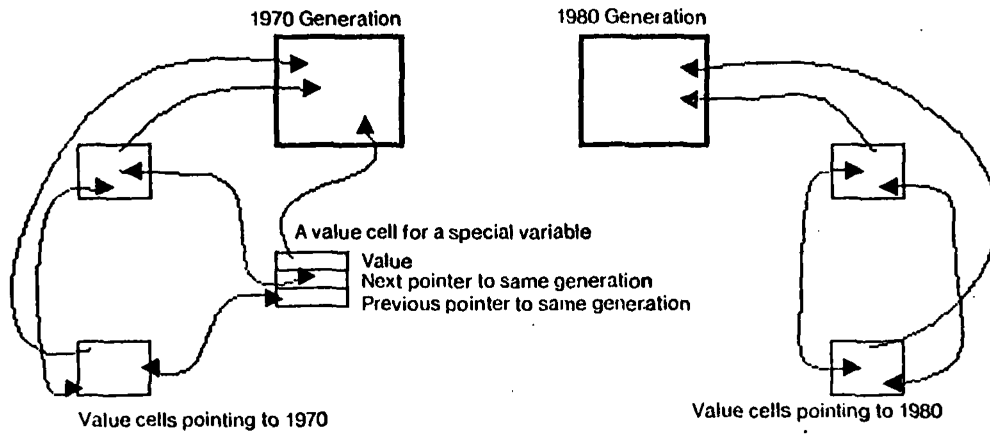
A special trick makes it easy to maintain this data structure. It can be kept in the form of a *doubly-linked circular list.* Each symbol will be given, in addition to a value cell, two more cells, one to point forward in the circle, to the next value cell pointing to the same generation, one cell pointing to the previous value cell which points to the same generation. When an assignment statement changes the value of a cell, we check to see if the generation of the new value is the same as the generation of the old value. If the generations are the same, no special action need be taken. If they're different, the value cell must by *spliced out* of the circular list of the generation of the old value, and *spliced in* to the list corresponding to the generation of the new value. This takes just a handful of memory references. Note that the list can be maintained without doing any CONSing at all!

Figure [5]

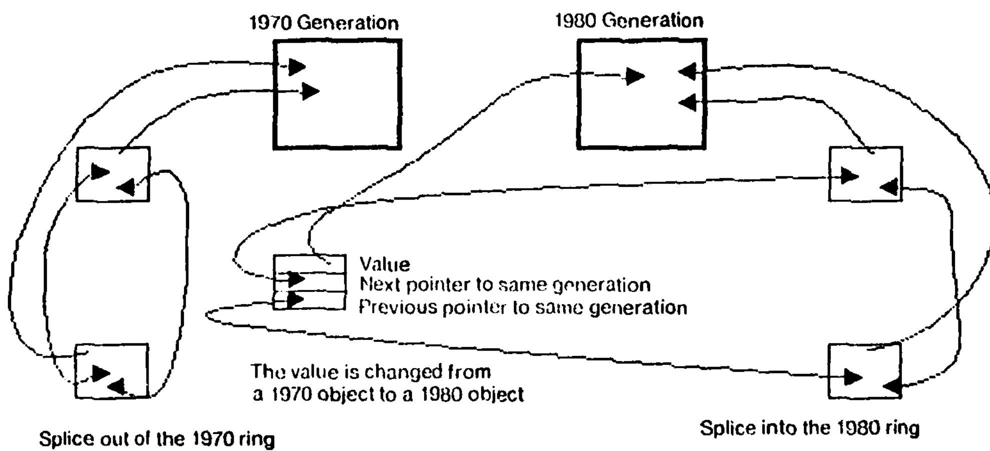
(See next page)

Figure 5

Keeping Track Of Value Cells Pointing To Each Generation



Doubly linked rings of value cells kept for each generation



When value cells are changed, the rings must be updated

This trick for monitoring entry of pointers into generations from value cells also provides another possible solution to the problem of limiting forward pointers to reduce scavenging. Whenever a modifying operation attempts to create a pointer from an older generation to a younger one, we can enter the older object in the circular list containing references to the younger generation, just as if a value cell pointed to the young object. When scavenging for pointers to the younger generation, we need only consult the list, which will contain all extant pointers from earlier generations.

To enter an object in the list requires that two cells be allocated the first time the object is modified to contain a forward pointer, since the list should be linked in both directions. To avoid having to create room for these pointers in every modifiable object, another level of indirection can be used. The old object can contain an invisible pointer to a group of cells containing the pointer to the young object, and the cells which link it with others which point to the same generation. Pointers from the circular lists to objects must be *weak pointers*, pointers which don't protect the objects they point to from garbage collection. If an old region is condemned, there should be a mechanism which removes all pointers which point from that region to younger generations from the entry lists for those generations.

This may make scavenging easier, but at the cost of decreasing locality and complicating the algorithm. It does remove the need to evacuate objects which contain forward pointers to the generation to which they point. Another possibility is to consider evacuating in the other direction, moving the younger object and everything it points to into the older generation containing the object that points to it. This might be done on the theory that a old object pointing to a newer one will increase the chances that the newer one will remain accessible, and should therefore be garbage collected less frequently.

9. How good is the performance of our garbage collector?

Judging garbage collection algorithms is tricky. They 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 which will be prevalent in the next couple of years, and the needs of large-scale AI software.

We consider a crucial aspect of the performance of any garbage collection scheme to be its *paging* behaviour. With the new personal computers such as our Lisp

Machine, the limiting factor in performance is often the time spent paging. Although the size of main memory in computer systems is increasing, virtual memory address spaces are increasing at a much faster rate. (Consider the VAX's address space of 32 bits, hundreds of times larger than that of the previous generation. Main memory sizes have increased only by several times over the previous generation.) Since the next generation of machines will have large address spaces, garbage collection for the purpose of increasing *locality of reference* may be more important than for the purpose of re-using address space. Locality of reference is also very important in systems which have a high-speed *cache* for memory references.

For a rough estimate of the benefits of increasing locality, consider that processing a page fault on the Lisp Machine takes 60 milliseconds. If we estimate the cost of doing scavenging for a single object at about 10 microseconds, then we can scavenge 6000 objects in the time it takes for a single page fault. Since the page size on our machine is 256 words, this says that it would be worth scavenging 24 pages, if increasing locality of reference could avoid even a single page fault by doing so!

These considerations encourage *compacting* garbage collectors such as ours and Baker's, which continually copy accessible objects, grouping them together and leaving behind the garbage. We claim that our algorithm will perform much better than Baker's in promoting locality. Since Baker's fromspace and tospace are large, scavenging may chase pointers anywhere and cause thrashing. Our regions are much smaller, so scavenging will cause less paging. We expect that current regions and regions of recent vintage will usually be paged in, so there should seldom be a need to go to secondary storage.

It is our opinion that sophisticated AI programs will make increasing use of temporary storage, so we have designed our algorithm to optimize the use of temporary storage. Programs which do a lot of internal thinking will need lots of temporary storage 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 in helping to make decisions has been served. If a large proportion of objects which are created are eventually lost, garbage collectors which *trace the accessible objects* will be preferred to alternatives like reference counts, which *trace the inaccessible objects*.

Consider what will happen in our proposal if most of the objects are in fact temporary, and become inaccessible soon after they are created. Shortly after a region fills up, it is condemned. If it contains mostly temporary storage which has become inaccessible by the time it is condemned, then there are only a few accessible

objects to be evacuated. Since the region was created a short time ago, there will be at most a few more recent generations that need to be scavenged, so the scavenger will finish quickly. Then we can recycle the memory for the original region!

The region to which the objects are moved will probably be condemned again soon after the first operation, which will recover anything which became garbage since the first time. As the region ages, the condemnation operations will come less and less frequently. If the rate of recovery of memory by the scavenger is roughly equal to the rate of object creation, we will do very little paging.

What happens if nearly all the data turns out to be permanent instead of temporary? Initially, almost all the objects will be evacuated, and very little memory will be recovered as a result. However, the objects that survive many condemnations will only again be evacuated once in a great while. It might be worthwhile to keep track of what proportion of objects were evacuated from a region until its reclaimed, and use that as a guide to decide how long to wait to condemn that generation the next time. The theory is that generations containing a high proportion of accessible objects will continue to do so.

In Baker's scheme, if we let go of a temporary object, it might take a long time to recover the storage for the object, since we must wait for a flip, which is a relatively rare event. Since Baker's scheme doesn't distinguish between temporary and permanent objects, a lot of wasted time will be spent moving permanent objects again and again, with no memory recovered as a result. We speed up the rate of garbage collection to get temporary objects back faster and slow it down to avoid moving permanent objects.

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 decide between short term and long term memory

Often, a sophisticated user is in a position to know whether a particular object is likely to be temporary or more permanent. The system should be able to take advantage of such knowledge to improve the performance of the 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. Temporary 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, more permanent 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.

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

11. Parallelism can be used to speed up garbage collection

Since processors are continually getting cheaper, an attractive way to improve the performance of garbage collection would be to have additional processors which could perform garbage collection continually while the user's program is running. Our proposals for garbage collection allow this to be done.

A simple way to exploit parallelism would be to use one other processor to act as the scavenger. While the user's program is running, the scavenger could be constantly evacuating accessible objects from obsolete areas of memory into areas currently being used.

Whenever parallelism is introduced, care must be taken to avoid timing errors. The major potential trouble spot with our scheme occurs when objects are being evacuated by the scavenger. There may be transient states where the pointers are inconsistent. So, it is necessary to *lock out* the user processor from accessing objects while they are being evacuated by the scavenger processor. If we have only one scavenger processor, it can only be evacuating one object at a time. Thus, the scavenger can keep the addresses of memory cells in motion in special registers. The user processor must check the address of a cell whenever it is being accessed. If that cell is being used by the scavenger, the user processor must wait, but only for

the few moments that it will take the scavenger to complete the move. The address of cells being accessed by the user must be checked anyway to see if the cell is in an obsolete area of memory, so an additional check will not be expensive. It should be possible to build this check into the hardware.

12. Cheaper short term memory may improve programming style

It's our hope that making the use of temporary storage cheaper will lead to improvements in program clarity. Often, complications in program structure are motivated by the need to avoid creating temporary storage for intermediate results.

Here's an example of how the cost of temporary storage 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.

```
(1 2)     (5 6)     (19 22)
(   )  x  (   )  =  (     )
(3 4)     (7 8)     (43 50)
```

This example would be represented as

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

Let's imagine, that as part of our mathematics library we already have a function which takes the *dot product* of vectors, and a function which produ · 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 an appendix.)

This solution has a potential efficiency problem: The TRANSPOSE function creates a new list which is thrown away after the matrices are multiplied.

In a conventional Lisp system, using lists as temporary storage 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 want 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, the need to avoid using temporary storage encourages more complex and obscure techniques. This example is an illustration of a general situation where an N-pass procedure will use temporary storage for the output of intermediate passes. There is a temptation to substitute a one-pass procedure to avoid using temporary storage, 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.

Our aim is to make the use of temporary storage more efficient, so that the creation of temporary objects is not much worse than allocating temporary results on a stack. If programmers aren't severely penalized in terms of efficiency for choosing cleaner programming styles, we hope this will encourage programmers to improve their style.

Acknowledgments

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 noticing that our scheme for value cells also provides another solution to the problem of forward pointers, and for suggesting several plausible alternatives to specific aspects of our proposals.

We would like to thank Hal Abelson, Tom Knight, Jack Holloway, Kenneth Kahn,

L. Peter Deutsch, Jonl White, Alan Bawden, Henry Baker, Danny Hillis, William Kornfeld, and Marc LeBrun for their helpful comments on this paper.

Appendix 1. 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. MODIFY performs assignments to components of objects, like Lisp's RPLACA and RPLACD.

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 there's no more room in the CREATION area of TOSPACE,
  do a FLIP, exchanging FROMSPACE and TOSPACE.
 Call the SCAVENGER to perform a bit of the work to reclaim memory.
 Fill the memory for the object with its INITIAL-CONTENTS.
 Advance the pointer to the end of the CREATION area past the new object.
 Return the pointer to the new object.

Define ACCESS the contents of an OBJECT:
 If it's in TOSPACE, just return it.
 If it's in FROMSPACE,
  Check to see if the OBJECT contains a FORWARDING-ADDRESS.
   If so, change the OBJECT's contents to where the FORWARDING-ADDRESS points,
   and return the object in TOSPACE.
  If it's in FROMSPACE and there's no FORWARDING-ADDRESS,
   EVACUATE it from FROMSPACE to TOSPACE.
   Update the contents of 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. A FORWARDING-ADDRESS is left behind so references to it will still work.

```
Define EVACUATE an OBJECT:
 If the OBJECT is in FROMSPACE,
  Copy the object into the EVACUATION area of TOSPACE, creating a NEW-OBJECT.
  Leave a FORWARDING-ADDRESS in the old cell to the NEW-OBJECT.
```

The SCAVENGER makes sure all objects in tospace also have their contents in tospace. There's a variable SCAVENGER-SLICE which controls how much work in reclaiming storage is performed every time an object is created. There's a pointer SCAVENGE-HERE which 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 its contents, moving it from FROMSPACE to TOSPACE.
  Advance the SCAVENGE-HERE pointer.
```

Appendix 2. 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 begins moving the accessible objects out of the region, scavenging to remove pointers to it, so that the memory for the region can be recycled.

```
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 RECYCLE to incrementally perform some of the work to reclaim memory.
 If the POPULATION of the current generation is high enough,
  Start a new GENERATION by
  Incrementing the CURRENT-GENERATION-NUMBER.
 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 RECYCLE:
 [The process of recycling memory is interleaved with creation of objects.]
 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.
 RECYCLE 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 the contents of an OBJECT:
 If the object is OBSOLETE [resides in a CONDEMNED region],
  Check to see if the OBJECT contains a FORWARDING-ADDRESS.
   [which means the OBJECT has already been evacuated.]
   If so, change the OBJECT's contents to
    where the FORWARDING-ADDRESS points
   and return that forwarded object.
  If it's OBSOLETE, and there's no FORWARDING-ADDRESS,
   EVACUATE it.
    Update the contents of OBJECT to point to the EVACUATED object,
    and return that new object.
 If the object is not OBSOLETE, just return it.

Define EVACUATE an OLD-OBJECT:
 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.
 Leave a FORWARDING-ADDRESS 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,
   the immediately previous GENERATION,
   and all more recent GENERATIONS up to the current GENERATION:
  For each OBJECT in each REGION:
  SCAVENGE the OBJECT,
   looking for pointers to CONDEMNED-REGION.

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

Define the procedure to MODIFY an OBJECT to have a NEW-CONTENTS:
 If the NEW-CONTENTS is OBSOLETE, EVACUATE it.
 Is the GENERATION of the NEW-CONTENTS
   younger than the GENERATION of the OBJECT?
  If it is, bring the OBJECT into NEW-CONTENTS's GENERATION, by
   EVACUATING the OBJECT to the next generation, then the next,
   until the GENERATION of NEW-CONTENTS is reached.
   This creates a chain of forwarding pointers between the generations.
 Store the NEW-CONTENTS in the memory location of the OBJECT.
```

Appendix 3. 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 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)))))
```

Bibliography

[1] Henry Baker, List Processing in Real Time on a Serial Computer, Communications of the ACM

[2] Henry Baker, Actor Systems for Real Time Computation, MIT Lab for Computer Science report TR-197

[3] Henry Lieberman, A Preview of Act 1, in "Society Models of Intelligence", Luc Steels, ed., forthcoming 1980

[4] Carl Hewitt, Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence,

[5] Dan Ingalls, The Small Talk 76 Programming System: Design and Implementation, Fifth ACM Conference on Principles of Programming Languages, 1978

[6] Edsger Dijkstra, Leslie Lamport, et al, On The Fly Garbage Collection: An Exercise in Co-operation, Communications of the ACM, November 1978

[7] Jonl White, Memory Management in a Gigantic Lisp Environment, or GC Considered Harmful,

[8] David Moon, MacLisp Reference Manual, MIT Lab for Computer Science report

[9] Daniel Weinreb, David Moon, Lisp Machine Manual, MIT Artificial Intelligence Lab report, 1978

[10] MIT Lisp Machine Group, Lisp Machine Progress Report, MIT Artificial Intelligence Lab memo

[11] L. Peter Deutsch, Daniel Bobrow, An Efficient, Incremental, Automatic Garbage Collector, Communications of the ACM, Sept. 1976

[12] Alan Snyder, An Object-Oriented Machine Architecture, MIT LCS PhD thesis

[13] Jon Allen, Anatomy of Lisp, McGraw Hill 1979

[14] Peter Bishop, Garbage Collection in a Very Large Address Space, MIT Lab for Computer Science report TR-178

[15] Joel Moses, The Function of Function in Lisp,