Garbage Collection and Fast Allocation for CMU Common Lisp

Christopher Hoover

June 3, 1990

1. Introduction

Most modern programming languages provide facilities for programmers to perform dynamic memory allocation. Dynamic memory allocation may be broadly classified as either stack-based or heap-based. In the case of stack-based allocation, allocation occurs on the runtime control stack and the storage is usually deallocated automatically when the function or procedure which performed the allocation returns. Stack-based allocation is generally used for small amounts of temporary storage since, in most systems, the stack is relatively small.

Heap-based dynamic allocation is typically used far more frequently and for much larger data structures. In Pascal and The C Programming Language, heap-based dynamic memory allocation is performed by using a language primitive (`new`) or a library routine (`malloc`). In a typical Common Lisp implementation on stock hardware, almost all allocation is heap-based – calls to functions such as `CONS` and `MAKE-ARRAY` cause storage to be allocated from the heap.

Heap-allocated storage may be deallocated explicitly by the programmer or implicitly by the run time system. C, Pascal, and most other Algol-like languages use the former method. In such languages, the programmer is burdened with the task of explicitly calling deallocation primitives (*e.g.* `new`) or routines (*e.g.* `free`) with any block of storage the program no longer needs or fear running out of memory. Further, the programmer must ensure that no other pointers in the program refer to the block of storage. Performing explicit deallocation can be non-trivial in a large program with complex data structures and it is often performed incorrectly or incompletely in such programs. When performed incorrectly, the bugs created may be insidious as the side effects of incorrect deallocation are often subtle and can have many unrelated manifestations. It can be quite difficult to locate and fix such bugs. When performed incompletely, a storage "leak" occurs which may cause the program to perform poorly or run out of memory and crash.

The alternative to explicit heap deallocation is implicit deallocation by the runtime system. In such a scheme, the runtime system, not the programmer, is responsible for detecting blocks of storage that can no longer be accessed by the user's program and allowing those blocks to be reused. This is often known colloquially as "garbage collection." Garbage collection frees the programmer to solve the problem at hand [35] and not get bogged down in the details of storage reclamation.

This paper primarily focuses on garbage collection in Lisp systems but the techniques and algorithms described within are applicable to a wide variety of programming languages and environments.

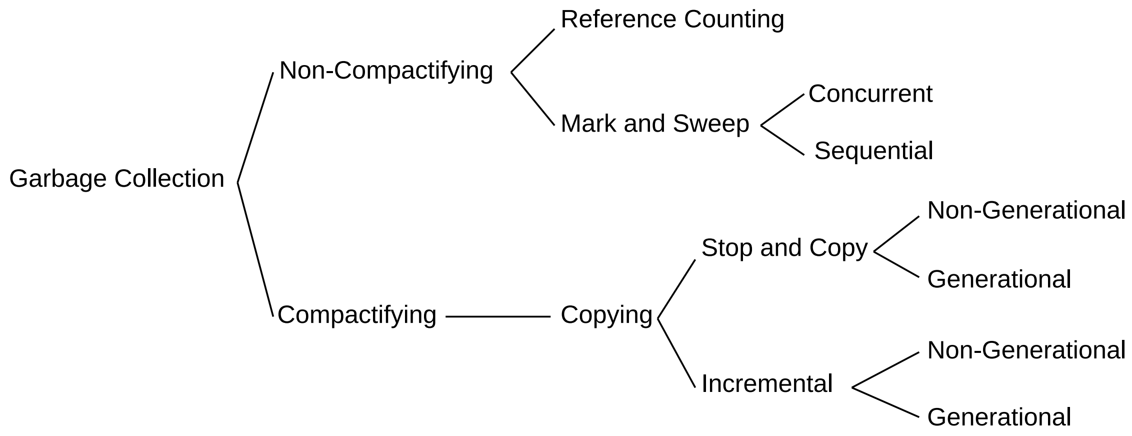


Figure 1: Taxonomy of Garbage Collection Techniques

2. Garbage Collection

Garbage collection may be implemented using a number of different techniques. These techniques may be broadly separated into two categories: compactifying and non-compactifying. Figure 1 shows a taxonomy of these techniques. Non-compactifying garbage collection is discussed first.

2.1. Non-Compactifying Garbage Collection

A non-compactifying garbage collector makes no attempt to separate the garbage from the rest of the storage and compact the live storage into a contiguous space. Instead, the system maintains a list of free regions in the heap and allocation is performed by choosing one of these free regions. The two most common non-compactifying garbage collection techniques are reference counting and "mark and sweep."

2.1.1. Reference Counting

Reference counting [15] refers to the technique whereby all cells allocated in the system contain a special field called a reference count. The reference count is an integer which is equal to the total number of pointers referring to that cell. Creating a new pointer to a cell causes that cell's reference count to be incremented. Likewise, when a pointer is modified, the reference count of the original referent of the pointer is decremented. Thus, a reference count of zero implies that there are no external references to the cell and that the cell is garbage. At this point, the cell is returned to the system's free list and any pointers within the cell are overwritten with the null pointer possibly freeing additional cells.

Reference counting is referred to as an incremental garbage collection technique [35] since storage reclamation occurs immediately when a cell is freed. A variation on this technique, due to Weizenbaum [42], provides a smoother incremental response than the method described above by lazily handling free cells. In this variation, when a cell is returned to the end of the free list,

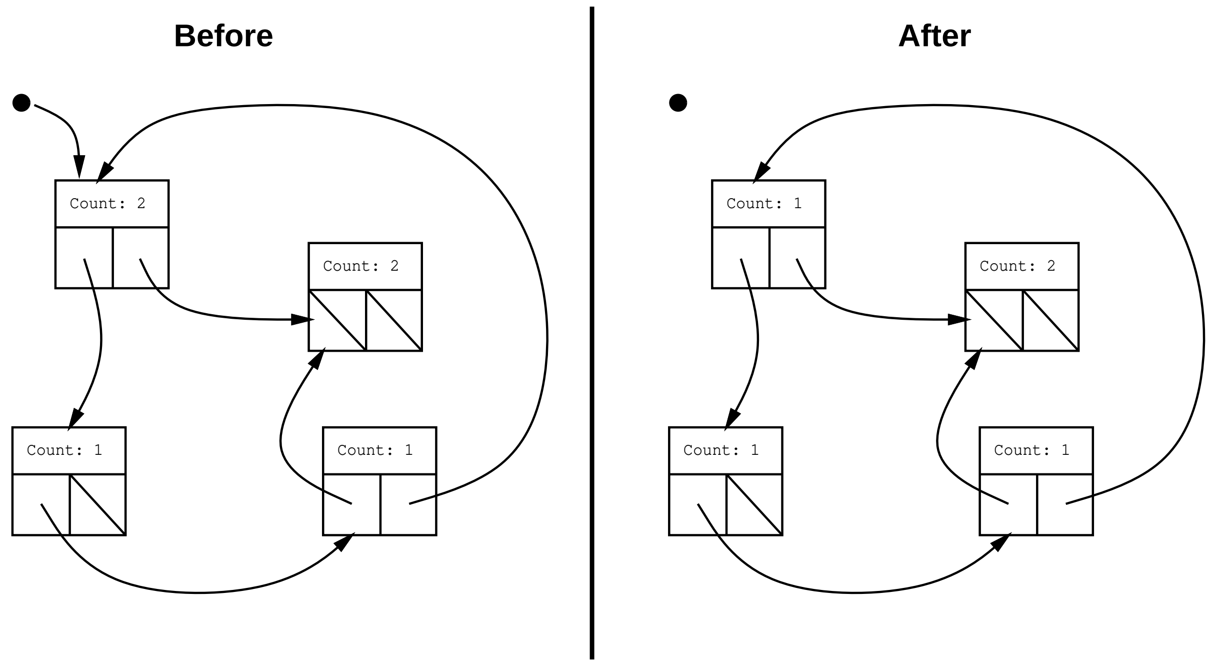


Figure 2: Reference Counting: An Unreclaimable Cyclic Structure

its internal pointers are not processed; rather, before the cell is allocated again, all of its internal pointers are set to null. This delays the reference count processing for any cells referred to in the original cell until allocation time.

Reference counting is used in many systems, but it suffers from several major drawbacks:

1. Necessity of a reference counting field for every object.

   The need for a reference count field for every cell allocated can greatly increase the memory requirements for the heap – note that it is crucial that the reference count field be made large enough so that it cannot overflow. If an undetected overflow ever occurred, the system might not be able to reclaim free cells and, worse, it might reclaim live cells.

2. Inability to reclaim cycles of unreachable objects.

   The inability to reclaim cycles of unreachable objects is another serious drawback since circular structures are relatively common in Lisp systems [25]. Figure 2 shows an example of a structure which would not be reclaimed by reference counting. Even though a cycle may be unreachable, the cyclic structure ensures that all of the cells in the cycle will have a non-zero reference count.

3. Overhead of maintaining reference counts.

   The overhead of maintaining reference counts can be quite prohibitive. Deutsch and Bobrow [18] have suggested a modification to the standard technique to reduce the overhead involved. Their modification involves lazily updating the reference count for cells. Instead of continuously updating reference counts for cells, they record a set of interesting "transactions" – cell allocations, pointer creations, and pointer deletions. At suitable intervals, they process these transactions to update the reference counts. However, the base cost of reference counting remains expensive; they suggest using a separate processor to handle the overhead of processing the transactions.

2.1.2. Mark and Sweep

Mark and sweep is another non-compactifying garbage collection scheme which was first introduced in 1960 [28]. There are many variations of mark and sweep but the basic idea is simple. The "mark" phase of the algorithm consists of traversing all accessible storage and marking it as "in use." This is generally accomplished by performing a depth-first search of a set of root pointers – in a typical Lisp system these are the machine registers and the runtime stack. The "sweep" phase of the algorithm consists of examining every record in the heap to see if it has been marked. All unmarked records are placed on the free list and all marked records are reset in preparation for the next collection. This is shown graphically in Figure 3.

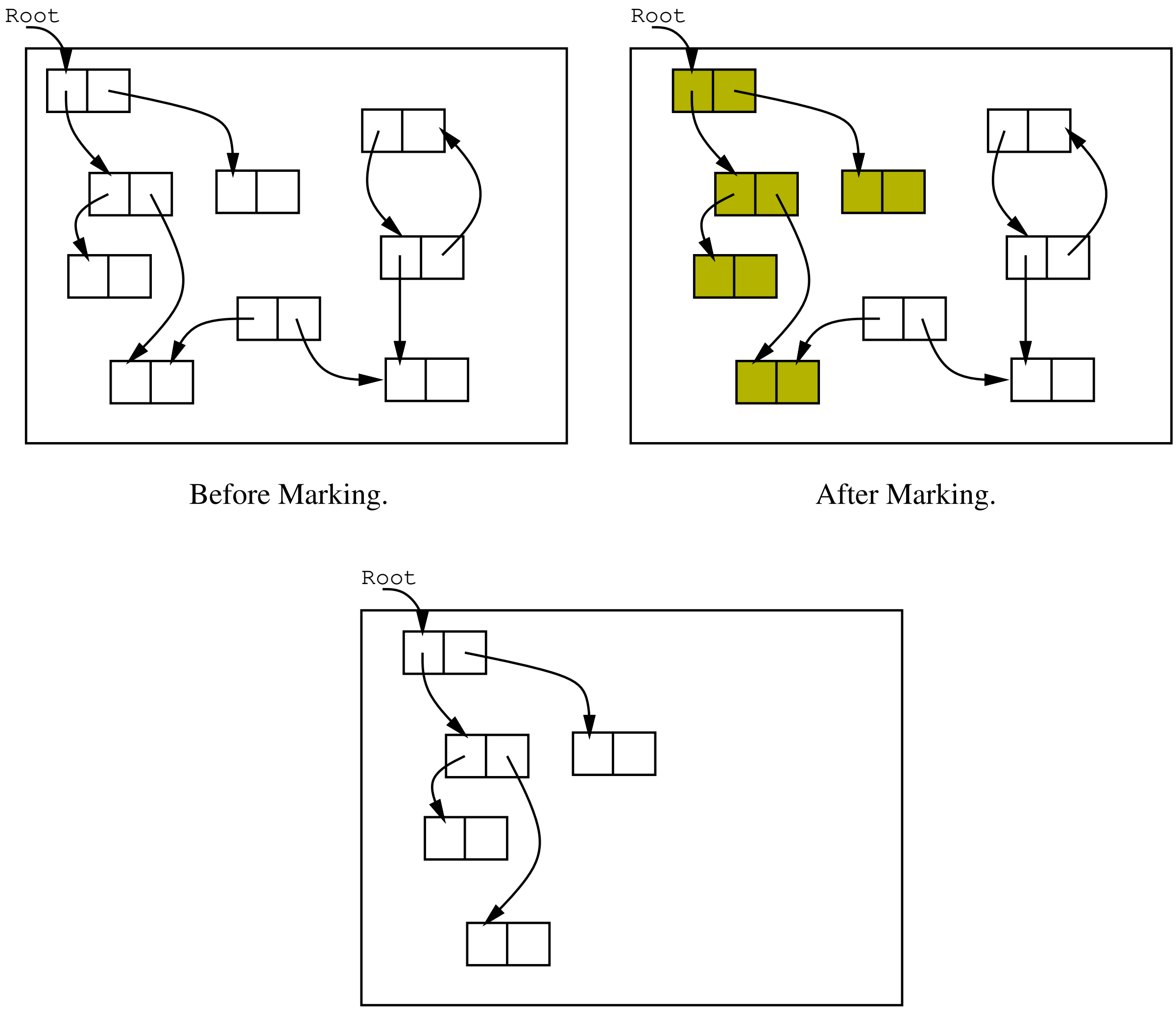
There are a number of algorithms which may be used for marking. It is important that any algorithm under consideration be able to mark all the accessible cells using little more than a marking bit in each cell and constant workspace. Mark and sweep garbage collection is usually invoked only when storage is in short-supply; thus, techniques which do not use a constant amount of workspace are not suitable. Further, it is desirable for the algorithm to run in $\mathcal{O}(n)$ time where n is the number of cells accessible from the roots.

Deutsch-Schorr-Waite marking [24] is one technique which meets these criteria but it requires an additional tag bit per cell. Wegbreit marking [41] is similar (as both use a link-inversion traversal algorithm) but Wegbreit marking does not require a per-cell tag bit; instead, this technique uses an explicit stack for the tag bits. Unfortunately, the space required for this algorithm is unbounded. Lindstrom marking [26] requires no additional tag bits; however this algorithm is $\mathcal{O}(n \log n)$ in time. All of these algorithms are detailed elsewhere. No algorithm is known which runs in $\mathcal{O}(n)$ time which uses no tag bits; consequently, Deutsch-Schorr-Waite marking is usually used.

Mark and sweep suffers from the fact that its runtime is $\mathcal{O}(n + m)$, where n is the number of accessible cells and m is the number of reclaimable cells. Thus, mark and sweep takes time proportional to the amount of storage in use since it always scans all of the heap and touches every cell ever allocated.

Various efforts have been made to make mark-and-sweep garbage collection run concurrently with the mutator[1]. Minsky is credited (by Knuth [24]) for first suggesting the use of a concurrent garbage collector; Steele [36] later improved upon this design. The basic idea involves running the mutator on one processor and the collector on another. The collector continuously performs garbage collection and is always in one of three phases: marking, sweeping, or reallocating. The marking and sweeping phases behave as described above – they walk the heap to find all inaccessible storage. The reallocation phase involves moving objects around in an attempt to do compaction. Wadler [39] has performed an extensive analysis of the time and space complexity of the algorithm. There are several major drawbacks to this method of garbage collection. First, implementation of

[1]The user's program is known as the "mutator."



Before Marking.

After Marking.

After Sweeping.

Figure 3: Mark and Sweep Garbage Collection

the algorithm is quite complex. Further, an external stack is needed for marking. More seriously, the algorithm requires special hardware and thus is not appropriate for stock hardware.

2.1.3. Comments on Non-Compactifying Garbage Collection

Non-compactifying garbage collectors are not particularly useful on modern machines with virtual memory because they suffer from the fact that they inherently do not perform heap compaction. This is problematic because objects in modern Lisps vary in sizes and the internal fragmentation caused by a non-compactifying garbage collection makes allocation difficult and, when severe, impossible. Most implementations of non-compactifying garbage collection have some compaction stage which is invoked when fragmentation becomes severe. As we will see, there are other algorithms which feature better time complexity (than mark and sweep) and which perform compaction during the garbage collection process; thus non-compactifying algorithms will no longer be discussed in this paper.

2.2. Compactifying Garbage Collection

Compactifying garbage collection is preferable for a number of reasons including improved paging performance and the ability to do fast allocation. Compactifying garbage collectors are often called copying garbage collectors since garbage collection is performed simply by moving[2] all accessible storage into a new section of the heap that is not currently in use. After the move is accomplished, the old space is discarded; thus the costly sweep phase is eliminated. Note that if the allocation of copies is performed sequentially in the new space, the collection will make all allocated space contiguous. This scheme is depicted in Figure 4.

2.2.1. Copying Garbage Collection

The simplest implementation of a copying garbage collector divides the heap into two halves which are called semispaces. This idea was first suggested by Fenichel and Yochelson [19]. The mutator works in only one of the two semispaces. When this active semispace becomes full, the garbage collector stops the mutator and copies all of the data accessible from a set of root pointers (as before these are usually the machine registers and the runtime stack) into the other space. The source space is usually called either "fromspace" or "oldspace" and the destination space "tospace" or "newspace." After the copy occurs, the spaces are "flipped" – newspace becomes oldspace and vice versa and oldspace is simply discarded. Finally, the mutator is resumed. Because the mutator does not run during the collection, this type of garbage collection is known as "stop-and-copy" garbage collection.

Basic stop-and-copy garbage collection has several nice features. The first and foremost is simplicity. Further, stop-and-copy garbage collection is efficient. When enough memory

[2] The distinction between moving and copying here is important. When moving a list, the original copy may be destroyed; when copying a list, it may not be. The term "copying" garbage collector is, to some extent, a misnomer.

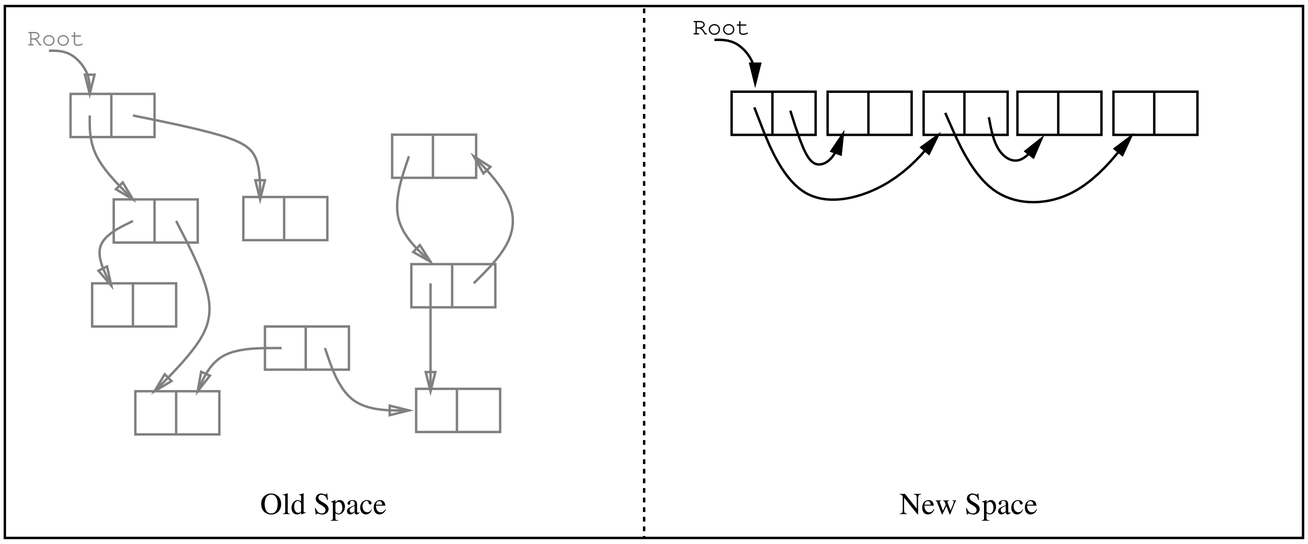


Figure 4: Copying Garbage Collection

is available, heap allocation with a stop-and-copy garbage collection may be faster than stack allocation [3]. The major disadvantage of stop-and-copy garbage collection is that it can cause long pauses in the mutator. This may be disruptive in some applications. The need for semispaces[3] is also a disadvantage as it requires doubling the size of the heap; however, this is not a significant concern on systems with virtual memory as only one of the semispaces is in use most of the time.

Now, we detail the operation of the copy algorithm. Most copying garbage collector use some variation of Cheney's algorithm [11]. Cheney's algorithm is a non-recursive algorithm for moving arbitrary linked-list structures. The only requirement of Cheney's algorithm is the existence of a primitive, *new*(p), which can differentiate between pointers into fromspace and newspace[4].

The algorithm is quite simple. There are two basic operations: scavenging and transporting. Scavenging refers to the operation of replacing a pointer into fromspace with a pointer into newspace. It is defined, in pidgin C, in Figure 5. Transporting refers to the operation of copying an object from oldspace to newspace. An object is only copied once – after the transport operation copies an object, it leaves a forwarding pointer in the original object which points to the new copy. The transport operation always checks for the presence of a forwarding pointer (using the *new* primitive) before copying a cell. This operation is defined in Figure 6.

To copy all accessible storage, `scavenge` is applied to all of the root pointers. This causes cells to be copied into newspace by `transport` and all of the root pointers to be updated to point at the new copies. To complete the copy operation, `scavenge` is applied successively to every pointer in newspace. Cheney's algorithm effectively copies the heap in breadth-first order.

---

[3]There are sliding-window, copying garbage collection algorithms which do not require semispaces. They are very complex and beyond the scope of this paper.

[4]The primitive is used to distinguish between normal pointers and forwarding pointers. In a Lisp which uses low tags and semispaces, it may be implemented simply as a bounds check. In a Lisp which uses high tags, a special forwarding pointer type would be needed.

```
scavenge(location)
object_t **location;
{
        /* if the pointer in location does not point to an object in
           new space, then let transport return a pointer to newspace
           for it. */

        if (!new(*location))
                *location = transport(*location);
}
```

Figure 5: Definition of `scavenge`

```
object_t *
transport(object)
object_t *object;
{
        /* check to see if the object contains a forwarding pointer.
           if it does, it has already been transported into new space
           and transport simply needs to return this pointer. */

        if (new(*object))
                return *object;

        /* there is no forwarding pointer so the object must be copied
           into newspace. */

        /* copy_object allocates space in new space, copies the object
           pointed to by its argument, and returns a pointer to the
           copy. */

        where = copy_object(object);

        /* need to leave behind a forwarding pointer. */
        *object = where;

        return where;
}
```

Figure 6: Defintion of `transport`

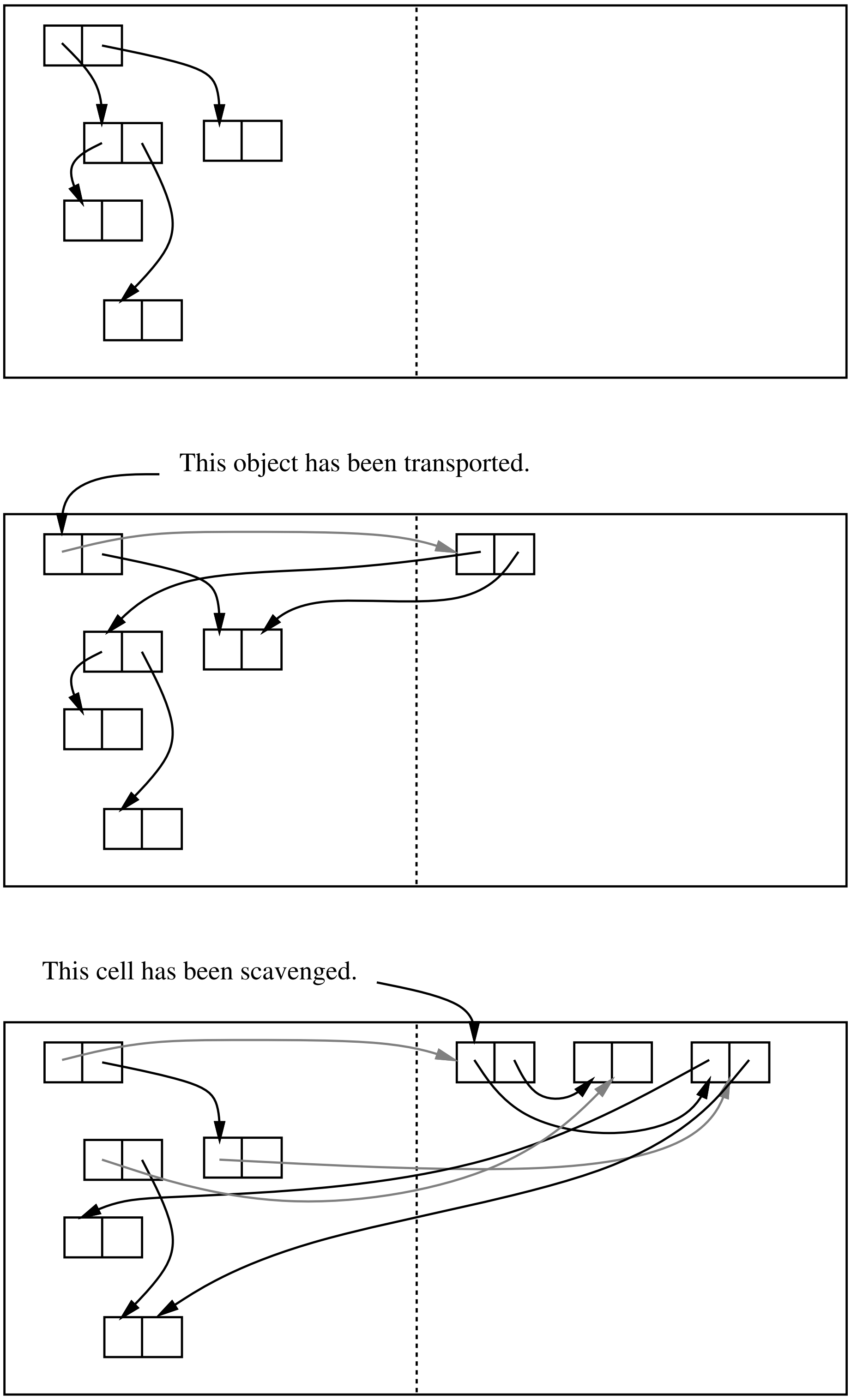


Figure 7: Cheney's Algorithm

Figure 7 is useful in visualizing the operation of the algorithm. The black arrows represent normal pointers whereas the gray arrows represent forwarding pointers. The top diagram shows the initial state of the heap. The middle diagram shows the state of the heap after one of the cells has been transported (possibly because one of the root pointers pointed at that cell and was scavenged). The bottom picture shows the state of the heap after the transported cell has been scavenged.

While Cheney's algorithm takes constant workspace and runs in linear time, for circular data structures, Cheney's algorithm visits each cell twice. Clark's algorithm [12] is similar to Cheney's algorithm in that it uses constant workspace, runs in linear time, and requires a *new*(p) primitive; however, Clark's algorithm only revisits cells that must be revisited (i.e., they contain pointers). According to some empirical data [13], on average, only 25% of the cells in typical Lisp lists would be revisited using Clark's algorithm. Clark's algorithm is used in the T 3.0 garbage collector although Rees [31] believes that this was a mistake as Clark's algorithm, while linear in time, has a much larger constant factor associated with it than Cheney's algorithm.

Unlike a mark and sweep garbage collector, a copying garbage collector based on Cheney's or Clark's algorithm runs in time $\mathcal{O}(n)$ where n is the number of accessible cells. A rule of thumb from computer architecture [21] indicates that memory consumption for the typical program (and consequently the size of the typical computer memory) grows by a factor of 1.5 to 2 every year. It is likely that memory consumption for the average Lisp program grows even faster than this rate. Thus, it becomes increasingly important that as memories grow, the time complexity of a garbage collector be linear in the amount of retained storage and independent of the total amount of allocated storage.

2.2.2. Incremental Copying Garbage Collection

The goal of incremental garbage collection is to allow the garbage collector to run either concurrently or as a co-routine with the mutator. This is particularly desirable in interactive applications since it can eliminate large mutator pauses which are often annoying to the users of these applications. It is less useful in computation-intensive applications since the overhead of incremental garbage collection on stock hardware is, in general, more costly than the standard stop-and-copy style garbage collection.

Baker's real-time garbage collector [6] was the first copying garbage collector to interleave the work of the scavenger and the mutator. As before, the heap is divided into two semispaces and the mutator runs in one of the two spaces. When that space fills up, a flip occurs and the garbage collector is invoked to scavenge only the root pointers. The important difference in Baker's garbage collector is that the mutator is now resumed and every time a new object is allocated, the scavenger is invoked to scavenge a few words of memory[5]. In order to make this interleaved execution work correctly, it is important that the mutator never see a pointer into oldspace. In order to prevent this from happening, whenever the mutator attempts to dereference a pointer referring to oldspace, some mechanism must stop the mutator, transport the referent from oldspace to newspace, update the pointer, and, finally, resume the mutator.

[5]The number of words scavenged is chosen so that memory will never be exhausted.

Unfortunately, such a mechanism is difficult to implement efficiently in both time and space without special hardware since every pointer dereference must be checked before it is performed. Brooks [10] offers a solution for stock hardware that costs only an extra word per object, an extra indirection per object reference, and additional complexity when modifying object; however, this is still an expensive price to pay for incremental garbage collection.

Appel, Ellis, and Li [5] have suggested a different way to perform incremental garbage collection on stock hardware that utilizes virtual memory protection. In their scheme, the heap is divided into two semispaces and a garbage collection is performed by flipping the two spaces and scavenging all of the root pointers. However, instead of scavenging newspace next as a stop-and-copy garbage collector would, the unscanned section of newspace is simply protected against read access from the mutator and the mutator is resumed. If the mutator attempts to dereference a pointer in any unscanned area of newspace, a fault occurs. A fault handler (running concurrently on another processor in their implementation) then locates the page the mutator tried to reference, unprotects it, and scavenges it. When not handling page faults, the separate processor concurrently scans any unscanned area of newspace. The Appel, Ellis, Li algorithm is elegant and can be made to run efficiently on a multiprocessor with sufficient operating system support; however, the algorithm would incur a large overhead on a stock uniprocessor. One difficulty with the algorithm, which would particularly degrade performance on a uniprocessor, is that page faults may occur in dense clusters. In the worst case, a fault would occur with every dereference and faulting alone would perform a complete garbage collection. Unfortunately, such a pathological garbage collection would take longer than if a stop-and-copy garbage collector had been used due to the significant overhead of fault handling.

2.2.3. Paging Issues for Copying Garbage Collectors

Heap management using a copying garbage collector has an interesting advantage over most other heap management techniques – objects may be *moved*. Thus, a copying garbage collector can improve the virtual memory paging performance of a program by increasing the locality of reference of objects in the heap. By placing objects which are often accessed together onto the same virtual page, the rate of virtual memory page faults may be reduced.

Incremental garbage collectors can reorganize the heap with very good locality since they transport objects from oldspace to newspace in the order they are touched by the mutator. This places all of the active objects in the same area of the heap. It is important, however, that the background scavenger in such a garbage collector place the objects it transports into a separate area of the heap; otherwise the set of active objects will become diluted with inactive objects thus degrading locality.

As discussed earlier, incremental garbage collection can be expensive on stock uniprocessors so it is usually the case that stop-and-copying garbage collectors are employed. Recall that most stop-and-copy garbage collectors are based on some variation of Cheney's algorithm and, further, that Cheney's algorithm performs a breadth-first copy of the heap. This is unfortunate as a breadth-first copy of the heap will generally yield poorer locality than a depth-first copy as a depth-first copy will tend to place the children of an object on the same page as the object itself [30].

Luckily, it is possible to modify Cheney's algorithm to approximate a depth-first copy and Moon's ephemeral garbage collector [30] for the Symbolics 3600 does exactly this. Instead of scavenging newspace linearly from start to end after all of the root pointers have been scavenged, Moon's variant *first* scans any partially-filled page at the end of newspace. If no partially-filled page exists at the end of newspace or if no references to oldspace exist on that page, then scavenging resumes linearly from the last address scavenged until either a new object is transported (in which case there is a now a new partially-filled page) or until the end of newspace is reached (in which case the garbage collection is now complete). If a new partially-filled page is created during the linear scan, the scavenger will always scavenge it *before* returning to its linear scan.

A similar variation due to MacLachlan [27] is to effectively scan newspace backwards. In this scheme, the last n words of newspace are scavenged in reverse order *before* scanning newspace linearly from the start. If a new object is transported during the backwards scan, the backward scan starts over at the end of newspace. If no references to oldspace exists in the last n words of newspace, scavenging resumes linearly from the last address scavenged until either a new object is transported or until the end of newspace is reached (in which case the garbage collection is now complete). When an object is transported during a forward scan, the scavenger will always start a new backward scan *before* returning to its linear scan.

Both of these variants do only slightly more work than the original version of Cheney's algorithm. While they will in some cases attempt to scavenge locations in newspace multiple times, the total number of objects transported will always be the same. Transporting objects is much more expensive than scanning so the total increase in runtime will be relatively small. Thus, for a small price, these variations may greatly improve paging performance.

A more straightforward optimization to Cheney's algorithm for Lisp is to perform `CDR` *linearization*. `CDR` linearization refers to the placement of all of the `CONS` cells in a list into consecutive locations in memory. Most Lisp programs use lists as data structures and many use lists even when arrays are more suitable. Thus, it is important that accessing list elements and, more importantly, iterating over lists be implemented as efficiently as possible. `CDR` linearization can be a substantial benefit since it can dramatically reduce paging during these operations. For example, consider the typical `CONS` cell which is 8 bytes long and the typical virtual memory page which is 4096 bytes long. By simple division, it is clear that 512 `CONS` cells fit on a single page. In the worst case, without `CDR` linearization, iterating over a list of length n would cause n page faults; with `CDR` linearization at most $\lceil \frac{n}{512} \rceil$ faults would occur. The implementation of Cheney's algorithm with `CDR` linearization is straightforward and it may be implemented in conjunction with either of the other two locality improving optimizations mentioned above.

3. Generational Garbage Collection

Stop-and-copy garbage collection is an efficient technique for managing a heap [3]; however, a stop-and-copy garbage collector may cause unacceptable disruptive pauses in the mutator as the garbage collector must halt the mutator for the duration of each garbage collection. Fortunately, a simple variation known as generational garbage collection can make most of these pauses relatively short.

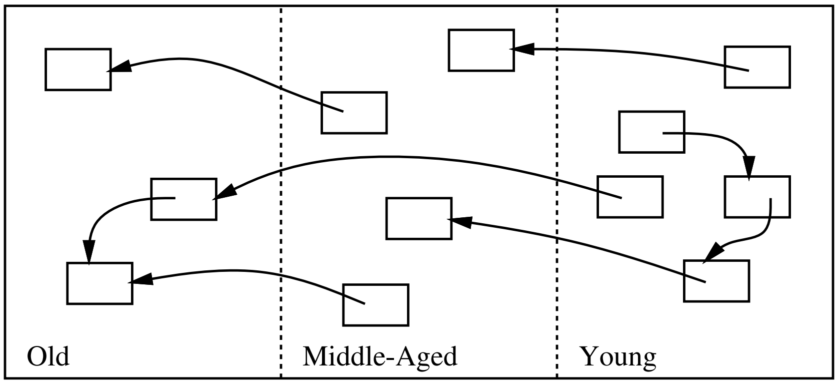


Figure 8: Layout of Generations

Generational garbage collections take advantage of an empirical observation about Lisp programs and the objects they manipulate: an object's age is a good predictor of its lifetime. Young objects tend to have short lifetimes and old objects tend to have long lifetimes. This is not surprising – Lisp is primarily a functional language and temporary results are often created and discarded during a computation. Further, Lisp programs tend to use heap allocation in places where stack allocation would be used in other programming language. Consequently, much of the heap contains short-lived objects which quickly become garbage. The few objects in the heap which have survived for a while are likely part of permanent or semi-permanent data structures and, in general, will continue to exist for a long time.

The difference between a generational garbage collector and all of the garbage collectors presented thus far is that a general garbage collector incorporates some notion of object lifetime [25]. In the simplest scheme, the heap is divided into a number of distinct regions known as generations and each generation contains objects of approximately the same age.

Figure 8 shows how this might look with three generations. Assume for now that, as in the figure, all pointers within a given generation refer only to objects in that or older generations; this invariant will be relaxed later in Section 3.1. If this does hold, notice that it is possible for the garbage collector to scavenge younger generations without scavenging the older ones. This is crucial to generational garbage collection as it allows the garbage collector to concentrate its efforts on the areas of the heap that are most likely to contain garbage.

New objects are created in the youngest generation which is scavenged frequently (relative to the older generations). The youngest generation is scavenged frequently since it is the area of the heap that is most likely to contain garbage. Recall that the work done by a copying garbage collector based on Cheney's algorithm does work proportional to the number of live objects copied. Since few objects should survive from the youngest generation, the garbage collector does little work scavenging it. If an object does survive a number of scavenges in a generation, it is advanced to the next older generation which is subject to less frequent scavenges since it contains somewhat older data and fills more slowly than its younger sibling. By concentrating effort on the areas of the heap that are most likely to contain garbage, the garbage collection becomes more efficient.

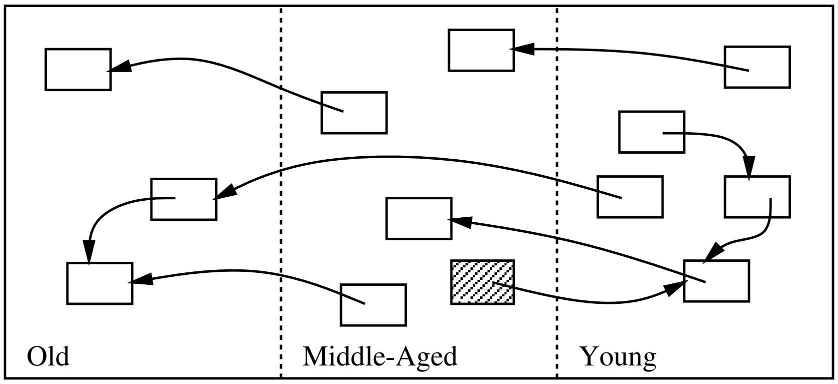


Figure 9: A Forward Pointer in A Generational Scheme

3.1. Allowing for Intergenerational References

As stated previously, the ability to perform generational scavenging depends on maintaining the invariant that all generations contain only backward pointers, i.e., pointers to objects in the generation itself or in older generations. This invariant may be relaxed if all the pointers in older generations which refer to objects in younger generations may be identified. The highlighted cell in Figure 9 is such a pointer. The garbage collector must treat it as an explicit root pointer when scavenging the "young" generation.

If assignments which create forward references occur frequently in a system, then the cost of detecting and handling these forward pointers will be prohibitive and will outweigh the relative benefits of using a generational garbage collector. In Lisp systems, such a reference can only be created by destructive operations such as `RPLACA`[6]. Fortunately, in a mostly-functional programming language such as Lisp, programs do not usually use many destructive operations and thus do not create a large number of forward references.

Lieberman and Hewitt [25], who first proposed generational garbage collection, solve the forward pointer problem by entering all forward pointers into a per-generation table of indirections known as an "entry" table. Thus, forward pointers do not point directly at objects in a younger generation – rather they point at an entry in the generation's entry table which then points at the object. When a generation is scavenged, it is necessary to scavenge the generation itself and all younger generations as well as all of the entries in the entry tables for those generations.

While this scheme seems simple and straightforward, it has several major drawbacks. First, every assignment must be checked to see if it would create a forward pointer. If it would, the entry table for the referent's space needs to be updated properly. Second, every pointer indirection must be checked to see if it first points into an entry table. If it does, a second indirection must be made to obtain the desired object. Last, these assignment tables may become quite large and they, too, need to be garbage collected. These drawbacks along with the special needs of Baker's algorithm make it prohibitive to implement this algorithm on stock hardware.

[6] Allocation cannot create forward references since a new object can only refer to older objects.

Ungar [37] solves the forward pointer problem in a different manner. Instead of using indirection tables, Ungar simply records all assignments from older objects to younger objects in a "remembered set." The remembered set (along with the root set) is scavenged during every garbage collection and all entries in the remembered set which are still forward references are retained. As an optimization, all assignments may be recorded unconditionally and the overhead of checking for forward references deferred until scavenge time [37, 33, 47]. Handling forward references in this manner may be implemented efficiently on stock hardware although, as with the assignment tables in Lieberman and Hewitt's solution, these lists of forward pointers may become quite large.

Another variation on Ungar's solution solves this last problem by dividing the heap into small units called pages or cards[7] and recording into a bitmap all of the pages or cards into which assignments have been made [30, 46]. This technique is known as "marking" and it can dramatically decrease the amount of storage necessary for the remembered set; however, the area of the heap corresponding to each marked page or card must now be scanned during a scavenge to find forward pointers.

The tradeoffs between page marking and card marking can be complex. Page marking may be implemented very efficiently if virtual memory hardware is used to detect assignments into the heap [30, 33]. The primary disadvantage of VM-assisted page marking is that the unit of marking is very large. This translates into large scanning costs at scavenge time since every marked page will cause many[8] locations in the heap to be scanned for forward pointers.

Card marking does not suffer from this disadvantage since card sizes are software-defined and can be made much smaller than the average virtual memory page. Wilson and Moher [46] cite the use of card marking with a small card size of 32 bytes for this reason among others. Card marking on stock hardware, however, must be performed by software. The cost of card marking in a compiled Lisp system has been estimated [33] to be around six percent in both runtime and code space as software card marking requires the addition of about six instructions to every heap assignment.

With any marking scheme, it must be possible to identify all of the valid Lisp pointers on a page or a card while scanning. "Unboxed" slots of objects – slots which contain raw bits which might be mistaken for pointers – must be avoided. Fortunately, there are several solutions. If objects are either entirely boxed or unboxed, i.e. they either contain only Lisp pointers or only raw bits, a simple solution is to maintain two heaps – one boxed and one unboxed. The unboxed heap will never need to be marked (and thus never need to be scanned) since it cannot possibly contain references to other objects. While this scheme has its merits, the use of a dual heap (each of which may contain a number of generations) requires a very large heap and can complicate the implementation of a garbage collector.

Another solution is to arrange for the system to be able to identify and interpret all of the objects on a page or card. This may be accomplished by placing a tagged header word at the beginning of any object which contains unboxed data and arranging for all pages or cards to begin either in the

[7] A page generally corresponds to a page in the hardware's virtual memory system while a card corresponds to a smaller, software-defined division of the heap.

[8] In a typical system, 1024 pointers would have to be examined per page.

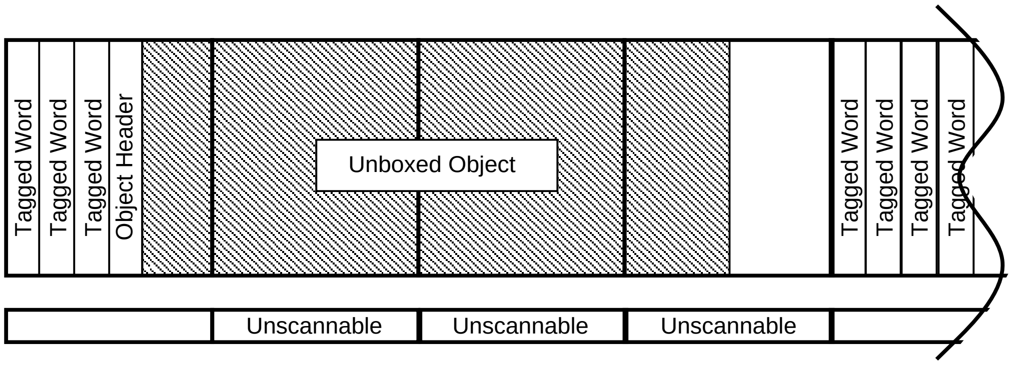


Figure 10: Scanning Large Unboxed Objects

middle of a boxed object or at the start of an object with a header. A page or card may be scanned by starting at the beginning of the card or page and stepping through it. Every word will be a tagged object – either a pointer, an object header, or some other datum. When an object header is found it must be interpreted to find any boxed slots and to find the end of the object (and subsequently the start of the next object) on the page or card.

Unfortunately, these restrictions on how objects may be placed on a page or a card make it impossible for an object containing unboxed data to be larger than the unit of scanning. These restrictions may be relaxed by utilizing a "crossing map" [5] or an "unscannable set" [46]. Both schemes are similar – they simply indicate which pages contain unboxed data and thus cannot be scanned arbitrarily. Scanning a page or card begins by first checking to see if it is scannable. If it is, it is scanned as before. If it is not, scanning begins by backing up to the first scannable page or card and scanning forward from the last object header on that page or card. Figure 10 illustrates how a large unboxed object might be laid out in conjunction with an unscannable set. Note that for large unboxed objects, it is sometimes necessary to pad the last page or card out to the next page or card boundary. If the space wasted due to padding becomes significant, the garbage collector could reduce the fragmentation by moving other unboxed objects into the free space.

3.2. Advancement Policies, Generations, and Heap Layout

Two important parameters in a generational garbage collector are the advancement policy and the number of generations. The advancement policy (sometimes called the "tenuring" policy) is the policy whereby objects surviving a specific number of scavenges become promoted to older generations. Advancement polices may be strict, adaptive, or mutator-controlled, or a combination of those. Various advancement policies may dictate different heap layouts. There are many complex tradeoffs here and many complicated interactions. This section attempts to briefly survey some of these policies.

Moon's Ephemeral Garbage Collector [30] uses a very simple advancement policy. At every scavenge, all data are advanced. The chief advantage of such an advancement policy is that objects are not repeatedly copied within a single generation. The chief disadvantage of such a simple policy is poor age precision – data in a generation may have been created immediately before the last scavenge or immediately before the scavenge prior to the last. In the first case, the data may have been advancement prematurely – before given a chance to die. Further, the only way to increase the advancement threshold for a generation is to increase the size of the generation.

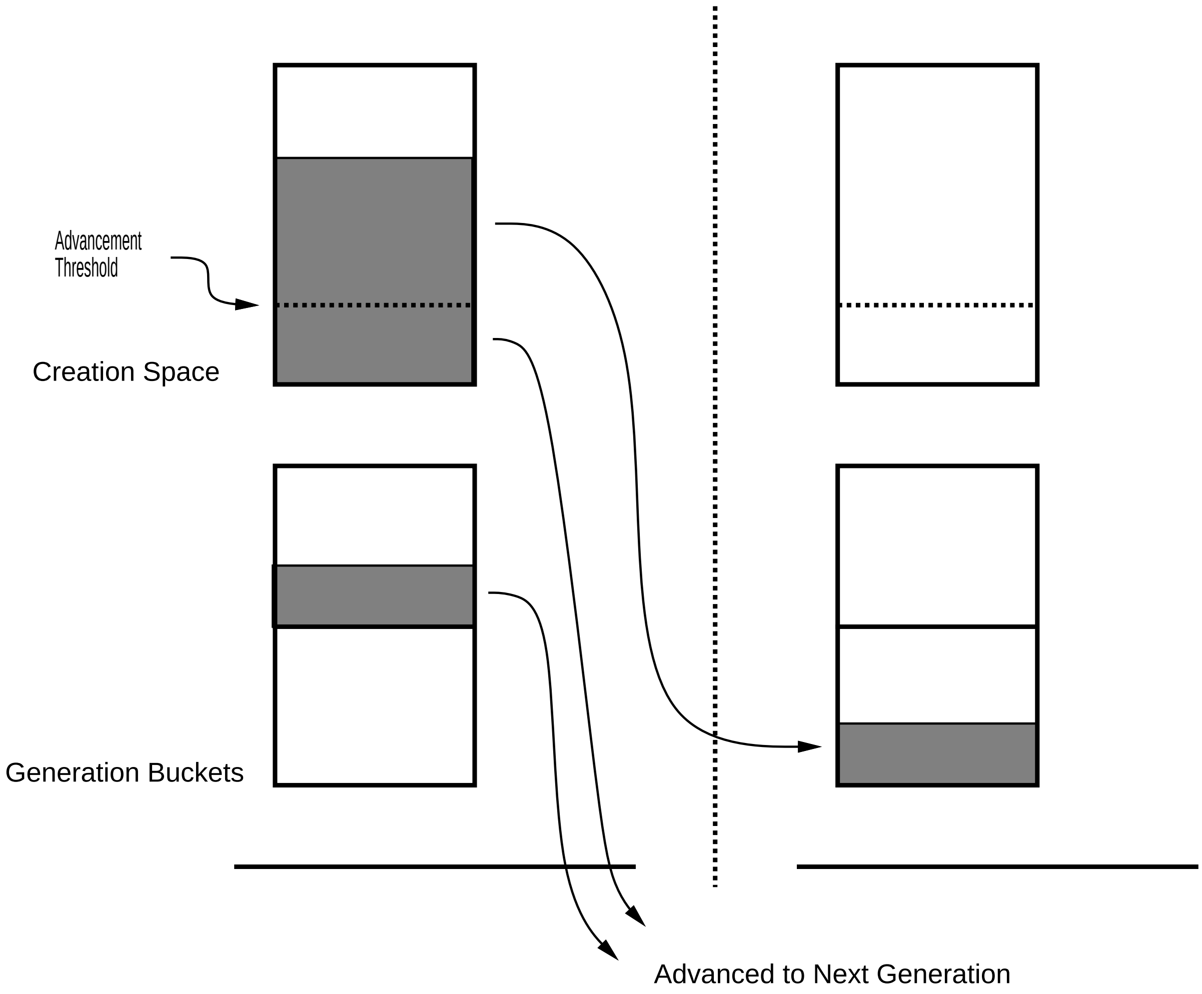
Ungar [37] achieves improved age precision by associating a scavenge count with every object in the heap. The scavenge count is incremented every time an object survives a scavenge. Objects advance to the next generation when their scavenge count exceeds a specific threshold. Ungar uses three generations – a creation generation, a survivor generations, and an old generation. Objects are always advanced from the creation generation to the survivor generation on the first scavenge. Objects stay in the survivor generation and are recycled between the semispaces that compose the survivor generation until they survive enough scavenges. At that point, they are promoted to the old generation which is not subject to online scavenging. Ungar's solution gives excellent age precision but is expensive on stock hardware due to the need to store and manipulate per-object scavenge counts.

Shaw [33] has suggested the use of a bucket brigade mechanism to achieve good age precision without the need for a per-object scavenge count. In Shaw's scheme, each generation is a set of buckets holding objects of approximately the same age. Objects are usually transferred from bucket to bucket on each scavenge but they may also be returned to their original bucket during special non-transferring scavenges. Each bucket must consist of a pair of semispaces since objects may be copied out of and into a bucket simultaneously. Objects in the last bucket of a generation are advanced to the next generation during a transferring scavenge and are placed in the first bucket in that generation. By varying the number of buckets and the ratio of transferring to non-transferring scavenges, the advancement policy may be varied over a wide range while still maintaining good age precision.

Wilson [45] and Wilson and Moher [47] have devised a solution based on Shaw's system which is simpler but still allows for reasonable control of the advancement policy. They claim that empirical evidence indicates that a two-scavenge advancement threshold is sufficient in most case (and more desirable than Moon's one-scavenge threshold) and that a large increase of the threshold beyond two scavenges is necessary to yield further benefits. This is fortunate as a two-scavenge threshold is easy to implement by dividing each generation into two semispaces each of which acts as a bucket. One of the semispaces acts as a newspace for the data being copied in from the previous generation while the other acts as a fromspace for data being copied out to the next generation. Allocation of new data in a generation may be facilitated by having a separate creation space for that generation [9] which is completely emptied when that generation is scavenged. This layout is illustrated in Figure 11.

Wilson's heap organization and advancement policy strikes a nice balance between maintaining good age precision and allowing for easy control of the advancement policy. Notice that the data

[9]If memory is scarce or virtual memory support is poor, it is not necessary to have a separate creation space [45, 47]. Instead data may be allocated directly into the bucket containing the data retained from the last scavenge and the division between the retained data and newly allocated data recorded. When a scavenge occurs, everything below the dividing line represents retained data and those data are advanced to the next generation; everything above the dividing line represents newly allocated data and those data are retained by copying them into the other bucket.



The figure shows a generation before (left) and after (right) a scavenge.

Figure 11: Layout of Generations in Wilson's Garbage Collector

in the creation space are ordered chronologically by age in memory. By advancing some fraction of these newly allocated objects, the advancement policy may be varied continuously from Moon's one-scavenge threshold (if all of the newly allocated objects are advanced immediately) to a full two-scavenge threshold (if none of the newly allocated objects are advanced immediately) all the while maintaining good age precision. Further, note that this flexible control makes it easy to implement an adaptive advancement policy.

4. CMU Common Lisp

This section describes work in progress for the new port of CMU Common Lisp to the DECStation 3100 using MacLachlan's Python compiler. This section assumes only a minimal knowledge of the new object format [14] used in this port.

4.1. Stop-and-Copy Heap Organization

The current organization of the heap in the new port of CMU Common Lisp consists of four distinct spaces:

- Read-Only Space

  Read-only space is occupied by objects which are permanent, are never moved, are never reclaimed, and are never modified. Further, objects in read-only space may refer only to objects in read-only space or in static space. This makes it unnecessary for the garbage collector to scavenge read-only space.

- Static Space

  Static space is occupied by objects which are permanent, are never moved, and are never reclaimed. They never need to be transported by the garbage collector.

- Dynamic-0 Space

- Dynamic-1 Space

  Dynamic-0 space and Dynamic-1 space are the active areas of the heap. At any given time, one of these spaces acts as newspace and the other as oldspace.

4.2. Fast Allocation

Object allocation in the new port is very fast. One machine register, ALLOC, always contains the free pointer for the current dynamic space. By simply incrementing this register, allocation may be performed inline. Currently, most fixed-length objects are allocated and initialized inline by the compiler. For example, the function `CONS` is always coded inline and it only takes 10 MIPS R2000 instructions to allocate *and* initialize the new cell from arguments in registers. The code sequence for `CONS` is shown in Figure 12; it will be explained in more detail later.

Allocation of some variable-length objects is also performed inline when the compiler has enough information. In particular, allocation and initialization of lists using `LIST` or `LIST*` for a known number of arguments is coded inline with a number of instructions proportional to the number of arguments. The allocation of vectors and arrays, however, is not coded inline directly; rather, for a number of common cases, an inline call to one of three small assembly routines performs the allocation. The other less common cases of vector and array allocation require the intervention of Lisp-level code to first select an appropriate representation.

```
        and     $FLAGS, $FLAGS, 65533       ; Clear the INTERRUPTED flag.
        or      $FLAGS, $FLAGS, 1           ; Set the ATOMIC flag.
        addu    $A0, $ALLOC, 3              ; Save the pointer to the CONS.
                                            ; (Add 3 to give it a list tag.)
        addu    $ALLOC, $ALLOC, 8           ; Allocate the CONS (8 bytes).
        sw      $LEXENV, $A0, -3            ; Store the CAR.
        sw      $CNAME, $A0, 1              ; Store the CDR.
        and     $FLAGS, $FLAGS, 65534       ; Clear the ATOMIC flag.
        and     $NL0, $FLAGS, 2             ; Check the INTERRUPTED flag.
        beq     $NL0, $ZERO, L0             ; If not set, go on.
        nop
        break   9                           ; Cause a trap
L0:
```

Figure 12: Code Sequence for `CONS`

4.2.1. Implementing a Garbage Collection Threshold

Heap allocation beyond a certain point must force a garbage collection to prevent the system from running out of memory. This point is called the garbage collection threshold. A problem with the code sequence for `CONS` shown in Figure 12 is that it never checks to see if the allocation pointer has exceeded the garbage collection threshold.

Fortunately, this is not necessary as it would require more instructions and slow down the allocation sequence. To implement a garbage collection threshold, we use virtual memory primitives (as exported by the MACH operating system [7]) to make the virtual memory page starting at the garbage collection threshold unwritable. This page is known as a "guard" page and writing to it will cause the operating system to raise an exception [9] which can be handled by user-level code.

To make this work, each of the allocation primitives must follow a special discipline (which may be seen in Figure 12):

1. Clear the INTERRUPTED flag.
2. Set the ATOMIC flag.
3. Perform the allocation.
4. Initialize the new object.
5. Clear the ATOMIC flag.
6. Check the INTERRUPTED flag. If set, cause a trap.

This discipline is necessary so that the exception handler can detect the case when the allocation pointer has exceeded the garbage collection threshold and allow the allocation to complete while regaining control afterwards. It is important that the allocation and initialization occur *atomically* as otherwise the heap could be left in an inconsistent state. A heap in such a state could fatally confuse the system and, particularly, the garbage collector.

When the exception handler is invoked by the operating system, it first examines the ATOMIC flag to make sure it is set[10]. The handler then unprotects the page so the allocation may continue and resumes the allocation; however, before it does so, the handler sets the INTERRUPTED flag so that the allocation primitive will trap when it is done. This mechanism allows the allocation to finish while still allowing the system to regain control immediately after the allocation. The trap handler, after examining the state of the system and realizing the exception handler arranged for this trap to occur, calls some Lisp-level code to decide if a garbage collection is really warranted.

This scheme works well for objects which are completely initialized; however, it fails for objects which do not require explicit initialization since the allocation of such objects will not write to the guard page and, consequently, will not cause an exception to be raised. The allocation primitives for such objects follow a different discipline: they simply perform an explicit comparison of the allocation pointer and the start of the guard page after allocating the object and invoke a trap when necessary.

4.3. Stop-and-Copy Garbage Collection

The current garbage collector in the new port of CMU Common Lisp is a copying garbage collector based on Cheney's algorithm. It is written in The C Programming Language for portability. It is nearly operating system independent and the sections of the garbage collector which are operating system dependent have been modularized. Further, the garbage collector is easily extensible as it is completely table driven. Adding new "methods" for scavenging and transporting objects is almost trivial. Emphasis so far has been placed on simplicity, clarity, style, and portability [11], not on performance. The code is presented in the appendix.

4.3.1. Special Considerations

Garbage collectors in real, high-performance Lisp systems are seldom as elegant as the algorithms on which they are based – there are always special considerations which must be addressed.

One such consideration in the new CMU Common Lisp garbage collector involves derived pointers. A derived pointer is a pointer into the middle of an object and such pointers are often created during array or structure accesses. Derived pointers must be handled specially by a garbage collector and they must not be interpreted as normal pointers. Fortunately, there are

---

[10]The ATOMIC flag is always set whenever this exception handler is invoked. It *must* be set since all allocation happens with the ATOMIC flag set and this exception is only raised during allocation. So why is the ATOMIC flag needed? The ATOMIC flag is needed for other exception and interrupt handlers which have no *a priori* knowledge of the state of the system when they are invoked (often asynchronously).

[11]To give an example, the same code with only minor changes has been used to implement a preliminary version of the CMU Common Lisp `PURIFY` function. `PURIFY` is responsible for moving objects out of dynamic space and into static and read-only space. Its primary use is to help build Lisp core images.

several solutions. A simple solution which was adopted for the new port is to compute and store any derived pointer in a special register and to always keep the original base pointer in some other register for the life of its derivative. Since this invariant is maintained, it is trivial for the garbage collector to find the derived pointer (as it is always in the same register) and its base pointer. The base pointer is located by finding the register in the set of valid Lisp registers whose value is closest to the derived pointer and less than or equal to it[12]. This register's index and the offset between the base pointer and the derived pointer are recorded and after all of the registers have been scavenged, the derived pointer is recomputed from the new value of the register and the offset. The program counter requires similar attention if it was in fromspace before the garbage collection as it is essentially a derived pointer of another register in this case.

Another special consideration regards the handling of `EQ` hash tables. The implementation of `EQ` hash tables in CMU Common Lisp uses an object's pointer or its immediate representation as a parameter to the hashing function. Therefore, whenever an object is moved it must be rehashed since moving an object changes its hash key. The garbage collector facilitates this by taking special action when transporting `EQ` hash tables. This special action consists of setting a bit in the hash table indicating that the hash table needs to be rehashed. All of the Lisp-level functions which operate on `EQ` hash tables check this bit before operating on a hash table; if it is set, the Lisp-level code first calls a routine to rehash the hash table.

The garbage collector also needs to have intimate knowledge of how the system handles interrupts. The UNIX operating system saves the state of the world upon an interrupt in a structure known as an interrupt context. The saved state includes among other things information about the interrupt and a copy of the set of machine registers at the time of interrupt. The garbage collector must scavenge all registers in these saved sets that contain Lisp objects during a garbage collection. This is further complicated by the fact that the system may be interrupted while in a non-Lisp state[13] – the garbage collector must be very careful *not* to examine any saved registers from such an interrupt since they will not contain valid Lisp objects.

4.3.2. Extensions

As a simple extension, weak pointers have been added to the CMU Common Lisp system with the help of the garbage collector. A weak pointer is a pointer which may point at an object in the heap but, unlike normal pointers, it does not prevent the object from becoming garbage. Weak pointers are useful for building doubly-linked and persistent data structures. Consider the data structure shown in Figure 13. Suppose the user's program is using some package[14] which has a large persistent data structure. Further, suppose that the user's program then calls some function in that package which creates and returns some object to which the data base refers with pointer P_a (because, for example, it must keep track of the object to update it). Now the user's program has a handle on the object through pointer P_b. Ideally, if the user's program modifies or destroys

---

[12]If the set of valid Lisp pointers is R_i and R_d is the derived pointer, the task is to find $i \mid R_i \leq R_d \wedge R_i > R_j, i \neq j$.

[13]Such a state exists, for example, during a foreign function call.

[14]*Package* here is used to denote a software module rather than the Common Lisp `PACKAGE` data type.

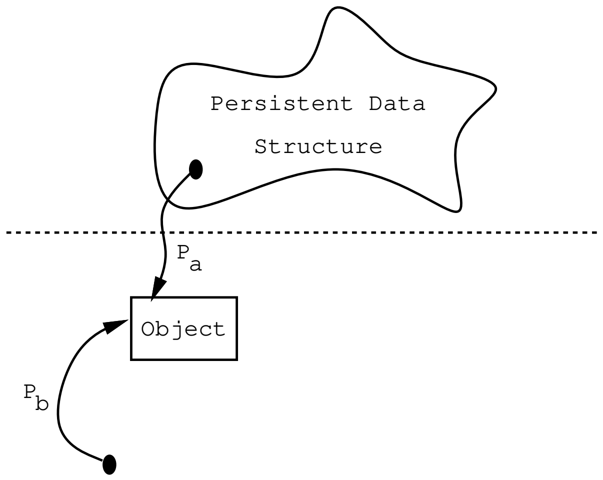


Figure 13: Weak Pointer Example

pointer P_b, the object should become garbage; however, it will not since the object will always be reachable from some root pointer via the persistent data structure. Weak pointers are useful because if a weak pointer is used in place of P_a this problem does not arise, since a weak P_a would *not* cause the object to be retained even though it would be reachable.

The implementation of weak pointers is straightforward. Weak pointers are simply pointers out of objects called weak cells. While scavenging newspace, the garbage collector chains together any weak cells it finds so that it may scan them when it is done; it does *not*, however, scavenge the pointer inside the weak cell. After completing the scavenge of newspace, the garbage collector then scans all of the weak cells it found either updating them or marking them "broken." A weak cell is updated if the weak pointer in the cell points to an object in oldspace and that object contains a forwarding pointer; this indicates that the object was referred to by some other pointer and was copied into new space. The forwarding pointer points to the copy of the cell in new space and this value is placed in the weak cell. If no forwarding pointer exists in the cell, then the weak pointer is marked as invalid. (Note that it would not be difficult to add finalization of weak pointers.)

4.3.3. Future Work

There are a number of areas in which further work is needed. The first is adding more instrumentation. Currently, the instrumentation consists of measuring and reporting elapsed real time, system time, and user time along with the total number of bytes scavenged and the total number of bytes retained. Additional instrumentation needs to be added so that new "improvements" may be evaluated quantitatively on more dimensions. For example, a table listing the frequency of use of all of the scavenging and transporting methods would be extremely informative. A breakdown on the distribution of the size of objects scavenged segregated by object type might also be handy.

With additional instrumentation, it will be relatively easy to improve performance by simply recoding parts of the garbage collector. As mentioned earlier, emphasis so far has been placed on clarity and style. There are known inefficiencies in the code – in particular there are a number of function calls which could easily be eliminated without obfuscating the code to any serious degree.

There are two algorithmic changes which deserve further attention. Both strive to improve post-garbage collection performance of the system by reducing paging. Paging is hopefully reduced by increasing the locality of reference in the heap. The first change is minor. It involves experimenting with the order used to scavenge the pointers in the root set to find the order that typically best improves the locality of reference of objects in the heap. The root set for the new port consists of any valid interrupt contexts, the interrupt handler functions, the control stack, the binding stack, and all of static space.

The other change involves adding one of the pseudo depth-first traversal modifications described in Section 2.2.3 to the garbage collector. While doing so will make garbage collections slower, the potential exists for a large improvement in post-garbage collection performance due to the modification.

4.4. The Design of a Generational Garbage Collector under a Sophisticated Operating System

This section contains the draft of a design for a generational garbage collector for the new port of CMU Common Lisp. Work on this garbage collector is currently in progress.

4.4.1. Operating System Support

This design requires sophisticated operating system support. In particular, flexible virtual memory support is assumed. The MACH operating system [7] has such support, and, consequently, this design specifically addresses the use of the MACH operating system although other operating systems could be used.

4.4.2. Number of Generations

Four generations are used each with a scavenge time, t_s, roughly an order of magnitude larger than the previous:

- Generation 0

  The young generation. Most objects are created in this generation. It flips often and will be sized and scheduled to have a scavenge time $t_s \leq 1$ second.

- Generation 1

  The intermediate generation. The size of this generation will depend on the amount of physical memory available and will be chosen so that paging is not a dominant factor. The scavenge time, t_s, will be tens of seconds.

- Generation 2

  The advanced generation. This generation holds user objects advanced from generation 1; thus, it should fill very slowly. It is only flipped when there is a danger of running out of virtual memory or backing store. The scavenge time, t_s, will be large – tens of minutes.

- Generation 3

  The initial generation. This generation holds objects which are part of the Lisp system itself and, possibly, any user objects which have advanced from generation 2. It is somewhat similar to the static and read-only spaces described in Section 4.1. It is never flipped.

More generations may be necessary for increased throughput.

4.4.3. Heap Organization and Advancement Policy

Each generation is divided into two semispaces (or buckets) and a creation space. When scavenging a generation, one of the semispaces acts as a newspace for objects being copied out of the previous generation while the other acts as a fromspace for objects being copied into the next generation. The creation space is completely emptied upon each scavenge and the data (normally) remains in the same generation.

This results in a two scavenge advancement threshold. By realizing that the data in the creation space are ordered by age and immediately advancing some percentage of that data, the advancement policy may be varied continuously from a one scavenge advancement threshold to a two scavenge advancement threshold. This is essentially the organization proposed by Wilson [45] and Wilson and Moher [47] and it is detailed in Section 3.2. It is a well-balanced solution as it maintains a decent guarantee of object age, allows for a flexible advancement policy, and advances data quickly enough that copying costs are reasonable.

4.4.4. Recording Intergenerational References

A generational garbage collector must be able to find all of the intergenerational references in the heap. In this design, intergenerational references are recorded via page marking using a MACH memory manager [7]. A MACH memory manager is a server which handles paging requests from a MACH kernel for memory objects. Most memory objects are created by the kernel and are handled by the default memory manager that is part of the kernel. However, the MACH virtual memory system allows a user to create specialized memory objects and to supply external, user-level memory managers to manage these objects.

A memory object may be mapped into the address space of any number of tasks; in fact, a task's address space is simply a mapping between contiguous virtual memory addresses and memory objects. The MACH kernel uses physical memory pages as a cache for the contents of mapped memory objects. A memory manager is responsible for supplying data for any page in a mapped memory object when a cache miss (page fault) occurs. The MACH kernel may also decide to flush a page from its cache at any time. This is known as a page out operation and a memory manager is responsible for supplying backing store to handle page outs for all of its memory objects.

External memory managers allow a task to manage its address space in interesting ways. Using an external memory manager, a thread may map a memory object created by the external memory manager into its address space and then access that memory directly in the normal manner. The kernel will call the corresponding external memory manager to handle any paging requests for the mapped memory.

In this design, an external memory manager is used to perform "transparent" page marking. The Lisp external memory manager runs as one of two threads in the Lisp task: the Lisp thread proper and the Lisp memory manager thread. Ranges of virtual memory which require marking are created by allocating a special *with-marking* memory object managed by the Lisp external memory manager and mapping that memory object into the Lisp task's address space. Backing store for each of these memory objects is allocated in the normal manner from the kernel.

When a page fault occurs on a memory object, the kernel sends a *memory object data request* message to the appropriate memory manager requesting the data for the faulted page . The Lisp external memory manager handles these request by simply sending the kernel the corresponding page of backing store. When a page out occurs on a memory object and the paged-out page is *dirty* (i.e. it has been modified), the kernel sends a *memory object data write* message containing the data for that page to the appropriate memory manager. The Lisp external memory manager handles these request by copying the dirty page into the backing store and marking the page for the garbage collector. Thus, page marking in this design is implemented in a manner similar to Shaw's virtual memory page dirty bit marking mechanism; however, page marking in this design is portable[15].

Page marking was chosen because it is cheap, unobtrusive, and simple to implement (under MACH). Unfortunately, with page marking, scavenging performance can suffer dramatically due to the large number of locations which must be scanned for each marked page. This overhead can become excessive – especially if the locality of writes into older generations is poor as many pages will become marked.

It may be necessary to use different strategies for different programs. Some programs might opt to use software card marking (conditional or unconditional) whereas other programs might use page marking exclusively. To make this work, all system code would have to be compiled with software card marking. Programs which used page marking would have to run in a special dynamic environment which would work as described above except that instead of marking dirty pages, all of the cards corresponding to a dirty page would be marked. This scheme would facilitate the flexible use of conditional and unconditional software card marking and page marking in the same Lisp system.

---

[15]That is, it is portable to any machine running the MACH operating system.

4.4.5. Scanning Marked Pages

During a scavenge, all marked pages must be scanned for intergenerational references. This is complicated by the fact that Lisp objects in the heap may contain unboxed data – raw bits that may look like valid Lisp pointers. Several solutions to this problem were outlined in Section 3.1.

The solution adopted for this design is to divide each generation in half and store boxed and unboxed objects separately. This layout is illustrated in Figure 14. Note that the zero generation is *not* segregated since it is never marked and, consequently, never scanned. The novel use of an unsegregated zero generation is interesting because it greatly simplifies allocation (which is discussed further in Section 4.4.7); it may also result in improved paging and cache performance since the density of all of the truly "hot" objects will be greater.

While scavenging the zero generation, the garbage collector will have to sort the advancing boxed objects from the advancing unboxed objects and transport them into the appropriate areas of the next generation. The overhead to do this will be almost negligible since the garbage collector already must determine the type of an object in order to transport it.

Separating the boxed from the unboxed objects not only makes scanning simpler, it makes scanning more efficient too. Since boxed objects contain only valid Lisp objects, marked pages may be scanned *blindly* without trying to interpret the objects or the contents of each page. This should help reduce the extra scavenge-time costs due to scanning which are associated with page marking.

Separating boxed and unboxed objects has another performance advantage. It reduces the total amount of memory which must be page marked since the unboxed section of each generation does not need to marked (since it cannot contain valid Lisp pointers). While the previously described implementation of page marking is fairly efficient, there is still some overhead incurred. Thus, eliminating the need to mark several large sections of memory will improve performance, if only slightly.

Using separate spaces for boxed and unboxed objects does have one major drawback – it could potentially increase the size of the heap by as much as a factor of two. This, however, is not a concern under MACH because MACH implements large, sparse address spaces efficiently.

4.4.6. Code Objects

Code objects are problematic. Code objects may not be placed in either the boxed area or unboxed area of the later generations as code objects contain both boxed and unboxed data. Thus, code objects must be handled specially.

One solution is to place all code objects in special generations and to handle intergenerational references for these generations differently. Two generations should be sufficient as code objects most likely have a bimodal lifetime distribution. Code objects either die off quickly (as is this case with those that are top-level forms) or they live virtually forever (as is the case with those code objects that compose a program).

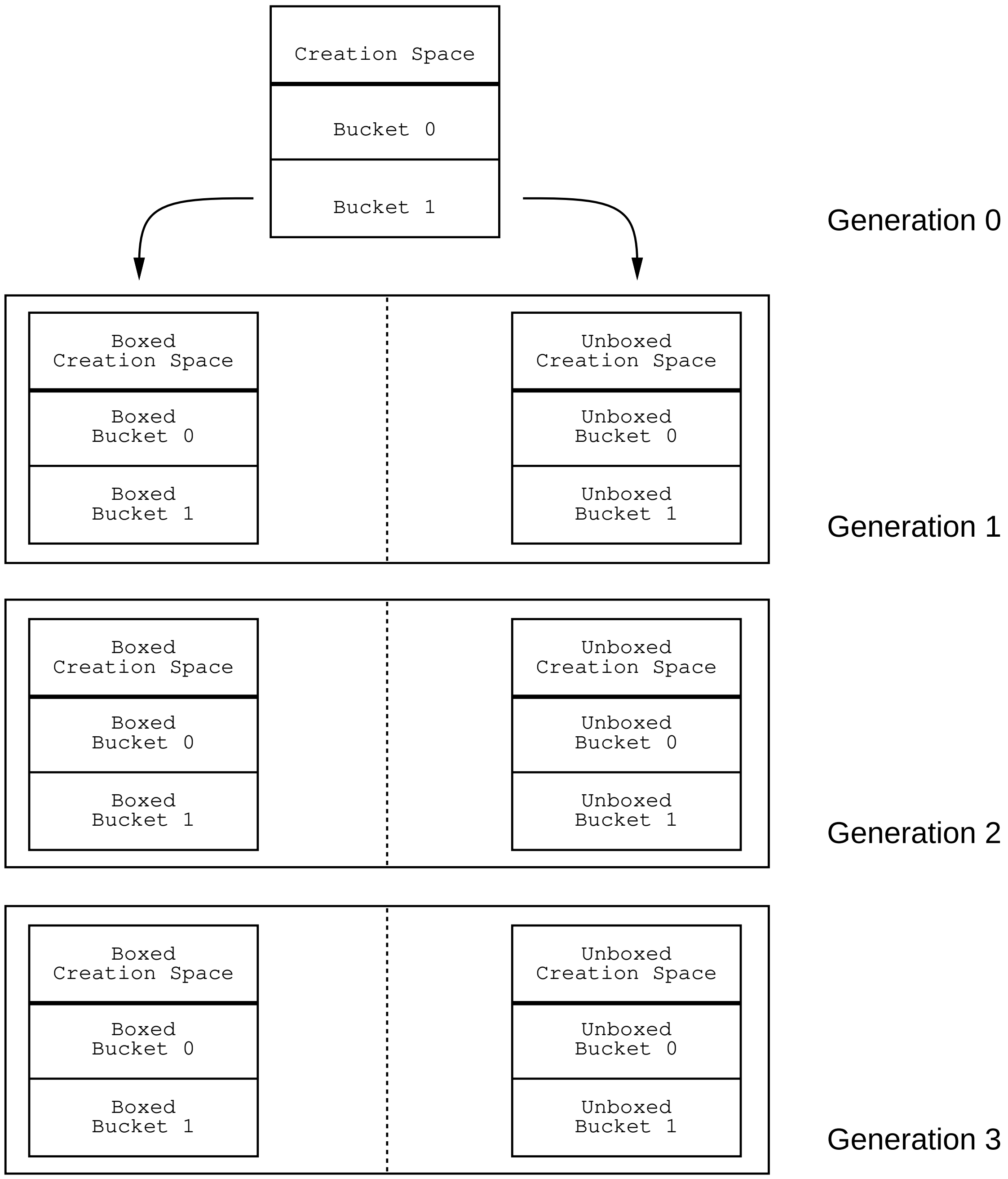


Figure 14: Layout of Generations with Separate Boxed and Unboxed Spaces

It is important to note that code objects are only created by the system functions `LOAD` and `COMPILE` and that they are immutable. Thus, it would be easy to special case the allocation of code objects if necessary. However, a simpler solution is to allocate code objects in the creation space in the zero generation where most other other objects are allocated. Allocation of code objects in the zero generation would give code objects an initial chance to die off before being advanced into the "permanent" code generation. Thus, only one additional generation, a "permanent" code generation, is needed for code objects.

Placing code objects in their own generation does not solve the problem with intergenerational references. Further, it is not possible to page mark the "permanent" code generation and scan it naively. A brute force solution would be to scan all of the advanced code objects at every scavenge in order to find the intergenerational references; this, however, would be much too costly. A viable solution would be to maintain an unscannable set and to use it to interpret code objects and marked pages during scanning as described described in Section 3.1. This solution should be explored further; it would be advantageous, however, to find a solution which does not require interpreting code objects at scavenge time.

It may be possible, through a special advancement policy for code objects, to keep the number of intergenerational references in a code object to a minimum by transporting any of the referents of a code object into an older generation when transporting the code object itself. However, it is impossible to cheaply guarantee that there will never be any intergenerational references in all advanced code objects. For example, some referent of a code object might have already been advanced (due to some other reference) when it comes time to advance the code object; this would make it impossible to advance the referent to an older generation without also scavenging the referent's generation.

Even though it is impossible to rely on a special advancement policy alone, such a policy is still useful because any remaining intergenerational references can simply be explicitly recorded and scanned at scavenge time. This would be beneficial since it would reduce the overhead of scanning code objects at scavenge time. To implement such a scheme, all intergenerational references would have to be copied into per-generation data structures along with pointers back to the original references. The garbage collector would scan the copied references at scavenge time looking for intergenerational references; when one was found, it would scavenge the reference and update the copy *and* the original reference by following the back pointer. The advantage of explicitly tracking the intergenerational references in code objects is that it eliminates the need for the scavenger to understand how to interpret code objects and marked pages. This reduction in complexity would offset the complexity of maintaining this separate data structure.

This scheme also has the potential to significantly reduce paging while scanning code objects for intergenerational references. If all of the intergenerational references due to code objects are recorded as pairs (the copy of the reference and a pointer back to the original reference) in contiguous addresses in memory, all of the intergenerational references will be contained on a small number of pages. Scanning these pages will cause only a small fraction of the page faults that scanning the code objects themselves would cause.

4.4.7. Allocation with Generational Garbage Collection

The novel use of an unsegregated zero generation greatly simplifies most allocation. Since there is only a single free pointer and since boxed objects and unboxed objects are treated identically, allocation in the zero generation may be performed as described in Section 4.2. Thus allocation in this design is also "fast."

It will, however, be advantageous to allocate certain objects directly into later generations. Large objects (and in particular large arrays) that would otherwise suffer significant copying costs should be allocated directly into an older generation. This is really a practical consideration but it makes sense from a generational viewpoint too – an object's lifetime is related to its size (although not as directly as it is to its age). Large objects tend to have longer lifetimes than smaller ones.

Further, it would be nice to export a primitive to the user to control where allocation occurs and to effectively facilitate the direct advancement of objects. This primitive would most likely be a macro or special form perhaps called `WITH-ALLOCATION-IN-GENERATION`. Any allocation occurring within the dynamic extent of the form would occur in the specified generation. This would allow the programmer to tell the system which data structures in the program are permanent so that the data would not incur repeated scavenging and copying costs.

Unfortunately, the use of an unsegregated zero generation makes allocation in later generations more difficult because allocation in later generations must be handled differently than allocation in the zero generation. This is not a problem for system code (such as the code which performs array allocation) as it can use special primitives to do its job; it does, however, make the implementation of the `WITH-ALLOCATION-IN-GENERATION` form difficult. The form cannot simply rebind the dynamic space allocation pointer (ALLOC) to the specified generation's allocation pointer[16] during its dynamic extent. Several solutions have been proposed but none have been both general and efficient. This is an area which requires more investigation.

4.4.8. Instrumentation and Tuning

A significant amount of work will have to be done in the area of instrumentation. Tuning a generational garbage collector can be very difficult due to the large number of variables involved and the complex manner in which they interact. Good instrumentation is one of the few tools which can help reduce this complexity. Effective and efficient instrumentation is also useful for implementing adaptive advancement policies and scavenging schedules.

4.4.9. Other Ideas

There are several other interesting ideas which may be incorporated into this design. The first involves the observation that the lifetime of some heap-allocated data are closely associated with

[16] It cannot because, if for no other reason, there is not a single allocation pointer for the older generations. The real reason, of course, is that boxed objects must go in one area and unboxed objects in another.

function calls and returns [25, 47]. Another related observation is that it is usually a good time to garbage collect when the control stack is near its top[17]. When the control stack is near its top, the program is typically running in one of its top-level functions and has usually just completed some major computation. Consequently, this is a good time to garbage collect as the heap is likely to contain a large percentage of garbage. While developing heuristics to utilize these observations and integrating them into the system may be difficult, they merit further attention.

Another idea which could be incorporated into this design is to add Appel-Ellis-Li-style incremental garbage collection for those programs which require pseudo-real-time performance. This could be integrated nicely into the Lisp external memory manager and, modulo code objects and the zero generation, could work without a crossing map due to the segregated heap. This style of incremental garbage collection is discussed in detail in Section 2.2.2.

[17] This assumes the reader pictures stacks growing down.

Acknowledgments

I would especially like to thank Robert A. MacLachlan who contributed significantly to the design of the generational garbage collector and with whom I have had many useful and interesting conversations. Paul Wilson, by sharing his experience, has also been very helpful. Additionally, I would like to extend my gratitude to Scott E. Fahlman for making all of this possible and for putting up with many slipped deadlines. Finally, I would like to thank Jefferey Shufelt who painfully proofread this paper and offered many helpful suggestions on the presentation; and, certainly, Martha McCrory who provided both encouragement and moral support.

References

[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. **The Design and Analysis of Computer Algorithms**. Addison-Wesley, Reading, Massachusetts, 1975.

[2] Aho, A. V., Sethi, R., and Ullman, J. D. **Compilers: Principles, Techniques, and Tools**. Addison-Wesley, Reading, Massachusetts, 1986.

[3] Appel, A. W. *Garbage Collection can be Faster than Stack Allocation*. no. CS-TR-045-86, Princeton University, Department of Computer Science, Princeton, New Jersey, June 1986.

[4] Appel, A. W. *Simple Generational Garbage Collection and Fast Allocation*. no. CS-TR-143-88, Princeton University, Department of Computer Science, Princeton, New Jersey, March 1988.

[5] Appel, A. W., Ellis, J. R., and Li, K. *Real-time Concurrent Collection on Stock Multiprocessors*. no. CS-TR-133-88, Princeton University, Department of Computer Science, Princeton, New Jersey, March 1988.

[6] Baker Jr., H. G. *List Processing in Real Time on a Serial Computer*. **Communications of the ACM**, vol. 21 (1978), pp. 280–294.

[7] Baron, R. V., Black, D., Bolosky, W., Chew, J., Draves, R. P., Golub, D. B., Rashid, R. F., Tevanian, Jr., A., , and Young, M. W. *Mach Kernel Interface Manual*. January 1990.

[8] Bartlett, J. F. *Mostly-Copying Garbage Collection Picks Up Generations and C++*. no. TN-12, DEC Western Research Laboratory, October 1989.

[9] Black, D. L., Golub, D. B., Rashid, R. F., Tevanian, Jr., A., and Young, M. W. *The Mach Exception Handling Facility*. no. CMU-CS-88-129, Carnegie Mellon University, Department of Computer Science, Pittsburgh, Pennsylvania, April 1988.

[10] Brooks, R. A. *Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware*. in: **Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments**. 1984, pp. 256–262.

[11] Cheney, C. J. *A Nonrecursive List Compacting Algorithm*. **Communications of the ACM**, vol. 13 (1970), pp. 677–678.

[12] Clark, D. W. *An Efficient List-Moving Algorithm Using Constant Workspace*. **Communications of the ACM**, vol. 19 (1976), pp. 352–354.

[13] Clark, D. W. and Green, C. *An Empirical Study of List Structures in LISP*. **Communications of the ACM**, vol. 20 (1977), pp. 78–87.

[14] CMU Common Lisp Group. *The New Object Layout for CMU Common Lisp*. 1990. *Will eventually be available as a CMU SCS Technical Report.*

[15] Collins, G. E. *A Method for Overlapping and Erasure of Lists*. **Communications of the ACM**, vol. 3 (1960), pp. 655–657.

[16] Cooper, E. C. and Draves, R. P. *C Threads*. no. CMU-CS-88-154, Carnegie Mellon University, Department of Computer Science, Pittsburgh, Pennsylvania, February 1988.

[17] Courts, R. *Improving Locality of Reference in a Garbage-Collecting Memory Management System*. **Communications of the ACM**, vol. 31 (1988), pp. 1128–1138.

[18] Deutsch, L. P. and Bobrow, D. G. *An Efficient, Incremental, Automatic Garbage Collector*. **Communications of the ACM**, vol. 19 (1976), pp. 522–526.

[19] Fenichel, R. R. and Yochelson, J. C. *A LISP Garbage-Collector for Virtual-Memory Computer Systems*. **Communications of the ACM**, vol. 12 (1969), pp. 611–612.

[20] Hansen, W. J. *Compact List Representation: Definition, Garbage Collection, and System Implementation*. **Communications of the ACM**, vol. 12 (1969), pp. 499–507.

[21] Hennessy, J. L. and Patterson, D. A. **Computer Architecture: A Quantitative Approach**. Morgan Kaufmann Publishers, San Mateo, California, 1989.

[22] Hickey, T. and Cohen, J. *Performance Analysis of On-the-Fly Garbage Collection*. **Communications of the ACM**, vol. 27 (1984), pp. 1143–1154.

[23] Kernighan, B. W. and Ritchie, D. M. **The C Programming Language**. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[24] Knuth, D. E. **Fundamental Algorithms**. Second. **The Art of Computer Programming**, vol. 1, Addison-Wesley, Reading, Massachusetts, 1968.

[25] Lieberman, H. and Hewitt, C. *A Real-Time Garbage Collector Based on the Lifetimes of Objects*. **Communications of the ACM**, vol. 26 (1983), pp. 419–429.

[26] Lindstrom, G. *Copying List Structures Using Bounded Workspace*. **Communications of the ACM**, vol. 17 (1974), pp. 198–202.

[27] MacLachlan, R. A. *personal communications*. May 1990.

[28] McCarthy, J. *Recursive Functions of Symbolic Expressions and Their Computation by Machine: Part I*. **Communications of the ACM**, vol. 3 (1960), pp. 184–195.

[29] McDonald, D. B., Fahlman, S. E., and Wholey, S. *Internal Design of CMU Common Lisp on the IBM RT PC*. no. CMU-CS-87-157, Carnegie Mellon University, Department of Computer Science, Pittsburgh, Pennsylvania, April 1988.

[30] Moon, D. A. *Garbage Collection in a Large Lisp System*. in: **ACM Symposium on LISP and Functional Programming**. 1984, pp. 235–246.

[31] Rees, J. A. *personal communications*. May 1990.

[32] Rees, J. A., Adams, N. I., and Meehan, J. R. *The T Manual*. Fourth, Yale University, January 1984.

[33] Shaw, R. A. *Empirical Analysis of a Lisp System*. Stanford University, 1988. *CSL-TR-88-351*.

[34] Shaw, R. A. *Improving Garbage Collector Performance in Virtual Memory*. no. CSL-TR-87-323, Stanford University, March 1987.

[35] Standish, T. A. **Data Structure Techniques**. Addison-Wesley, Reading, Massachusetts, 1980.

[36] Steele Jr, G. L. *Multiprocessing Compactifying Garbage Collection*. **Communications of the ACM**, vol. 18 (1975), pp. 495–508.

[37] Ungar, D. *Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm*. in: **Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments**. 1984, pp. 157–167.

[38] Ungar, D. and Jackson, F. *Tenuring Policies for Generation-Based Storage Reclamation*. in: **OOPSLA '88 Conference Proceedings**. 1988, pp. 1–17. *Also published in ACM SIGPLAN Notices.*

[39] Wadler, P. L. *Analysis of an Algorithm for Real Time Garbage Collection*. **Communications of the ACM**, vol. 19 (1976), pp. 491–500.

[40] Wegbreit, B. *A Generalised Compacting Garbage Collector*. **Computer Journal**, vol. 15 (1972), pp. 204–208.

[41] Wegbreit, B. *A Space-Efficient List Structure Tracing Algorithm*. **IEEE Transactions on Computers**, vol. C-21 (1972), pp. 1009–1010.

[42] Weizenbaum, J. *Symmetric List Processor*. **Communications of the ACM**, vol. 6 (1963), pp. 524–544.

[43] White, J. L. *Address/Memory Management For A Gigantic LISP Environment or, GC Considered Harmful*. in: **Conference Record of the 1980 Lisp Conference**. 1980, pp. 119–127.

[44] Wilson, P. R. *personal communications*. 1990.

[45] Wilson, P. R. *A Simple Bucket-Brigade Advancement Mechanism for Generation-Based Garbage Collection*. **SIGPLAN Notices**, vol. 24 (1989), pp. 38–46.

[46] Wilson, P. R. and Moher, T. G. *A "Card-Marking" Scheme for Controlling Intergenerational References in Generation-Based Garbage Collection on Stock Hardware*. **SIGPLAN Notices**, vol. 24 (1989), pp. 87–91.

[47] Wilson, P. R. and Moher, T. G. *Design of the Opportunistic Garbage Collector*. 1989, pp. 23–35.

A The Code

```c
/*
 * Header file for GC
 *
 * $Header: gc.h,v 1.3 90/05/13 15:30:58 ch Locked $
 */

#if !defined(_INCLUDED_GC_H_)
#define _INCLUDED_GC_H_

#include "lisp.h"

extern void gc_lose();

#define gc_assert(ex) { \
        if (!(ex)) { \
                fprintf(stderr, "GC invariant lost!  File \"%s\", line %d\n", \
                __FILE__, __LINE__); \
                gc_lose(); \
        } \
}

#define CEILING(x,y) (((x) + ((y) - 1)) & (~((y) - 1)))
#define FIXNUM_TO_INT(x) ((x)>>2)
#define INT_TO_FIXNUM(x) ((x)<<2)

#endif
```

```
/*
 * Stop and Copy GC based on Cheney's algorithm.
 *
 * $Header: gc.c,v 1.6 90/05/13 22:36:23 ch Locked $
 *
 * Written by Christopher Hoover.
 */

#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <signal.h>
#include "lisp.h"
#include "ldb.h"
#include "gc.h"
#include "globals.h"
#include "interrupt.h"
#include "validate.h"
#include "lispregs.h"

lispobj *from_space;
lispobj *from_space_free_pointer;

lispobj *new_space;
lispobj *new_space_free_pointer;

static int (*scavtab[256])();
static lispobj (*transother[256])();

static struct weak_pointer *weak_pointers;
```

```
/* Predicates */

#define from_space_p(ptr) \
        ((from_space <= ((lispobj *) ptr)) && \
         (((lispobj *) ptr) < from_space_free_pointer))

#define new_space_p(ptr) \
        ((new_space <= ((lispobj *) ptr)) && \
         (((lispobj *) ptr) < new_space_free_pointer))
```

```
/* GC Lossage */

void
gc_lose()
{
	exit(1);
}
```

```
/* Copying Objects */

static lispobj
copy_object(object, nwords)
lispobj object;
int nwords;
{
        int tag;
        lispobj *new;
        lispobj *source, *dest;

        gc_assert(Pointerp(object));
        gc_assert(from_space_p(object));
        gc_assert((nwords & 0x01) == 0);

        /* get tag of object */
        tag = LowtagOf(object);

        /* allocate space */
        new = new_space_free_pointer;
        new_space_free_pointer += nwords;

        dest = new;
        source = (lispobj *) PTR(object);

        /* copy the object */
        while (nwords > 0) {
                *dest++ = *source++;
                *dest++ = *source++;
                nwords -= 2;
        }

        /* return lisp pointer of new object */
        return ((lispobj) new) | tag;
}
```

```
/* Collect Garbage */

static double tv_diff(x, y)
struct timeval *x, *y;
{
    return (((double) x->tv_sec + (double) x->tv_usec * 1.0e-6) -
            ((double) y->tv_sec + (double) y->tv_usec * 1.0e-6));
}

collect_garbage()
{
        struct timeval start_tv, stop_tv;
        struct rusage start_rusage, stop_rusage;
        double real_time, system_time, user_time;
        lispobj *current_static_space_free_pointer;
        long static_space_size;
        long control_stack_size, binding_stack_size;
        long size_retained, size_discarded;
        int oldmask;

        getrusage(RUSAGE_SELF, &start_rusage);
        gettimeofday(&start_tv, (struct timezone *) 0);

        printf("[Collecting garbage ... \n");

        oldmask = sigblock(BLOCKABLE);

        current_static_space_free_pointer =
                (lispobj *) SymbolValue(STATIC_SPACE_FREE_POINTER);


        /* Set up from space and new space pointers */

        from_space = current_dynamic_space;
        from_space_free_pointer = current_dynamic_space_free_pointer;

        if (current_dynamic_space == dynamic_0_space)
                new_space = dynamic_1_space;
        else if (current_dynamic_space == dynamic_1_space)
                new_space = dynamic_0_space;
        else {
                fprintf(stderr, "GC lossage.  Current dynamic space is bogus!\n");
                gc_lose();
        }

        os_validate(new_space, DYNAMIC_SPACE_SIZE);
        new_space_free_pointer = new_space;


        /* Initialize the weak pointer list. */
        weak_pointers = (struct weak_pointer *) NULL;


        /* Scavenge the roots. */
        printf("Scavenging interrupt contexts ...\n");
        scavenge_interrupt_contexts();

        printf("Scavenging interrupt handlers (%d bytes) ...\n",
               sizeof(interrupt_handlers));
        scavenge((lispobj *) interrupt_handlers,
                 sizeof(interrupt_handlers) / sizeof(lispobj));

        control_stack_size = current_control_stack_pointer - control_stack;
        printf("Scavenging the control stack (%d bytes) ...\n",
               control_stack_size * sizeof(lispobj));
        scavenge(control_stack, control_stack_size);

        binding_stack_size = current_binding_stack_pointer - binding_stack;
        printf("Scavenging the binding stack (%d bytes) ...\n",
               binding_stack_size * sizeof(lispobj));
        scavenge(binding_stack, binding_stack_size);

        static_space_size = current_static_space_free_pointer - static_space;
        printf("Scavenging static space (%d bytes) ...\n",
               static_space_size * sizeof(lispobj));
        scavenge(static_space, static_space_size);

        printf("Scavenging new space (%d bytes) ...\n",
               (new_space_free_pointer - new_space) * sizeof(lispobj));
        scavenge_newspace();

        printf("Scanning weak pointers ...\n");
        scan_weak_pointers();

        /* Flip spaces */
        os_invalidate(current_dynamic_space, DYNAMIC_SPACE_SIZE);
        current_dynamic_space = new_space;
        current_dynamic_space_free_pointer = new_space_free_pointer;

        size_discarded = (from_space_free_pointer - from_space) * sizeof(lispobj);
        size_retained = (new_space_free_pointer - new_space) * sizeof(lispobj);

        /* Zero stack */
```

```
        printf("Zeroing empty part of control stack ...\n");
        os_zero(current_control_stack_pointer,
                CONTROL_STACK_SIZE - control_stack_size * sizeof(lispobj));

        (void) sigsetmask(oldmask);

        printf("done.]\n");

        printf("Total of %d bytes out of %d bytes retained (%3.2f%%).\n",
               size_retained, size_discarded,
               (((float) size_retained) / ((float) size_discarded)) * 100.0);

        gettimeofday(&stop_tv, (struct timezone *) 0);
        getrusage(RUSAGE_SELF, &stop_rusage);

        real_time = tv_diff(&stop_tv, &start_tv);
        user_time = tv_diff(&stop_rusage.ru_utime, &start_rusage.ru_utime);
        system_time = tv_diff(&stop_rusage.ru_stime, &start_rusage.ru_stime);

        printf("Statistics:\n");
        printf("%10.2f msec of real time\n", real_time * 1000.0);
        printf("%10.2f msec of user time,\n", user_time * 1000.0);
        printf("%10.2f msec of system time.\n", system_time * 1000.0);

        printf("%10.2f M bytes/sec collected.\n",
               (((float) size_retained / (float) (1<<20)) / real_time));
}
```

```
/* Scavenging */

static
scavenge(start, nwords)
lispobj *start;
long nwords;
{
	while (nwords > 0) {
		lispobj object;
		int type, words_scavenged;

		object = *start;
		type = TypeOf(object);

		words_scavenged = (scavtab[type])(start, object);

		start += words_scavenged;
		nwords -= words_scavenged;
	}
	gc_assert(nwords == 0);
}

static
scavenge_newspace()
{
	lispobj *here;

	here = new_space;
	while (here < new_space_free_pointer) {
		lispobj object;
		int type, words_scavenged;

		object = *here;
		type = TypeOf(object);

		words_scavenged = (scavtab[type])(here, object);

		here += words_scavenged;
	}
	gc_assert(here == new_space_free_pointer);
}
```

```
/* Scavenging Interrupt Contexts */

scavenge_interrupt_contexts()
{
        int i, index;
        struct sigcontext *context;

        index = FIXNUM_TO_INT(SymbolValue(FREE_INTERRUPT_CONTEXT_INDEX));

        for (i = 0; i < index; i++) {
                context = lisp_interrupt_contexts[i];
                scavenge_interrupt_context(context);
        }
}

static int boxed_registers[] = {
        A0, A1, A2, A3, A4, A5, CNAME, LEXENV,
        ARGS, OLDCONT, LRA, L0, L1, L2, CODE
};

scavenge_interrupt_context(context)
struct sigcontext *context;
{
        int i;
        unsigned long lip;
        unsigned long lip_offset;
        int lip_register_pair;
        unsigned long pc_code_offset;

        /* Find the LIP's register pair and calculate it's offset */
        /* before we scavenge the context. */
        lip = context->sc_regs[LIP];
        lip_offset = 0xFFFFFFFF;
        lip_register_pair = -1;
        for (i = 0; i < (sizeof(boxed_registers) / sizeof(int)); i++) {
                unsigned long reg, offset;
                int index;

                index = boxed_registers[i];
                reg = context->sc_regs[index];
                if (reg <= lip) {
                        offset = lip - reg;
                        if (offset < lip_offset) {
                                lip_offset = offset;
                                lip_register_pair = index;
                        }
                }
        }

        /* Compute the PC's offset from the start of the CODE */
        /* register. */
        pc_code_offset = context->sc_pc - context->sc_regs[CODE];

        /* Scanvenge all boxed registers in the context. */
        for (i = 0; i < (sizeof(boxed_registers) / sizeof(int)); i++) {
                int index;
                unsigned long reg;

                index = boxed_registers[i];
                reg = context->sc_regs[index];
                scavenge((lispobj *) &(context->sc_regs[index]), 1);
        }

        /* Fix the LIP */
        context->sc_regs[LIP] =
                context->sc_regs[lip_register_pair] + lip_offset;

        /* Fix the PC if it was in from space */
        if (from_space_p(context->sc_pc))
                context->sc_pc = context->sc_regs[CODE] + pc_code_offset;
}
```

```
/* Code and Code-Related Objects */

static lispobj trans_function_header();
static lispobj trans_closure_function_header();
static lispobj trans_boxed();

static
scav_function_pointer(where, object)
lispobj *where, object;
{
        gc_assert(Pointerp(object));

        if (from_space_p(object)) {
                lispobj first, *first_pointer;

                /* object is a pointer into from space.  check to see */
                /* if it has been forwarded */
                first_pointer = (lispobj *) PTR(object);
                first = *first_pointer;

                if (!(Pointerp(first) && new_space_p(first))) {
                        int type;
                        lispobj copy;

                        /* must transport object -- object may point */
                        /* to either a function header, a closure */
                        /* function header, or to a closure header. */

                        type = TypeOf(first);
                        switch (type) {
                        case type_FunctionHeader:
                                copy = trans_function_header(object);
                                break;
                        case type_ClosureFunctionHeader:
                                copy = trans_closure_function_header(object);
                                break;
                        case type_ClosureHeader:
                                copy = trans_boxed(object);
                                break;
                        default:
                                fprintf(stderr, "GC lossage.  Bogus function pointer.\n");
                                fprintf(stderr, "Pointer: 0x%08x, Header: 0x%08x\n",
                                        (unsigned long) object, (unsigned long) first);
                                gc_lose();
                        }

                        first = *first_pointer = copy;
                }

                gc_assert(Pointerp(first));
                gc_assert(!from_space_p(first));

                *where = first;
        }
        return 1;
}

static struct code *
trans_code(code)
struct code *code;
{
        struct code *new_code;
        lispobj first, l_code, l_new_code;
        int nheader_words, ncode_words, nwords;
        unsigned long displacement;
        lispobj fheaderl, *prev_pointer;

        /* if object has already been transported, just return pointer */
        first = code->header;
        if (Pointerp(first) && new_space_p(first))
                return (struct code *) PTR(first);

        gc_assert(TypeOf(first) == type_CodeHeader);

        /* prepare to transport the code vector */
        l_code = (lispobj) code | type_OtherPointer;

        ncode_words = FIXNUM_TO_INT(code->code_size);
        nheader_words = HeaderValue(code->header);
        nwords = ncode_words + nheader_words;
        nwords = CEILING(nwords, 2);

        l_new_code = copy_object(l_code, nwords);
        new_code = (struct code *) PTR(l_new_code);

        displacement = l_new_code - l_code;

        /* set forwarding pointer */
        code->header = l_new_code;

        /* set forwarding pointers for all the function headers in the */
        /* code object.  also fix all self pointers */
```

```
	fheaderl = code->entry_points;
	prev_pointer = &new_code->entry_points;

	while (fheaderl != NIL) {
		struct function_header *fheaderp, *nfheaderp;
		lispobj nfheaderl, header;

		fheaderp = (struct function_header *) PTR(fheaderl);
		header = fheaderp->header;
		gc_assert(TypeOf(header) == type_FunctionHeader);

		/* calcuate the new function pointer and the new */
		/* function header */
		nfheaderl = fheaderl + displacement;
		nfheaderp = (struct function_header *) PTR(nfheaderl);

		/* set forwarding pointer */
		fheaderp->header = nfheaderl;

		/* fix self pointer */
		nfheaderp->self = nfheaderl;

		*prev_pointer = nfheaderl;

		fheaderl = fheaderp->next;
		prev_pointer = &nfheaderp->next;
	}

	return new_code;
}

static
scav_code_header(where, object)
lispobj *where, object;
{
	struct code *code;
	int nheader_words, ncode_words, nwords;
	lispobj fheaderl;
	struct function_header *fheaderp;

	code = (struct code *) where;
	ncode_words = FIXNUM_TO_INT(code->code_size);
	nheader_words = HeaderValue(object);
	nwords = ncode_words + nheader_words;
	nwords = CEILING(nwords, 2);

	/* Scavenge the boxed section of the code data block */
	scavenge(where + 1, nheader_words - 1);

	/* Scavenge the boxed section of each function object in the */
	/* code data block */
	fheaderl = code->entry_points;
	while (fheaderl != NIL) {
		lispobj header;

		fheaderp = (struct function_header *) PTR(fheaderl);
		header = fheaderp->header;
		gc_assert(TypeOf(header) == type_FunctionHeader);

		scavenge(&fheaderp->name, 1);
		scavenge(&fheaderp->arglist, 1);
		scavenge(&fheaderp->type, 1);

		fheaderl = fheaderp->next;
	}

	return nwords;
}

static lispobj
trans_code_header(object)
lispobj object;
{
	struct code *ncode;

	ncode = trans_code((struct code *) PTR(object));
	return (lispobj) ncode | type_OtherPointer;
}


static
scav_return_pc_header(where, object)
lispobj *where, object;
{
	fprintf(stderr, "GC lossage.  Should not be scavenging a ");
	fprintf(stderr, "Return PC Header.\n");
	fprintf(stderr, "where = 0x%08x, object = 0x%08x",
		(unsigned long) where, (unsigned long) object);
	gc_lose();
}

static lispobj
trans_return_pc_header(object)
```

```
lispobj object;
{
        struct function_header *return_pc;
        unsigned long offset;
        struct code *code, *ncode;

        return_pc = (struct function_header *) PTR(object);
        offset = HeaderValue(return_pc->header) * 4;

        /* Transport the whole code object */
        code = (struct code *) ((unsigned long) return_pc - offset);
        ncode = trans_code(code);

        return ((lispobj) ncode + offset) | type_OtherPointer;
}


static
scav_function_header(where, object)
lispobj *where, object;
{
        fprintf(stderr, "GC lossage.  Should not be scavenging a ");
        fprintf(stderr, "Function Header.\n");
        fprintf(stderr, "where = 0x%08x, object = 0x%08x",
                (unsigned long) where, (unsigned long) object);
        gc_lose();
}

static lispobj
trans_function_header(object)
lispobj object;
{
        struct function_header *fheader;
        unsigned long offset;
        struct code *code, *ncode;

        fheader = (struct function_header *) PTR(object);
        offset = HeaderValue(fheader->header) * 4;

        /* Transport the whole code object */
        code = (struct code *) ((unsigned long) fheader - offset);
        ncode = trans_code(code);

        return ((lispobj) ncode + offset) | type_FunctionPointer;
}


static
scav_closure_function_header(where, object)
lispobj *where, object;
{
        fprintf(stderr, "GC lossage.  Should not be scavenging a ");
        fprintf(stderr, "Closure Function Header.\n");
        fprintf(stderr, "where = 0x%08x, object = 0x%08x",
                (unsigned long) where, (unsigned long) object);
        gc_lose();
}

static lispobj
trans_closure_function_header(object)
lispobj object;
{
        struct function_header *fheader;
        unsigned long offset;
        struct code *code, *ncode;

        fheader = (struct function_header *) PTR(object);
        offset = HeaderValue(fheader->header) * 4;

        /* Transport the whole code object */
        code = (struct code *) ((unsigned long) fheader - offset);
        ncode = trans_code(code);

        return ((lispobj) ncode + offset) | type_FunctionPointer;
}
```

```
/* Structures */

static
scav_structure_pointer(where, object)
lispobj *where, object;
{
        if (from_space_p(object)) {
                fprintf(stderr, "GC lossage.  Cannot scavenge structure pointer in from space!\n");
                gc_lose();
        } else
                return 1;
}
```

```
/* Lists and Conses */

static lispobj trans_list();

static
scav_list_pointer(where, object)
lispobj *where, object;
{
        gc_assert(Pointerp(object));

        if (from_space_p(object)) {
                lispobj first, *first_pointer;

                /* object is a pointer into from space.  check to see */
                /* if it has been forwarded */
                first_pointer = (lispobj *) PTR(object);
                first = *first_pointer;

                if (!(Pointerp(first) && new_space_p(first)))
                        first = *first_pointer = trans_list(object);

                gc_assert(Pointerp(first));
                gc_assert(!from_space_p(first));

                *where = first;
        }
        return 1;
}

static lispobj
trans_list(object)
lispobj object;
{
        lispobj new_list_pointer;
        struct cons *cons, *new_cons;

        cons = (struct cons *) PTR(object);

        new_list_pointer = copy_object(object, 2);
        new_cons = (struct cons *) PTR(new_list_pointer);

        /* Set forwarding pointer. */
        cons->car = new_list_pointer;

        /* Try to linearize the list in the cdr direction to help reduce */
        /* paging. */

        while (1) {
                lispobj cdr, new_cdr;
                struct cons *cdr_cons, *new_cdr_cons;

                cdr = cons->cdr;

                if (!((LowtagOf(cdr) == type_ListPointer) && from_space_p(cdr)))
                        break;

                cdr_cons = (struct cons *) PTR(cdr);

                new_cdr = copy_object(cdr, 2);
                new_cdr_cons = (struct cons *) PTR(new_cdr);

                /* Set forwarding pointer */
                cdr_cons->car = new_cdr;

                /* Update the cdr of the last cons copied into new */
                /* space to keep the newspace scavenge from having to */
                /* do it. */
                new_cons->cdr = new_cdr;

                cons = cdr_cons;
                new_cons = new_cdr_cons;
        }

        return new_list_pointer;
}
```

```
/* Scavenging and Transporting Other Pointers */

static
scav_other_pointer(where, object)
lispobj *where, object;
{
	gc_assert(Pointerp(object));

	if (from_space_p(object)) {
		lispobj first, *first_pointer;

		/* object is a pointer into from space.  check to see */
		/* if it has been forwarded */
		first_pointer = (lispobj *) PTR(object);
		first = *first_pointer;

		if (!(Pointerp(first) && new_space_p(first)))
			first = *first_pointer =
				(transother[TypeOf(first)])(object);

		gc_assert(Pointerp(first));
		gc_assert(!from_space_p(first));

		*where = first;
	}
	return 1;
}
```

```
/* Immediate, Boxed, and Unboxed Objects */

static
scav_immediate(where, object)
lispobj *where, object;
{
	return 1;
}

static lispobj
trans_immediate(object)
lispobj object;
{
	fprintf(stderr, "GC lossage.  Trying to transport an immediate!?\n");
	gc_lose();
}


static
scav_boxed(where, object)
lispobj *where, object;
{
	return 1;
}

static lispobj
trans_boxed(object)
lispobj object;
{
	lispobj header;
	unsigned long length;

	gc_assert(Pointerp(object));

	header = *((lispobj *) PTR(object));
	length = HeaderValue(header) + 1;
	length = CEILING(length, 2);

	return copy_object(object, length);
}


static
scav_unboxed(where, object)
lispobj *where, object;
{
	unsigned long length;

	length = HeaderValue(object) + 1;
	length = CEILING(length, 2);

	return length;
}

static lispobj
trans_unboxed(object)
lispobj object;
{
	lispobj header;
	unsigned long length;


	gc_assert(Pointerp(object));

	header = *((lispobj *) PTR(object));
	length = HeaderValue(header) + 1;
	length = CEILING(length, 2);

	return copy_object(object, length);
}
```

```
/* Vector-Like Objects */

#define NWORDS(x,y) (CEILING((x),(y)) / (y))

static
scav_string(where, object)
lispobj *where, object;
{
        struct vector *vector;
        int length, nwords;

        /* NOTE: Strings contain one more byte of data than the length */
        /* slot indicates. */

        vector = (struct vector *) where;
        length = FIXNUM_TO_INT(vector->length) + 1;
        nwords = CEILING(NWORDS(length, 4) + 2, 2);

        return nwords;
}

static lispobj
trans_string(object)
{
        struct vector *vector;
        int length, nwords;

        gc_assert(Pointerp(object));

        /* NOTE: Strings contain one more byte of data than the length */
        /* slot indicates. */

        vector = (struct vector *) PTR(object);
        length = FIXNUM_TO_INT(vector->length) + 1;
        nwords = CEILING(NWORDS(length, 4) + 2, 2);

        return copy_object(object, nwords);
}


static lispobj
trans_vector(object)
lispobj object;
{
        struct vector *vector;
        int length, nwords;
        int subtype;

        gc_assert(Pointerp(object));

        vector = (struct vector *) PTR(object);

        length = FIXNUM_TO_INT(vector->length);
        nwords = CEILING(length + 2, 2);

        /* When transporting an EQ hashtable, GC must change subtype */
        /* so that the hash functions will know to rehash it.  */

        subtype = HeaderValue(vector->header);
        if (subtype == subtype_VectorValidHashing)
                vector->header = (subtype_VectorMustRehash<<8) |
                        type_SimpleVector;

        return copy_object(object, nwords);
}


static
scav_vector_bit(where, object)
lispobj *where, object;
{
        struct vector *vector;
        int length, nwords;

        vector = (struct vector *) where;
        length = FIXNUM_TO_INT(vector->length);
        nwords = CEILING(NWORDS(length, 32) + 2, 2);

        return nwords;
}

static lispobj
trans_vector_bit(object)
lispobj object;
{
        struct vector *vector;
        int length, nwords;

        gc_assert(Pointerp(object));

        vector = (struct vector *) PTR(object);
        length = FIXNUM_TO_INT(vector->length);
        nwords = CEILING(NWORDS(length, 32) + 2, 2);
```

```
        return copy_object(object, nwords);
}


static
scav_vector_unsigned_byte_2(where, object)
lispobj *where, object;
{
        struct vector *vector;
        int length, nwords;

        vector = (struct vector *) where;
        length = FIXNUM_TO_INT(vector->length);
        nwords = CEILING(NWORDS(length, 16) + 2, 2);

        return nwords;
}

static lispobj
trans_vector_unsigned_byte_2(object)
lispobj object;
{
        struct vector *vector;
        int length, nwords;

        gc_assert(Pointerp(object));

        vector = (struct vector *) PTR(object);
        length = FIXNUM_TO_INT(vector->length);
        nwords = CEILING(NWORDS(length, 16) + 2, 2);

        return copy_object(object, nwords);
}


static
scav_vector_unsigned_byte_4(where, object)
lispobj *where, object;
{
        struct vector *vector;
        int length, nwords;

        vector = (struct vector *) where;
        length = FIXNUM_TO_INT(vector->length);
        nwords = CEILING(NWORDS(length, 8) + 2, 2);

        return nwords;
}

static lispobj
trans_vector_unsigned_byte_4(object)
lispobj object;
{
        struct vector *vector;
        int length, nwords;

        gc_assert(Pointerp(object));

        vector = (struct vector *) PTR(object);
        length = FIXNUM_TO_INT(vector->length);
        nwords = CEILING(NWORDS(length, 8) + 2, 2);

        return copy_object(object, nwords);
}


static
scav_vector_unsigned_byte_8(where, object)
lispobj *where, object;
{
        struct vector *vector;
        int length, nwords;

        vector = (struct vector *) where;
        length = FIXNUM_TO_INT(vector->length);
        nwords = CEILING(NWORDS(length, 4) + 2, 2);

        return nwords;
}

static lispobj
trans_vector_unsigned_byte_8(object)
lispobj object;
{
        struct vector *vector;
        int length, nwords;

        gc_assert(Pointerp(object));

        vector = (struct vector *) PTR(object);
        length = FIXNUM_TO_INT(vector->length);
        nwords = CEILING(NWORDS(length, 4) + 2, 2);
```

```
        return copy_object(object, nwords);
}


static
scav_vector_unsigned_byte_16(where, object)
lispobj *where, object;
{
        struct vector *vector;
        int length, nwords;

        vector = (struct vector *) where;
        length = FIXNUM_TO_INT(vector->length);
        nwords = CEILING(NWORDS(length, 2) + 2, 2);

        return nwords;
}

static lispobj
trans_vector_unsigned_byte_16(object)
lispobj object;
{
        struct vector *vector;
        int length, nwords;

        gc_assert(Pointerp(object));

        vector = (struct vector *) PTR(object);
        length = FIXNUM_TO_INT(vector->length);
        nwords = CEILING(NWORDS(length, 2) + 2, 2);

        return copy_object(object, nwords);
}


static
scav_vector_unsigned_byte_32(where, object)
lispobj *where, object;
{
        struct vector *vector;
        int length, nwords;

        vector = (struct vector *) where;
        length = FIXNUM_TO_INT(vector->length);
        nwords = CEILING(length + 2, 2);

        return nwords;
}

static lispobj
trans_vector_unsigned_byte_32(object)
lispobj object;
{
        struct vector *vector;
        int length, nwords;

        gc_assert(Pointerp(object));

        vector = (struct vector *) PTR(object);
        length = FIXNUM_TO_INT(vector->length);
        nwords = CEILING(length + 2, 2);

        return copy_object(object, nwords);
}


static
scav_vector_single_float(where, object)
lispobj *where, object;
{
        struct vector *vector;
        int length, nwords;

        vector = (struct vector *) where;
        length = FIXNUM_TO_INT(vector->length);
        nwords = CEILING(length + 2, 2);

        return nwords;
}

static lispobj
trans_vector_single_float(object)
lispobj object;
{
        struct vector *vector;
        int length, nwords;

        gc_assert(Pointerp(object));

        vector = (struct vector *) PTR(object);
        length = FIXNUM_TO_INT(vector->length);
        nwords = CEILING(length + 2, 2);
```

```
        return copy_object(object, nwords);
}


static
scav_vector_double_float(where, object)
lispobj *where, object;
{
        struct vector *vector;
        int length, nwords;

        vector = (struct vector *) where;
        length = FIXNUM_TO_INT(vector->length);
        nwords = CEILING(length * 2 + 2, 2);

        return nwords;
}

static lispobj
trans_vector_double_float(object)
lispobj object;
{
        struct vector *vector;
        int length, nwords;

        gc_assert(Pointerp(object));

        vector = (struct vector *) PTR(object);
        length = FIXNUM_TO_INT(vector->length);
        nwords = CEILING(length * 2 + 2, 2);

        return copy_object(object, nwords);
}
```

```c
/* Weak Pointers */

#define WEAK_POINTER_NWORDS \
        CEILING((sizeof(struct weak_pointer) / sizeof(lispobj)), 2)

static
scav_weak_pointer(where, object)
lispobj *where, object;
{
        /* Do not let GC scavenge the value slot of the weak pointer */
        /* (that is why it is a weak pointer).  Note:  we could use */
        /* the scav_unboxed method here. */

        return WEAK_POINTER_NWORDS;
}

static lispobj
trans_weak_pointer(object)
lispobj object;
{
        int nwords;
        lispobj copy;
        struct weak_pointer *wp;

        gc_assert(Pointerp(object));

        /* Need to remember where all the weak pointers are that have */
        /* been transported so they can be fixed up in a post-GC pass. */

        copy = copy_object(object, WEAK_POINTER_NWORDS);
        wp = (struct weak_pointer *) PTR(copy);


        /* Push the weak pointer onto the list of weak pointers. */
        wp->next = weak_pointers;
        weak_pointers = wp;

        return copy;
}


scan_weak_pointers()
{
        struct weak_pointer *wp;

        for (wp = weak_pointers; wp != (struct weak_pointer *) NULL;
             wp = wp->next) {
                lispobj value;
                lispobj first, *first_pointer;

                value = wp->value;

                /* If not a pointer into fromspace, continue. */
                if (!(Pointerp(value) && from_space_p(value)))
                        continue;

                /* Now, we need to check if the object has been */
                /* forwarded.  If it has been, the weak pointer is */
                /* still good and needs to be updated.  Otherwise, the */
                /* weak pointer needs to be nil'ed out. */
                first_pointer = (lispobj *) PTR(value);
                first = *first_pointer;

                if (Pointerp(first) && new_space_p(first))
                        wp->value = first;
                else
                        wp->value = NIL;
        }
}
```

```
/* Initialization */

static
scav_lose(object)
lispobj object;
{
        fprintf(stderr, "GC lossage.  No scavenge function for object 0x%08x\n",
                (unsigned long) object);
        gc_lose();
}

static lispobj
trans_lose(object)
lispobj object;
{
        fprintf(stderr, "GC lossage.  No transport function for object 0x%08x\n",
                (unsigned long) object);
        gc_lose();
}

gc_init()
{
        int i;

        /* Scavenge Table */
        for (i = 0; i < 256; i++)
                scavtab[i] = scav_lose;

        for (i = 0; i < 32; i++) {
                scavtab[type_EvenFixnum|(i<<3)] = scav_immediate;
                scavtab[type_FunctionPointer|(i<<3)] = scav_function_pointer;
                /* OtherImmediate0 */
                scavtab[type_ListPointer|(i<<3)] = scav_list_pointer;
                scavtab[type_OddFixnum|(i<<3)] = scav_immediate;
                scavtab[type_StructurePointer|(i<<3)] = scav_structure_pointer;
                /* OtherImmediate1 */
                scavtab[type_OtherPointer|(i<<3)] = scav_other_pointer;
        }

        scavtab[type_Bignum] = scav_unboxed;
        scavtab[type_Ratio] = scav_boxed;
        scavtab[type_SingleFloat] = scav_unboxed;
        scavtab[type_DoubleFloat] = scav_unboxed;
        scavtab[type_Complex] = scav_boxed;
        scavtab[type_SimpleArray] = scav_boxed;
        scavtab[type_SimpleString] = scav_string;
        scavtab[type_SimpleBitVector] = scav_vector_bit;
        scavtab[type_SimpleVector] = scav_boxed;
        scavtab[type_SimpleArrayUnsignedByte2] = scav_vector_unsigned_byte_2;
        scavtab[type_SimpleArrayUnsignedByte4] = scav_vector_unsigned_byte_4;
        scavtab[type_SimpleArrayUnsignedByte8] = scav_vector_unsigned_byte_8;
        scavtab[type_SimpleArrayUnsignedByte16] = scav_vector_unsigned_byte_16;
        scavtab[type_SimpleArrayUnsignedByte32] = scav_vector_unsigned_byte_32;
        scavtab[type_SimpleArraySingleFloat] = scav_vector_single_float;
        scavtab[type_SimpleArrayDoubleFloat] = scav_vector_double_float;
        scavtab[type_ComplexString] = scav_boxed;
        scavtab[type_ComplexBitVector] = scav_boxed;
        scavtab[type_ComplexVector] = scav_boxed;
        scavtab[type_ComplexArray] = scav_boxed;
        scavtab[type_CodeHeader] = scav_code_header;
        scavtab[type_FunctionHeader] = scav_function_header;
        scavtab[type_ClosureFunctionHeader] = scav_closure_function_header;
        scavtab[type_ReturnPcHeader] = scav_return_pc_header;
        scavtab[type_ClosureHeader] = scav_boxed;
        scavtab[type_ValueCellHeader] = scav_boxed;
        scavtab[type_SymbolHeader] = scav_boxed;
        scavtab[type_BaseCharacter] = scav_immediate;
        scavtab[type_Sap] = scav_unboxed;
        scavtab[type_UnboundMarker] = scav_immediate;
        scavtab[type_WeakPointer] = scav_weak_pointer;


        /* Transport Other Table */
        for (i = 0; i < 256; i++)
                transother[i] = trans_lose;

        transother[type_Bignum] = trans_unboxed;
        transother[type_Ratio] = trans_boxed;
        transother[type_SingleFloat] = trans_unboxed;
        transother[type_DoubleFloat] = trans_unboxed;
        transother[type_Complex] = trans_boxed;
        transother[type_SimpleArray] = trans_boxed;
        transother[type_SimpleString] = trans_string;
        transother[type_SimpleBitVector] = trans_vector_bit;
        transother[type_SimpleVector] = trans_vector;
        transother[type_SimpleArrayUnsignedByte2] = trans_vector_unsigned_byte_2;
        transother[type_SimpleArrayUnsignedByte4] = trans_vector_unsigned_byte_4;
        transother[type_SimpleArrayUnsignedByte8] = trans_vector_unsigned_byte_8;
        transother[type_SimpleArrayUnsignedByte16] = trans_vector_unsigned_byte_16;
        transother[type_SimpleArrayUnsignedByte32] = trans_vector_unsigned_byte_32;
        transother[type_SimpleArraySingleFloat] = trans_vector_single_float;
        transother[type_SimpleArrayDoubleFloat] = trans_vector_double_float;
        transother[type_ComplexString] = trans_boxed;
```

```
        transother[type_ComplexBitVector] = trans_boxed;
        transother[type_ComplexVector] = trans_boxed;
        transother[type_ComplexArray] = trans_boxed;
        transother[type_CodeHeader] = trans_code_header;
        transother[type_FunctionHeader] = trans_function_header;
        transother[type_ClosureFunctionHeader] = trans_closure_function_header;
        transother[type_ReturnPcHeader] = trans_return_pc_header;
        transother[type_ClosureHeader] = trans_boxed;
        transother[type_ValueCellHeader] = trans_boxed;
        transother[type_SymbolHeader] = trans_boxed;
        transother[type_BaseCharacter] = trans_immediate;
        transother[type_Sap] = trans_unboxed;
        transother[type_UnboundMarker] = trans_immediate;
        transother[type_WeakPointer] = trans_weak_pointer;
}
```