Compacting Garbage Collection with Ambiguous Roots[1]

Joel F. Bartlett
Digital Western Research Laboratory
100 Hamilton Ave.
Palo Alto, Ca. 94301
bartlett@decwrl.dec.com

Copyright 1988, Digital Equipment Corporation

Introduction

Many modern garbage collectors [4] recover space by copying. Using an initial "root" set of pointers which are stored in known locations, all accessible objects are copied into a "new space". Two of the attractive properties of such a collector are that it results in memory compaction and it can have a running time proportional to the amount of accessible storage [2]. However, such schemes place a large burden on the underlying system as all pointers to the objects must be found and changed.

In most Lisp implementations, root finding is not a problem. In a specialized Lisp machine such as the MIT CADR and its descendants, everything is tagged and complex instructions perform references directly from tagged objects. In implementations on stock hardware such as VAX LISP, great efforts are made to control instruction sequences, stack layout, and register use. There, some registers may contain untagged pointers, or pointers into the middle of objects so some protocol must be provided to keep them updated.

Other environments present more serious problems in finding roots. If a Lisp system uses an intermediate language [7] as its target language, then it may have very little control over the actual code generated. While this approach may simplify the compiler and result in fast code because of the extensive machine dependent optimization provided by the intermediate language processor, it will not assure that Lisp pointers are treated in a uniform manner. Even a Lisp system which normally has complete control over its environment may find that it has problems supporting call-out to, and call-back from, foreign functions.

One collection method that has been used in the past in environments without reliable roots is mark-and-sweep. Each object which might be a root is treated in a conservative manner [6] [8]. That is, objects that might be valid pointers are treated as pointers for purposes of storage retention. As this type of collector will never relocate any objects, the only cost of guessing wrong is retaining extra data. While such a collector will work, it is not entirely satisfactory as it will not compact the heap, and its execution time is proportional to the total heap size.

To solve these problems, a collector has been devised which copies most objects in the heap without knowing exactly where the roots are. Instead of requiring that the root set be a known set of cells containing pointers which define all accessible storage, the new algorithm only requires that the root set include pointers which define all accessible storage. The cells in this new root set need not all be pointers, nor is it required that there not be cells which "look like pointers."

Using this root set, the algorithm divides all accessible objects in the heap into two classes: those which might have a direct reference from the root cells, and those which do not. The former items are left in place, and the latter items are copied into a compact area of memory. In practice, only a very small amount of the heap is left in place, so memory fragmentation is not a problem.

[1] This article is based on WRL Research Report 88/2 (available from the author).

An Overview of Stop-and-Copy Garbage Collection

The "mostly-copying" collector is best understood by showing how it is an evolution of the classical "stop-and-copy collector" [2]. The stop-and-copy algorithm manages a heap which is implemented using a contiguous block of storage. The algorithm divides the storage into two equal semispaces: "old space" and "new space". Storage is allocated by advancing a free space pointer over one of the semispaces (new space). Figure 1 shows this division of the heap and a sample list structure.

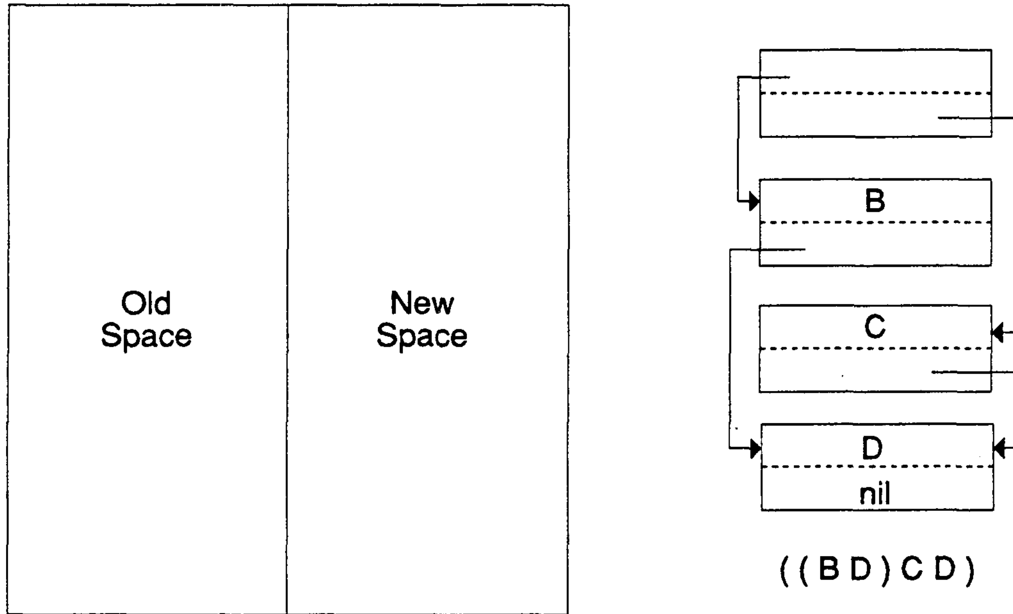

Figure 1: Stop-and-Copy memory organization and sample list

When all cells in a semispace have been allocated, the garbage collector is invoked. The first thing that it does is "flip" the semispaces, i.e. exchange old space and new space. Following this, the collector copys all accessible objects into the new space. This is done by processing a root set of known cells to find all immediately accessible items. As each root cell's contents will be changed during this process, the program must assure that each root cell always contains a valid pointer.

Pointers are updated (which causes objects to be copied) as follows. If the pointer references an object already in the new space, then it is correct. Otherwise, the object is examined to see if it contains a forwarding pointer to a copy of the object in new space. If so, then the pointer is updated to reference the new space copy. Failing these tests, an object in new space is allocated with the contents of the old object. A forwarding pointer is left in the old object, and the pointer is updated to reference the copy in new space.

Given that somewhere in the root set there exists a pointer to the head of the sample data structure in Figure 1, then applying these operations will result in copying the head of the list as in Figure 2 (In this figure and those which follow, the existing data structure is on the left, and newly allocated storage is on the right).

Once the initial items have been copied, the pointers that they contain are updated (see Figure 3). Since storage is sequentially allocated in new space, this can be done by sweeping across the new space and updating each pointer of each cell. Once all pointers in the new space have been updated, the garbage collection is complete.

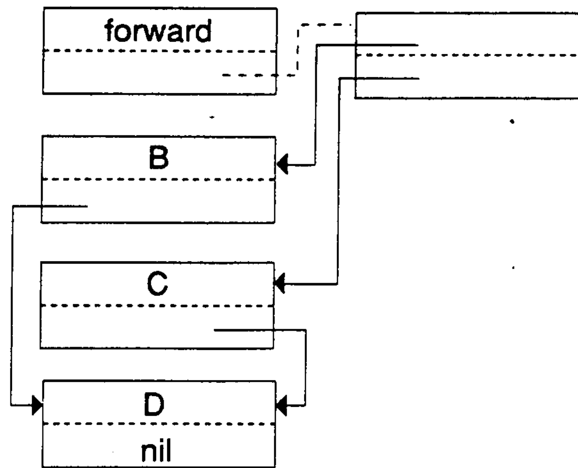

Figure 2: Copy the objects referenced by the roots

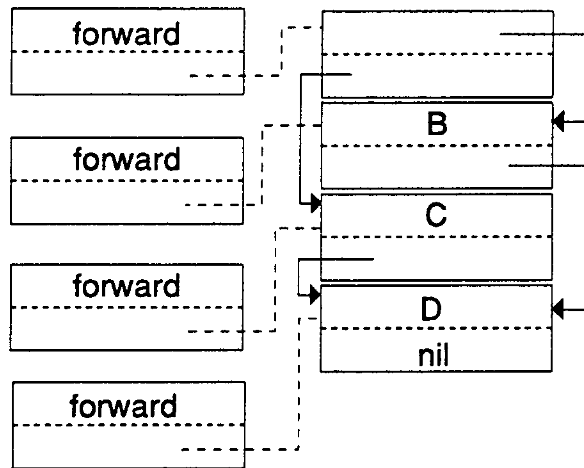

Figure 3: Copy the rest

Mostly-Copying Collector I

With this overview of the classical stop-and-copy collector in mind, our attention can now turn to the initial version of the mostly-copying collector. Its primary differences with the classical algorithm are in the root set and the heap organization.

The new algorithm makes few restrictions on the root set. It simply requires that somewhere in the set of cells, *R*, there be sufficient "hints" to find all accessible storage. A typical *R* would be the current program state, i.e. the entire contents of the processor's stack and registers.

The heap used by the new algorithm is a contiguous region of storage, divided into a number of equal-size pages, where page size is **independent** of the underlying hardware's page size. Associated with each page is a space identifier, *space*, which identifies the "space" that objects on the page belong to. In the figures illustrating this algorithm, the space identifier associated with the page containing the cell is the number to the left of the cell, and each cell is assumed to be in its own page (see Figure 4).

page space	page space
page space	page space
page space	page space
page space	page space

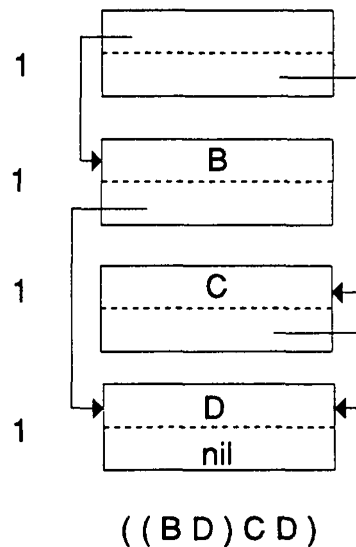

Figure 4: Mostly-copying memory organization and sample list

Two space identifiers: *current_space* and *next_space*, identify important sets of pages. During normal allocation, *current_space* and *next_space* are equal, but during garbage collection, *next_space* is set to the "next" space identifier. When comparing this algorithm with the classical one, it is reasonable to think of pages with their *space* identifier equal to *next_space* as the analogue of "new space", and those with their *space* identifier equal to *current_space* as the analogue of "old space". Like the classical one, this collector works by moving objects from "old space" to "new space". While this can be done by copying objects to newly allocated pages in *next_space*, it can also be done by changing the *space* identifier associated with the page holding the object to *next_space*. This later method is the key to mostly-copying collection as it leaves the object's address unchanged.

Memory allocation is a two part process; first, allocate a page of memory, and then allocate space from it. A page is free when its *space* field is not equal to *current_space* or *next_space*. When it is allocated, its *space* identifier is set to *next_space*.

The garbage collector is invoked when half the pages in the heap have been allocated. It starts by advancing *next_space* to the next space identifier. Next, it makes an educated guess as to what portions of the heap contain accessible items. This is done by examining each word in the stack and the registers and looking for "hints". If the word could be a pointer into a page of the heap allocated to the *current_space*, then that page is promoted to *next_space* by changing the page's *space* identifier (see Figure 5).

At the completion of this phase, all pages containing items which might be referenced by pointers in the stack or registers are in *next_space*. The items which they reference are now copied to *next_space* (see Figure 6). This is done by sweeping across all pages in the *next_space* and updating their pointers by a method similar to that used by the stop-and-copy collector. As before, the objects in the heap must be self-identifying at collection time, and their pointer fields must be valid. Once all pointers in pages in *next_space* have been updated, *current_space* is set to *next_space* and garbage collection is complete.

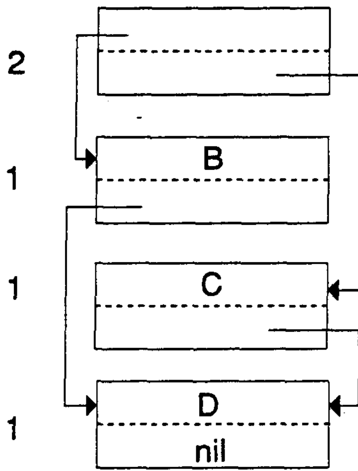

Figure 5: Promote possibly referenced pages to *next_space*

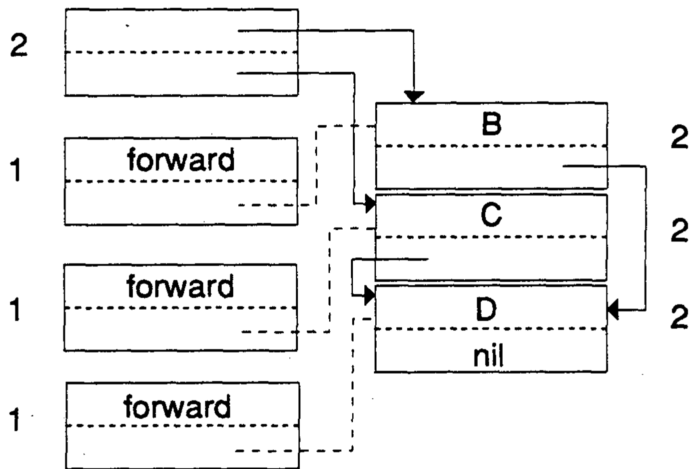

Figure 6: Copy the rest of the accessible objects

Mostly-Copying Collector II

The collection algorithm just described requires that all pages which might be promoted to the next generation be found at the start of collection. This may not possible in a Scheme [5] implementation where continuations (procedures which return the program to some previous state of the computation) are implemented by saving a copy of the program state [1]. Such continuations need not be visible at the start of garbage collection, but they may contain "hints" to objects which cannot be copied. In order to allow such objects, a second algorithm is introduced.

Using the register and stack contents for hints as in Algorithm I, pages containing the initial objects are marked for retention. Their contents are copied into the next space using an operation that is identical to that used in the stop-and-copy collector, except that no pointers are ever changed. As before, all accessible items are copied into the *next_space* by sweeping across these retained pages and all copied objects (see Figure 7). When continuations are found during this sweep, their saved state will be treated as hints which may identify more pages to retain and sweep.

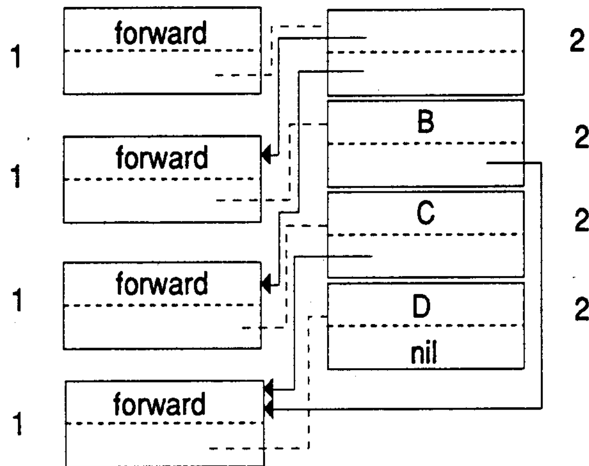

Figure 7: Copy all accessible items

When the sweep completes, all accessible objects will have copys in the *next_space*, but all pointers will point to the old copy of the object which in turn contains a forwarding pointer to the new copy. In addition, all pages which might be referenced by pointers in the stack, registers, or saved state in continuations are known (highlighted in Figure 8). Using this information, the pointers in the new copies of objects can now be corrected: if the pointer points into the heap to an old page and the page is not being retained, then the correct pointer is the forwarding pointer found in the object (see Figure 9).

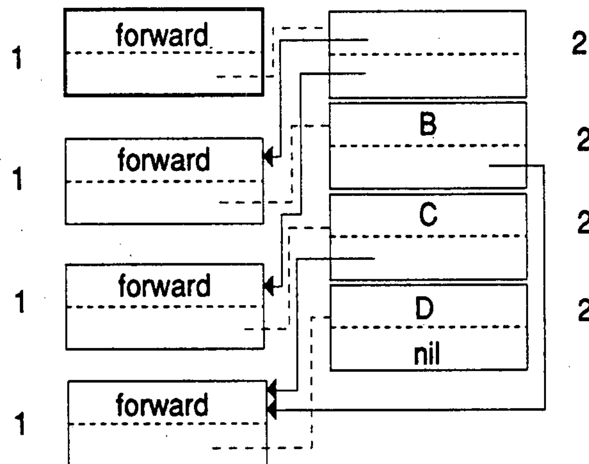

Figure 8: All promoted pages now known

Following the correction phase, the retained pages have their contents restored by copying back each object on a retained page using the forwarding pointer left in the object. The page's space identifier is updated at this time (see Figure 10).

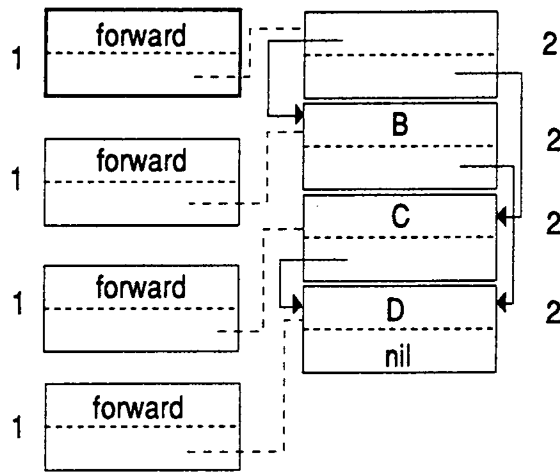

Figure 9: Correct pointers

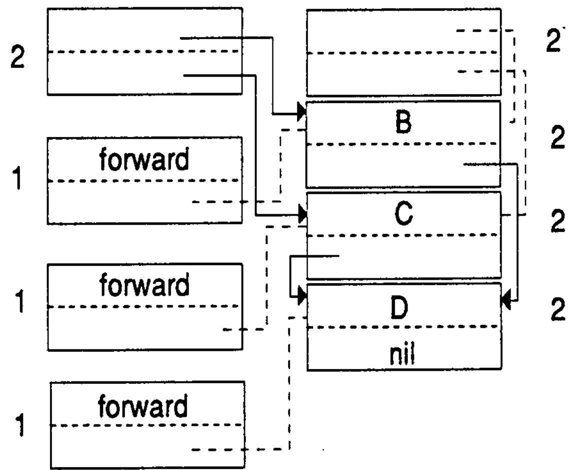

Figure 10: Copy back contents of promoted pages

Putting Mostly-Copying Collection to Work

The "mostly-copying" collection algorithm was developed to provide garbage collection for a Scheme implementation for the Titan, a high performance experimental workstation developed at WRL. The "official" definition for the machine is not the processor's instruction set, but the Mahler intermediate language [7] which is the object code produced by all compilers. When the Mahler code is compiled, extensive machine dependent optimization is done. Since details such as the number of registers and the mechanisms for local storage allocation and procedure call and return are hidden, conventional root finding methods are not applicable.

In order to be used in the Scheme system, the algorithm needed several extensions. First, it needed to handle varying length structures such as continuations, vectors of pointers, and strings of characters. Next, with multiple kinds of objects in the heap, a mechanism for identifying the type of object was provided to allow a page to be swept. Finally, objects larger than a page were allowed.

These additions were made by associating some *TYPE* bits with each page. For Titan Scheme, these bits have the

following values: *PAIR*, which indicates that the page contains cons cells, *EXTENDED*, indicating that the page contains objects whose type and length is encoded in a header word, and *CONTINUED*, which indicates that the page is part of the object defined on a previous page.

Storage allocation is more complex as there are now two "current free pages", one for cons cells and one for extended objects. Extended objects larger than a page are allocated as many pages as required, with some portion of the last page potentially unused. Extended objects smaller than a page are allocated on the current free extended page if they fit, or at the start of a new page if they do not. If an object does not fit on the remainder of the page, then the remainder is discarded.

Comparison with the Classical Algorithm

The new algorithm is similar to the classical algorithm in its resource demands [2]. The additional flags required for each page can be stored in two 32-bit integers. Given a page size of 512 bytes, this requires less than 2% additional storage.

Like the classical algorithm, it is able to operate using a constant amount of stack as its processing is iterative. This is highly desirable as one wishes to be able to garbage collect a heap containing arbitrary structures.

Finally, the new algorithm's running time remains $O(n)$, where n is the amount of retained storage. Algorithm I is very similar in running time to the classical algorithm, whereas algorithm II is probably twice as expensive due to the pointer correction scan. However, even it compares quite favorably with the running time of a mark-and-sweep collector which is $O(m)$, where m is the total size of the heap.

Advantages of the New Algorithm

The major advantage of this new algorithm is that it places far fewer restrictions on the initial roots. While both algorithms require that a set of initial roots cells, R, be designated, the classical algorithm requires that each member of R be a valid pointer. If this is not true, then programs using this algorithm will not operate correctly. The new algorithm requires that within R, there must be pointers to all accessible objects, however it makes no requirements on the individual members of R. Any given cell in R may contain any value, including values that "look like" valid pointers. At worst, this will result in the retention of unneeded storage.

This less restrictive method for root finding also solves problems with register values computed from tagged pointers. As the new algorithm does not differentiate between pointers which point directly to an object and those which point into the middle of an object, cells which might contain a pointer are simply made part of the root set. This will assure that the objects that they reference will be retained and left at the same address.

Possible Disadvantages of the New Algorithm

One concern about the new algorithm is that it might retain too much storage. By basing its decisions on hints and retaining all items on a page when a hint points to an object on a page, some amount of unneeded storage will be retained. A second concern is that too much storage may be locked in place, resulting in very little compaction. Before constructing a collector based upon these algorithms, one would like some assurance that one is neither constructing a "too-much-copying" collector, nor a "rarely-copying" collector.

Storage Retention

To get some understanding of possible storage retention problems, several different collectors for Titan Scheme were constructed. All collectors were based on algorithm II as they had to concern themselves with references contained in continuations.

The first collector, **MC-O**, was also the first "mostly-copying" collector constructed for Titan Scheme. At the time it was constructed, it was felt that significant steps should be taken to reduce retention of unnecessary storage. When the stack, register, and continuation cells are examined, the only objects that are considered to be a reference

to an object are those which are a valid, tagged pointer to an object.

Pointers are verified by checking that they have a valid tag and that they point to a page in the current space with the same type tag. Cons cell pointers must also be double word aligned. Pointers to other objects must point to the object header. Headers are verified by checking that the appropriate bit is set in an allocation bit map which is associated with the heap. A side table is used here because any bit pattern, including a valid header, could occur within a string. When a cell passes this pointer test, the page containing the object that it references is locked in place. However, that object is the only object that is traced to find further accessible storage.

Note that in order for this scheme to work, it requires that any object which has an untagged pointer in the root set also have a real pointer with the correct tag in the root set.

One can argue that the Titan implementation of this algorithm retains little or no unneeded storage. First, the stack will only contain Scheme pointers, stack pointers, and return addresses. A stack pointer or return address will never be confused with a Scheme pointer as they will always have the value of 0 in their low-order two bits, which is the tag for an immediate integer. Second, since the stack is always word aligned, only correctly aligned words need be examined as possible pointers. Thus, the registers are the only possible source of bogus pointers. As this implementation leaves very little to doubt, it is reasonable to believe its performance is similar to that of a classical stop-and-copy collector.

The second collector, **MC-II**, uses algorithm II found in this paper. Here, any item in the stack, registers, or a continuation which can be interpreted as a pointer into a page in the current space will lock the page and trace all items on the page.

Each of these collectors was then used to run two sample programs with varying page sizes. The sample programs were the Titan Scheme compiler and repeated executions of the Boyer benchmark [3]. The page size was varied from 128 to 4096 bytes. The effectiveness of each collector was measured by observing the number of times that garbage collection took place and the amount of storage that was retained after each collection.

While it is dangerous to draw too many conclusions from such a small sample, it does suggest a few things about these variants of the mostly-copying algorithm. For small pages, (less than or equal to 256 bytes), both collectors have similar behavior. As page size increased, MC-II retained too much data. For Boyer with a page size of 4096 bytes, this over-retention resulted in 50% more collections than MC-O. Performance differences with the Scheme compiler was no where near as extreme, though MC-II continued to be less efficient than MC-O. As expected, MC-O's behavior was independent of page size.

As the page size gets smaller, one concern is that more storage will be wasted because more fractional pages will have to be discarded during storage allocation. In these sample runs, the worst case waste was less than 2% of the heap which was observed when running the Scheme compiler with 128 byte pages.

Page Locking

Having shown that storage retention need not be a problem with mostly-copying collection, the problem of page locking will now be examined. The concern here is that too many pages will have to be locked which will result in too little compaction of storage.

The results of the previous section suggest that 512 bytes is a reasonable page size. For this page size, the worst case amount of heap being locked by any of the collectors was 2%. It is only by going to an extreme page size of 4096 bytes and using MC-II that too many pages were locked.

Summary

This paper has introduced a garbage collection algorithm which provides efficient compacting collection without requiring well-defined roots. The algorithm has been used within a Scheme system for the Titan, where the object code is an intermediate language.

While the discussion to this point has focused on Lisp, there is nothing in the algorithms which restrict them to Lisp's notions of data structures, nor is there anything which requires bookkeeping by the compiler. It is therefore reasonable to consider how to use it with other languages such as Modula-2 or C. First, the root set R must be identified. With compiler support, this could be done by declarations. Without it, the program could explicitly register such pointers with the collector or the collector could assume that the entire initial global area was to be included in R [8]. Second, the pointer fields in heap allocated objects must be known and valid. Third, the data structures must be self-identifying. Finally, the program cannot depend upon the numerical value of a pointer as objects will be relocated. While none of these requirements place a large burden on a program, they must be met in order for this type of collector to operate.

Finally, the "mostly-copying" collector compares favorably with the classical stop-and-copy algorithm in both processor and memory usage. Even though it has to "guess" which objects to keep, experience to date suggests that this does not lead to over-retention of storage.

References

[1] David H. Bartley, John C. Jensen.
The Implementation of PC Scheme.
In *1986 ACM Conference on LISP and Functional Programming*, pages 86-93. August, 1986.

[2] Jacques Cohen.
Garbage Collection of Linked Data Structures.
ACM Computing Surveys 13(3):341-367, September, 1981.

[3] Richard P. Gabriel.
Performance and Evaluation of Lisp Systems.
The MIT Press, 1985, pages 116-135.

[4] Timothy J. McEntee.
Overview of Garbage Collection in Symbolic Computing.
LISP Pointers 1(3):8-16, August-September, 1987.

[5] Jonathan Rees, William Clinger (Editors).
Revised^3 Report on the Algorithmic Language Scheme.
SIGPLAN Notices 21(12):37-79, December, 1986.

[6] Paul Rovner.
On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language.
Technical Report CSL-84-7, Xerox Palo Alto Research Center, July, 1985.

[7] David W. Wall, Michael L. Powell.
The Mahler Experience: Using an Intermediate Language as the Machine Description.
In *Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 100-104. October, 1987.

[8] Mark Weiser, Xerox Palo Alto Research Center.
Private communication, December 28, 1987.