Compacting Garbage Collection with Ambiguous Roots

Joel F. Bartlett

February, 1988

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

Abstract

This paper introduces a copying garbage collection algorithm which is able to compact most of the accessible storage in the heap without having an explicitly defined set of pointers that contain the roots of all accessible storage. Using "hints" found in the processor's registers and stack, the algorithm is able to divide heap allocated objects into two groups: those that might be referenced by a pointer in the stack or registers, and those that are not. The objects which might be referenced are left in place, and the other objects are copied into a more compact representation.

A Lisp compiler and runtime system which uses such a collector need not have complete control of the processor in order to force a certain discipline on the stack and registers. A Scheme implementation has been done for the Digital WRL Titan processor which uses a garbage collector based on this "mostly copying" algorithm. Like other languages for the Titan, it uses the Mahler intermediate language as its target. This simplifies the compiler and allows it to take advantage of the significant machine dependent optimizations provided by Mahler. The common intermediate language also simplifies call-outs from Scheme programs to functions written in other languages and call-backs from functions in other languages.

Measurements of the Scheme implementation show that the algorithm is efficient, as little unneeded storage is retained and only a very small fraction of the heap is left in place.

Simple pointer manipulation protocols also mean that compiler support is not needed in order to correctly handle pointers. Thus it is reasonable to provide garbage collected storage in languages such as C. A collector written in C which uses this algorithm is included in the Appendix.

> A further problem is the occasional difficulty of determining exactly what Lists are not garbage at any given stage; if the programmer has been using any nonstandard techniques or keeping any pointer values in unusual places, chances are good that the garbage collector will go awry.

Knuth, Volume I

1. Introduction

Garbage collection algorithms typically start by looking in known locations to find an initial set of pointers to objects in the heap. Using the pointers in this "root set", all accessible objects can then be located. The correct management of the locations containing these roots has been a serious concern for as long as garbage collected storage has existed [9]. While symbolic programming languages such as Lisp hide the mechanics of correct pointer management from the user, they still are a concern of the language implementer.

Many of the modern garbage collectors [10] recover space by copying accessible objects into some "new space". Two of the attractive properties of a copying collector are that it results in memory compaction and it can have a running time proportional to the amount of accessible storage [3]. However, such schemes place a large burden on the underlying system as not only must all objects be visible, but all pointers to the objects must be found and changed. In addition, a mechanism must be provided to update any values which have been derived from pointers. For example, a base register's contents which were computed by extracting the tag from a pointer to a vector would have to be recomputed when the vector is copied.

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 derived pointers do not exist as 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, derived pointers are a problem and 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 [15] 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.

To solve these problems, a copying collector has been devised which allows garbage collection 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 or derived 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 will divide 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.

In the sections which follow, existing practice is reviewed and the new algorithm is introduced, applied, and evaluated. A sample collector for use with C programs is provided in the Appendix.

1.1. Comparison with Existing Work

One scheme that does not need any roots to recover space is reference counting. In the classical method [9], each object contains a count of the number of references to it that exist. When a new reference is created, the count is incremented, and when a reference is deleted, the count is decremented. An object is explicitly freed when the reference count on the object becomes zero. While this method works, it has a few drawbacks. First, all objects must contain a reference count field. This field must be large enough to count the most possible references, or some overflow method must be provided. Second, a mechanism must be provided to maintain these reference counts. This can be done explicitly by the programmer, or implicitly by the compiler. Finally, this method cannot recover circular structures.

Significant improvements were made to this basic method, and a collector using reference counts was used for Interlisp [5]. In this collector, reference count maintenance is reduced because references which are on the stack are not counted. When space is being recovered, the stack contents are used as "hints" to retain items even though their reference counts were zero. Instead of maintaining counts within each object, counts are maintained in a separate side table. In this algorithm, the authors took the position that memory space was at a premium and processor power was cheap. In addition, they were able to microcode much of the implementation. While improving some of the aspects of the classical algorithm, the authors indicated that a compacting collector was still required to collect circular structures and compact memory. The collector they used is similar to the classical stop-and-copy collector and it requires a known set of roots in order to compact the heap.

A later reference count based storage system was done for Cedar [12]. While Cedar initially used a method similar to Interlisp's, a new collector was designed because the implementers felt that the Interlisp collector had too high an execution cost, and was too complex. Even though they were aware of proposals for managing reference counts in circular structures [2], they chose to implement a conventional trace-and-sweep (also known as mark-and-sweep) collector to reclaim inaccessible circular structures as it was simple and it was unclear how frequent circular structures were. Looking back at this decision, the author states that this may have been a mistake as circular structures appear frequently in some applications and the trace-and-sweep collections become disruptive.

If a set of objects is available which include root pointers to all accessible storage, a mark-and-sweep collector [9] can be constructed. Each object which might be a root is treated in a conservative manner [12]. That is, objects that might be valid pointers are treated as pointers for purposes of storage retention. As this type of collector will never move 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.

Recently however, there has been more interest in building collectors that are willing to deal with "bad roots". In parallel with the work described in this paper, a conservative trace-and-sweep collector is being constructed [16] for use in arbitrary environments. While it seems to pose no constraints on the client programs, it does not compact the heap. Also, one of the reviewers of this paper indicated that he was exploring the use of ideas similar to those expressed here in a concurrent collector [4].

With the overview of earlier and concurrent work completed, our attention will now turn to the new algorithms.

1.2. 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" [3]. In discussing the algorithms, we will restrict ourselves to the allocation of fixed-length cells containing two pointers, i.e. a Lisp cons cell. Each pointer in the cell will assumed to be either a pointer or NULL. The C [7] declaration for the cell is:

```
typedef struct cons_cell {
          struct cons_cell *car;
          struct cons_cell *cdr;
       } *CP, CONS_CELL;
```

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. Garbage collection occurs when all cells in the semispace have been allocated. Assuming the existence of a pointer to the last cell of the current semispace, *endspace*, then the following function could allocate storage:

```
CP cons( car, cdr )
        CP  car, cdr;
{
    CP p;

    if  (freespace > endspace)  collect();
    p = freespace;
    freespace = freespace+1;
    p->car = car;
    p->cdr = cdr;
    return( p );
}
```

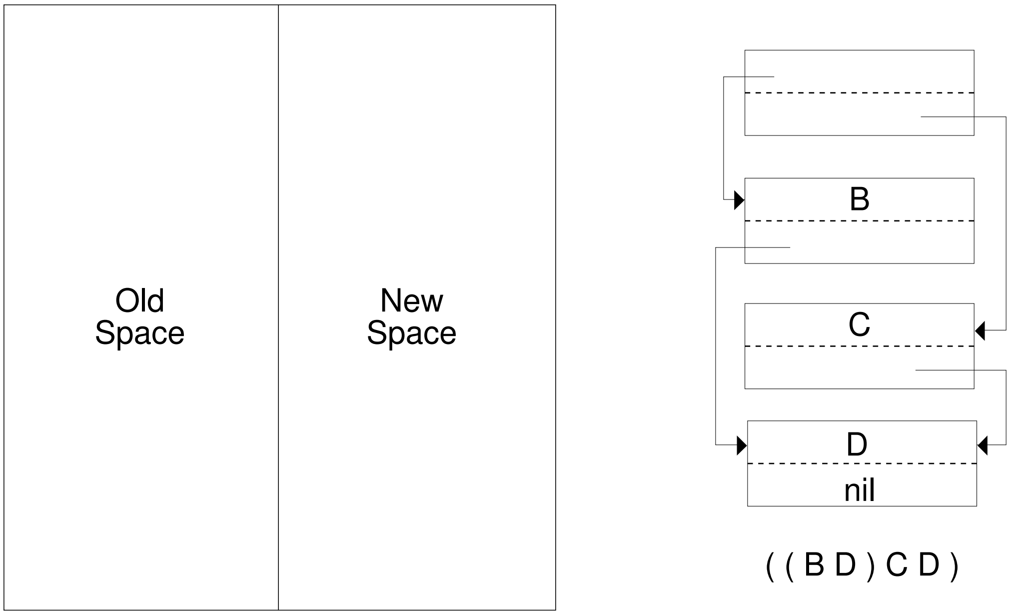

Figure 1: Stop-and-Copy memory organization and sample list

When free space is exhausted, the garbage collector is invoked. The first thing that it does is "flip" the semispaces, i.e. exchange old space and new space:

```
freespace = old_first;
endspace = old_last;
old_first = new_first;
old_last = new_last;
new_first = freespace;
new_last = endspace;
sweep = freespace;
```

Here, *..._first* points to the first available cell in a semispace, and *..._last* points to the last available cell in a semispace. A pointer to the beginning of the newspace is saved in *sweep*.

Following this, the collector moves all accessible objects into the new space. This is done by examining a set of known cells, *root*, to find all immediately accessible items. Each referenced item is moved by *move* (described below) and the contents of the root cell are changed. Since the root cell is changed, the program must assure that each root cell always contains a valid pointer.

```
for  (i = 0;  i < root_count;  i = i+1)  {
   root[ i ] = move( root[ i ] );
}
```

Objects are moved into the new space by *move*, which works as follows. If the pointer is NULL or it points to an object already in the new space, then no conversion is necessary. Otherwise, the object is examined to see if it contains a forwarding pointer to the copy of the object in new space. If so, then the pointer to the new space copy is returned. Failing these tests, the object must be copied to new space. The new cell is allocated with the contents of the old cell, and a forwarding pointer is left in the old cell. Note that when an object is copied into the new space, its constituent objects are not copied at the same time.

```
CP  move( cp )
        CP  cp; /* Pointer to a cons cell */
{
        /* NULL or objects already in the new space are OK */
        if  (cp == NULL  ||  (new_first <= cp  &&  cp <= new_last))
           return( cp );
        /* Return forwarding ptr for copied objects */
        if  (new_first <= cp->car  &&  cp->car <= new_last)
           return( cp->car );
        /* Copy object to new space, place forwarding ptr in old */
        cp->car = cons( cp->car, cp->cdr );
        return( cp->car );
}
```

Given that somewhere in *root* 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.

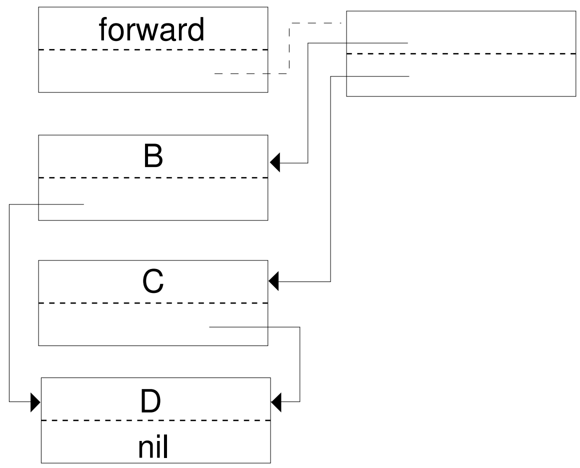

Figure 2: Move the objects referenced by the roots

Once the initial items have been moved, the items that they reference are moved (see Figure 3). Since storage is sequentially allocated in new space, this can be done by sweeping across the new space and moving each pointer of each cell. Once all pointers in the new space have been moved, the garbage collection is complete. Storage is then allocated out of new space until it is exhausted, at which point the garbage collector is again invoked.

```
while  (sweep != freespace)  {
   sweep->car = move( sweep->car );
   sweep->cdr = move( sweep->cdr );
   sweep = sweep+1;
}
```

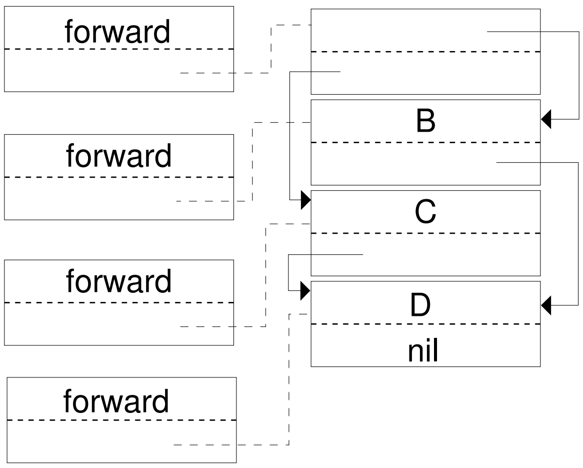

Figure 3: Move the rest

2. 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 how it finds its root set and how it organizes the heap.

The classical algorithm requires that one provide a set of cells, *R*, which contains valid pointers to find all accessible storage. While this can be done with an array of pointers as in the previous example, it is typically done by extracting *R* from the current program state which is contained in the processor's registers and stack. In order to do this, the collector must know how the registers and stack are used. At any given time, it must be able to correctly identify those registers which contain pointers or derived pointers. Similarly, the stack format must be known, as all items which are pointers or derived pointers must be found amongst the stack frames, local variables, and procedure arguments. In summary, all heap pointers must be found and changed, and no other objects, including those which "look like" pointers, found or changed.

The new algorithm makes very 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 simply 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 with *PAGEBYTES* bytes per page (see Figure 4). This page size is **independent** of the underlying hardware's page size. The page number of the first page of the heap is in *firstheappage* and the page number of the last page of the heap is *lastheappage*. A pointer can be converted to a page number by *CP_to_PAGE* and a page number can converted to a pointer by *PAGE_to_CP*.

Associated with each page is a space identifier, *space*, which identifies the "space" that objects on the page belong to. This level of indirection means that there are two ways to move an object from old space to new space. It can be copied as in the previous algorithm, or the page containing it can have its space identifier changed to that of the new space. 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.

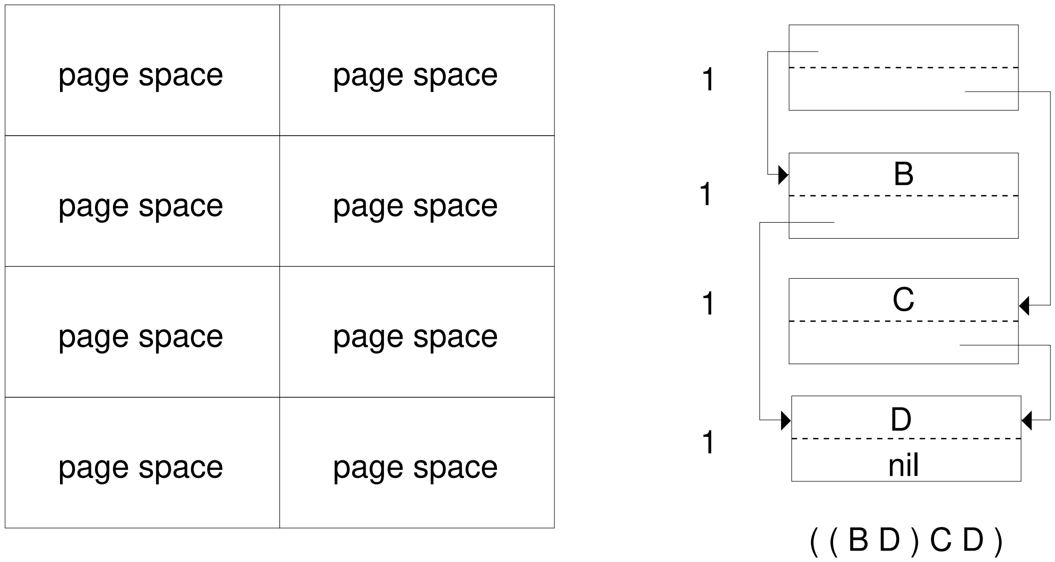

Figure 4: Mostly-copying memory organization and sample list

Like the classical algorithm, this one needs a way to sweep across all objects in the "new space". Since pages in this space do not have contiguous addresses, they are formed into a list using the link field, *link*, which is associated with each page.

Two space identifiers: *current_space* (initially = 1), and *next_space* (initially = 1), identify important sets of pages. During normal allocation, *current_space* and *next_space* are equal and pages are identified with their value when allocated. During garbage collection, *next_space* is set the the "next" space identifier. As objects are copied, they are placed in pages identified by it. Thus it is reasonable to think of *next_space* as the analogue of "new space", and *current_space* as the analogue of "old space" when comparing this algorithm to the classical one.

With memory broken up into pages, cons cell allocation is a two part process: first, allocate a page of memory, and then allocate space from it. Allocation from within a page is controlled by *consp* which points to the next available cons cell and *conscnt* (initially = 0) which is a count of the number of cons cells available. The allocation function is:

```
CP cons( car, cdr )
        CP  car, cdr;
{
        CP  p;

        while  (conscnt == 0)  allocatepage();
        p = consp;
        consp = consp+1;
        conscnt = conscnt-1;
        p->car = car;
        p->cdr = cdr;
        return( p );
}
```

When a page needs to be allocated, the heap is searched starting at *freepage* (initially = *firstheappage*). Pages whose *space* field (initially = 0) is not equal to *current_space* and not equal to *next_space* are considered to be free. When a page is allocated, its *space* field is set to *next_space* and *allocatedpages* is incremented. The page is also added to the tail of the page list (*queue_head* and *queue_tail*) using its *link* field if garbage collection is in progress. The C code to perform these operations is:

```
int next_page( page )
        int  page;
{
        if  (page == lastheappage)  return( firstheappage );
        return( page+1 );
}

queue( page )
        int  page;
{
        if  (queue_head != 0)
           link[ queue_tail ] = page;
        else
           queue_head = page;
        link[ page ] = 0;
        queue_tail = page;
}

allocatepage()
{
        if  (allocatedpages == HEAPPAGES/2)  {
           collect();
           return;
        }
        while  (space[ freepage ] == current_space  ||
                space[ freepage ] == next_space)
           freepage = next_page( freepage );
        conscnt = PAGEBYTES/sizeof(CONS_CELL);
        consp = PAGE_to_CP( freepage );
        space[ freepage ] = next_space;
        allocatedpages = allocatedpages+1;
        if  (current_space != next_space)  queue( freepage );
        freepage = next_page( freepage );
}
```

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. This is done by incrementing it modulo *X*. This number, *X*, must be large enough to assure that the *freepage* index will completely sweep all of memory before a space identifier value is ever reused.

```
/* Advance space */
next_space = (current_space+1) & 077777;
allocatedpages = 0;
```

Next, the garbage collector will make 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 moved into the next space by changing the page's space identifier (see Figure 5). The page is also added to the tail of the list of pages in the next space. Thus the objects which might have references in the stack or registers are now part of the next space, but their address has not changed. The code to do this is:

```
promote_page( page )
        int  page;
{
        if  (page >= firstheappage  &&  page <= lastheappage  &&
             space[ page ] == current_space)  {
           space[ page ] = next_space;
           allocatedpages = allocatedpages+1;
           queue( page );
        }
}
queue_head = 0;
for (fp = FRAMEPTR ; fp != STACKBASE ; fp = fp+1)  {
   promote_page( CP_to_PAGE( *fp ) );
}
for (reg = FIRSTREG ; reg <= LASTREG ; reg = reg+1)  {
   promote_page( CP_to_PAGE( processor_register( reg ) ) );
}
```

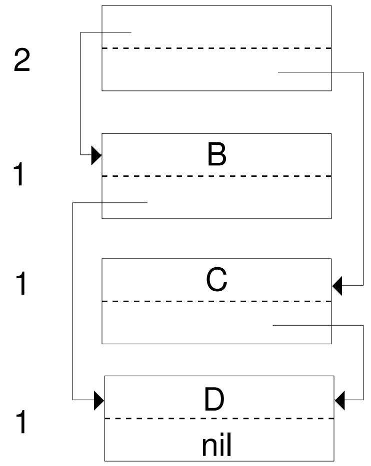

Figure 5: Move possibly referenced pages to next space

At the completion of this phase, all pages containing items which might be referenced by pointers in the stack or registers have been promoted to the next space. In addition, some amount of storage which is not needed may also have been saved. Once these initial items have been moved, the component items of these promoted items are now moved (see Figure 6). This is done by sweeping across all pages in the next space and moving their pointers:

```
while  (queue_head != 0)  {
   cp = PAGE_to_CP( queue_head );
   cnt = PAGEBYTES/sizeof(CONS_CELL);
   while (cnt != 0  &&  cp != consp)  {
      cp->car = move( cp->car );
      cp->cdr = move( cp->cdr );
      cp = cp+1;
      cnt = cnt-1;
   }
   queue_head = link[ queue_head ];
}
```

The *move* function differs from the *move* function in the stop-and-copy collector only in how it tests for a pointer being in the *next_space*, i.e. new space. As before, objects are copied at most once and forwarding pointers are left in the old objects.

```
CP move( cp )
        CP  cp;
{
        if  ((cp == NULL)  ||
             (space[ CP_to_PAGE( cp ) ] == next_space))
           return( cp );
        if  (space[ CP_to_PAGE( cp->car ) ] == next_space)
           return( cp->car );
        cp->car = cons( cp->car, cp->cdr );
        return( cp->car );
}
```

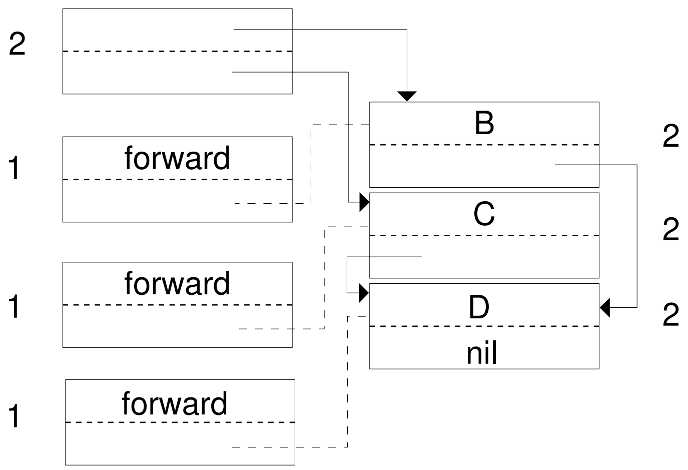

Figure 6: Move the rest of the accessible objects

Once all pointers in the next space have been corrected, *current_space* is changed and garbage collection is complete:

```
current_space = next_space;
```

2.1. A Problem - Continuations

While a garbage collector built on the ideas of the last section might be adequate for many applications, it could have difficulty with continuations in Scheme [11]. A continuation is a procedure which returns the program to some previous state of the computation. It can provide a more powerful control mechanism than the Common Lisp CATCH/THROW [13] or the C setjmp/longjmp [8], as it is more general than an "upexit" and may be repeatedly invoked.

One way of creating continuations is to copy the program state found in the registers and the stack to a heap allocated data structure [1]. When the continuation is invoked, the saved stack and register contents are restored. While this has the disadvantage of having to copy data, the format of the stack and the use of the registers need not be known.

As continuations are first class objects, they may be located anywhere in the heap and need not be visible at the start of garbage collection. However, they contain "hints" which may reference objects in the heap. Were these objects to move or disappear, the continuation would not correctly execute. Thus, the program state saved in the continuation must be examined in the same way that the program state currently visible in the registers and stack is examined.

A collector based upon algorithm I would attempt to collect a heap containing continuations as follows. First, it would examine the stack and registers to decide the set of pages to retain, P. Next, it would copy all objects to the next space. During this process, when a continuation is found, the saved state contained in it would be examined to see what heap pages it needed retained. However, there is no guarantee that these pages are contained in P, and therefore items on some of these pages may already have been copied which would be incorrect.

Algorithm I requires that all retained pages be discovered before any objects are copied. This cannot be done here as all continuations can only be discovered by starting to copy. This contradiction is resolved by introducing a second algorithm.

3. Mostly-Copying Collector II

This description will use the cons cell data structure of the last two examples, with a small addition. Pointers are now allowed to reference objects outside the heap. Such a pointer is assumed to point to a cell which might contain a pointer which references an object in the heap. The cell might also contain an object which looks like a pointer, but isn't. Therefore it must be treated as a "hint", like the registers and the stack. Thus additional pages which must remain in place can be found at any time while the garbage collector is moving data.

Functions for storage allocation and many of the primitive operations are identical to those used in algorithm I, so they will not be repeated here. Garbage collection starts by advancing the allocation space and searching the stack and the registers for initial roots:

```
/* Advance space */
next_space = (current_space+1) & 077777;
allocatedpages = 0;

/* Examine stack and registers for possible pointers */
queue_head = 0;
for (fp = FRAMEPTR ; fp != stackbase ; fp = fp+1)  {
   promote_page( CP_to_PAGE( *fp ) );
}
for (reg = FIRSTREG ; reg <= LASTREG ; reg = reg+1)  {
   promote_page( CP_to_PAGE( processor_register( reg ) ) );
}
```

Pages are "promoted" to the next space in a different manner from the first algorithm:

```
promote_page( page );
        int  page;
{
        if  (page >= firstheappage  &&  page <= lastheappage  &&
             space[ page ] == current_space  &&
             promoted[ page ] == 0)  {
           allocatedpages = allocatedpages+1;
           promoted[ page ] = 1;
           queue( page );
        }
}
```

Here, the space number is not changed, and instead a boolean, *promoted*, is set for the page. As in algorithm I, the page is also queued.

Once the initial roots have been found, the objects they directly reference are copied and the pointers in newly allocated objects are also copied until all accessible items have been copied to the "next space" (see Figure 7 and the code which follows).

When comparing the copying code below with that of the previous algorithm, two things should be noted. First, promoted pages and newly allocated pages are not treated the same. Since promoted pages are not a part of the new space, their contents must be copied. With newly allocated pages, on the other hand, only their constituent pointers need be copied. The second thing to note is that the pointers in the pages are not updated. Thus all pointers in the copied objects are indirect via the forwarding pointers contained in the old cells.

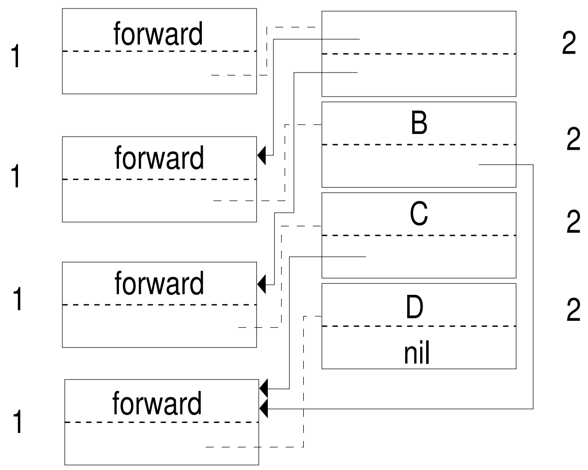

Figure 7: Copy all accessible items

```
/* Sweep promoted pages and move their constituent items */
i = queue_head;
while (i != 0) {
    cp = PAGE_to_CP( i );
    cnt = PAGEBYTES/sizeof(CONS_CELL);
    while (cnt != 0  &&  cp != consp) {
        if (promoted[ i ] == 1) {
            copy( cp );
        }
        else {
            copy( cp->car );
            copy( cp->cdr );
        }
        cp = cp+1;
        cnt = cnt-1;
    }
    i = link[ i ];
}
```

The *copy* function differs slightly from the *move* function of the previous algorithm. First, it does not return the pointer to the copied object, and second, it must handle the additional case of newly discovered roots:

```
copy( cp )
        CP  cp;
{
        int     page;

        /* OK if pointer is NULL or points into next space */
        if (cp == NULL ||  space[ CP_to_PAGE( cp ) ] == next_space)
           return;
        /* If pointer points outside the heap, then found
           another root */
        page = CP_to_PAGE( cp );
        if (page < firstheappage  ||  page > lastheappage) {
           promote_page( CP_to_PAGE( cp->car ) );
           return;
        }
        /* OK if cell is already forwarded */
        if (cp->car != NULL  &&
           space[ CP_to_PAGE( cp->car ) ] == next_space)
           return;
        /* Forward cell, leave forwarding ptr in car of
           old cell */
        cp->car = cons( cp->car, cp->cdr );
}
```

Once all accessible items have been copied, all promoted pages are known (highlighted in Figure 8). Using this information, a second sweep is made over the objects in the new space and their pointers are corrected (see Figure 9). During this phase, a list of promoted pages is formed:

```
i = queue_head;
promoted_head = 0;
while (i != 0)  {
   if  (promoted[ i ] == 1)  {
      x = link[ i ];
      link[ i ] = promoted_head;
      promoted_head = i;
      i = x;
   } else {
      cp = PAGE_to_CP( i );
      cnt = PAGEBYTES/sizeof(CONS_CELL);
      while (cnt != 0  &&  cp != consp)  {
         cp->car = correct( cp->car );
         cp->cdr = correct( cp->cdr );
         cp = cp+1;
         cnt = cnt-1;
      }
      i = link[ i ];
   }
}
```

Pointer correction is done by the following function. If the pointer points to an old page in the heap and the page was not promoted, then the correct pointer is the forwarding pointer found in the object.

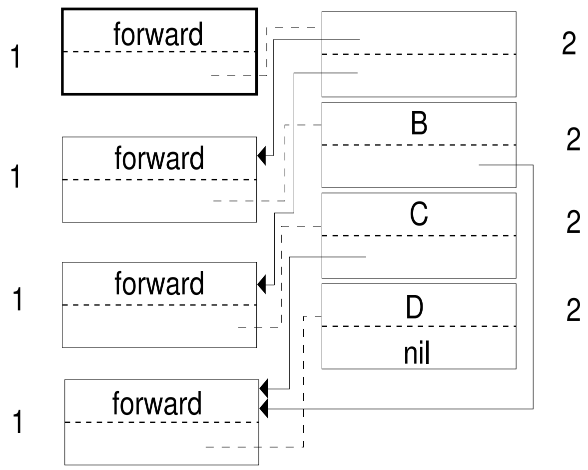

Figure 8: All promoted pages now known

```
CP  correct( cp )
        CP  cp;
{
        int  page;

        page = CP_to_PAGE( cp );
        if (page >= firstheappage  &&
            page <= lastheappage  &&  promoted[ page ] == 0)  {
           return( cp->car );
        }
        return( cp );
}
```

Following the correction phase, the promoted pages must have their contents restored (see Figure 10). This is done by copying back each object on a promoted page using the forwarding pointer left in the object. Setting the new value of *current_space* completes the collection.

```
i = promoted_head;
while  (i != 0)  {
   promoted[ i ] = 0;
   cp = PAGE_to_CP( i );
   cnt = PAGEBYTES/sizeof(CONS_CELL);
   while  (cnt != 0  &&  cp != consp)  {
      zp = cp->car;
      cp->car = zp->car;
      cp->cdr = zp->cdr;
      cp = cp+1;
      cnt = cnt-1;
   }
   space[ i ] = next_space;
   i = link[ i ];
}

current_space = next_space;
```

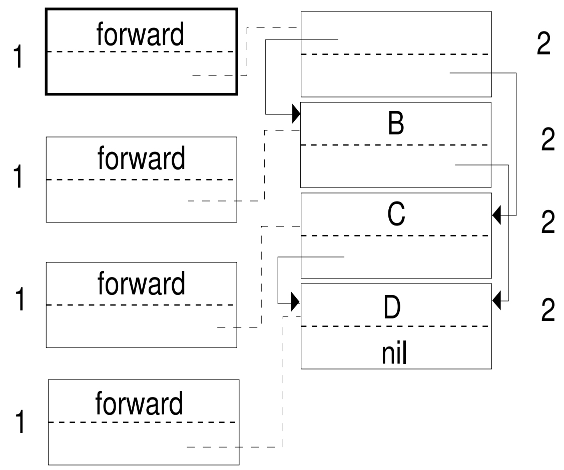

Figure 9: Correct pointers

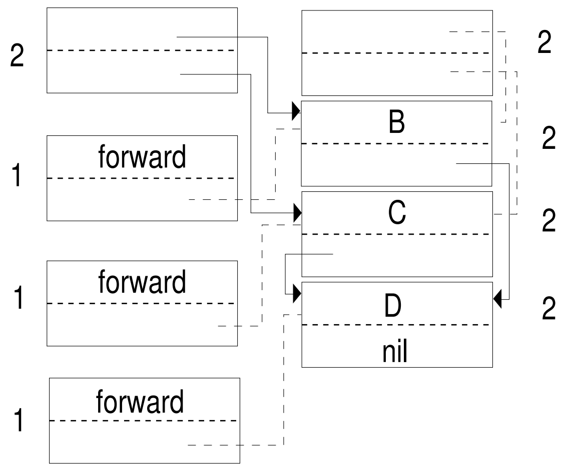

Figure 10: Copy back contents of promoted pages

4. 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 the Mahler intermediate language [15] which is the object code produced by all compilers. When the Mahler code is compiled, extensive machine dependent optimization is done. This includes allocation of frequently used local storage to registers, global register allocation [14], and code reordering to overlap execution with the arithmetic coprocessor, fill branch slots, and avoid pipeline stalls. 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.

4.1. Adding More Types of Storage

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. A cons cell is allocated as shown earlier, and other objects are allocated as follows. Objects larger than a page are allocated as many pages as required, with some portion of the last page potentially unused. 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. When discarding it, the unused space is marked as a string so that the garbage collector can scan the page correctly. Experience with Titan Scheme has shown that discarding partial pages in this manner does not waste significant amounts of space.

4.2. Applicability to Other Languages

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. If global variables or own variables are used to hold pointers to heap allocated objects, then neither of the current two algorithms will work as these cells will not be in the root set *R*. If compiler support is available, then such pointers could be declared as "pointers to heap allocated objects", and information automatically left in some appropriate place by the compiler for use by the garbage collector. Without compiler support, the programmer could explicitly register such pointers with the collector as is done in the sample collector in the Appendix. If neither of these approaches were desired, then the collector could assume that the entire initial global area was to be included in *R*. This is the approach that has been taken in [16].

Some language environments also have their own ideas of what heap allocated storage looks like. For example, run-time pointer validation may be done by checking the pointer against a header stored just before the object. This header might have to be changed when the object is relocated. As the new algorithms require that the objects be self-identifying when the page is swept, these features could be provided.

Finally, the algorithms do require that the pointer fields in heap allocated objects be known and that they be valid. In the sample collector, all objects contain a fixed number of pointers which must be located at the head of the object. However, other mechanisms for pointer identification can be used. For example, a very general mechanism would be for the collector to call a user (or compiler) provided procedure to locate the pointers in the object.

None of these requirements place a large burden on a program, but they must be met in order for this type of collector to operate.

4.3. A Sample Collector

The Appendix of this document contains the complete listing for a C version of Algorithm I that implements a storage allocator similar to the C-library function *malloc*. Once the heap is initialized by calling *gcinit*, storage is allocated by calling *gcalloc*. It is called with the size of the object in bytes, and a count which is the number of pointers to other heap allocated objects which are contained in the object. By convention, these pointers must occupy the initial words of the object. For further details about the calling sequence of these functions, the reader is directed to the Appendix.

In order to customize this code for a particular system, the user must supply some processor specific information. The first is *STACKINC* which is used to specify the alignment of pointers on the stack. The collector assumes that a pointer may be stored at any location whose byte-address modulo STACKINC is zero. In the case of the Titan, STACKINC is 4 as the stack only contains 32-bit words. While a VAX processor can increment the stack pointer by one byte, compilers typically keep it aligned on a four-byte boundary, so a value of 4 is used for it too. Processors which do not keep pointers aligned on four-byte boundaries will have to have examine the stack in smaller increments. Tests run with Titan Scheme indicate that only small amounts of additional storage are retained even with the assumption that a pointer could start at any byte address in the stack.

The second item that must be provided is information about registers which may contain pointers to heap based objects when additional storage is being allocated. Since Titan programs can retain globals in registers and pass procedure arguments in registers, the registers must be inspected. This is enabled by defining register numbers *FIRST_REGISTER* and *LAST_REGISTER*, and a function for obtaining the value of a register, *register_value*. In the case of the VAX, no register information need be specified. This is because the VAX employs a "caller save" protocol to retain register values across procedure calls, so no valid pointers are ever in the registers during storage allocation.

5. Comparison with the Classical Algorithm

The new algorithm is similar to the classical algorithm in its resource demands [3]. It requires slightly more storage in the form of the space, link, type, and promoted fields associated with each page. It would not be unreasonable to store these 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 trace-and-sweep collector which is $O(m)$, where m is the total size of the heap.

5.1. 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 derived pointers. As the new algorithm does not differentiate between pointers which point to an object and those which point within an object, cells which might contain a derived pointer are made part of the root set. This will assure that the objects that they reference will be retained and left at the same address.

5.2. 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. In the sections which follow, experience with Titan Scheme is reported which shows that these are not serious problems.

5.3. 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, **GC-0**, 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 pointer to an object.

In order for this scheme to work, it requires that any object which has a derived pointer in the stack or registers also have a real pointer with the correct tag in the stack or registers. 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. This is 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 furthur accessible storage.

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, **GC-1**, 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. This is the simplest and least selective algorithm.

The final two collectors are variants of the first two. The third collector, **GC-2**, extends GC-0 by handling derived pointers. Any item which could be interpreted as a pointer into an object will lock the page holding the object and then trace that object. As before, this collector uses the allocation side table to find object headers.

The final collector, **GC-3**, is a combination of GC-0 and GC-1. Items which can be interpreted as pointing to cons cells cause the page containing the cell to be locked, and the referenced cell is traced (like GC-0). Items which could point to other objects lock the referenced page, and trace all objects on the page (like GC-1). Unlike GC-2 and GC-0, this variation of the algorithm does not require the maintenance of an allocation side table. Cons cells are treated as a special case because they are regular in size,the most common item, and are never accessed via a derived pointer.

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 [6]. 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), all collectors have similar behavior as shown in Figures 11 and 14.

For a cons-intensive program such as Boyer, the page scan done by GC-1 starts retaining too much data for page sizes greater than 256 bytes (see Figure 12). As the page size continues to increase, GC-1 continues to get less efficient. At the same time, the other collectors have con-

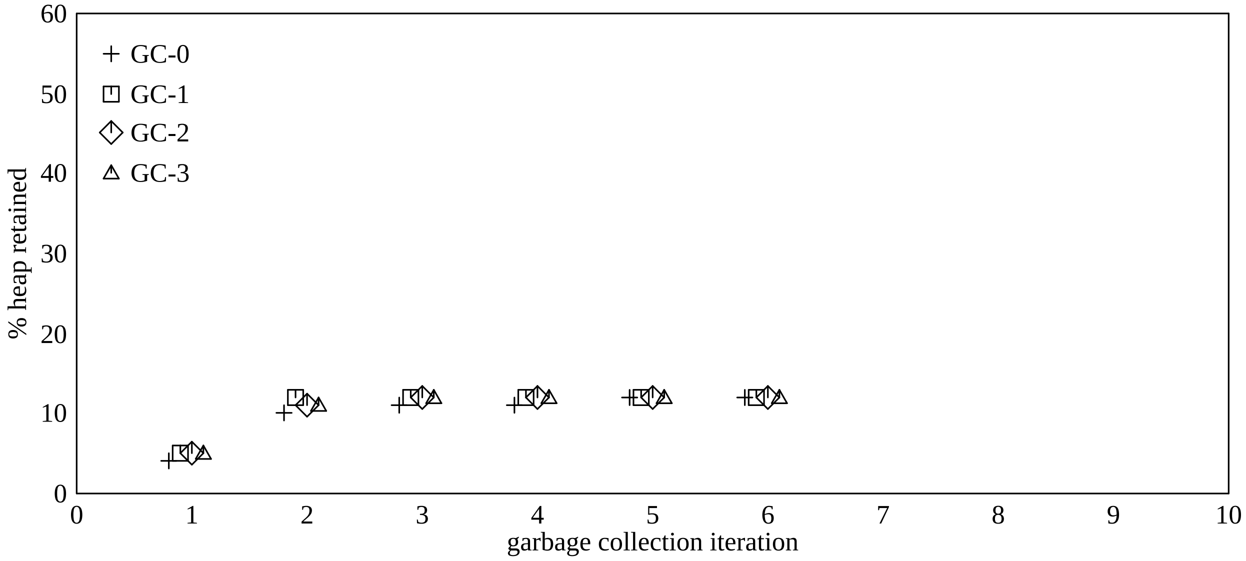

Figure 11: BOYER with 256 byte pages

stant performance, irrespective of the page size (see Figure 13). This occurs because the Titan Scheme implementation does not use derived pointers to reference cons cells and only cons cells are being allocated which makes GC-0, GC-2, and GC-3 equivalent.

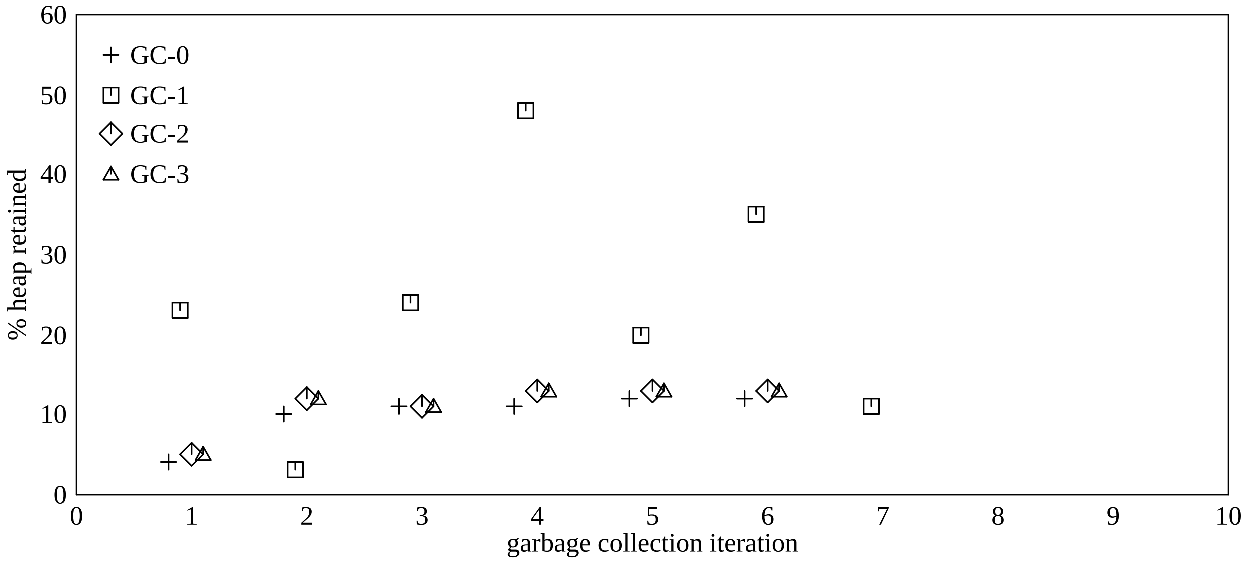

Figure 12: BOYER with 512 byte pages

The Scheme compiler allocates a variety of types of storage and might be considered a more normal program than Boyer. For small pages, (less than or equal to 256 bytes), all collectors have similar behavior (see Figure 14). As page size increases, the differences in the collectors become more marked, but not as extreme as with Boyer (see Figure 15).

Figure 13: BOYER with 4096 byte pages

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 compiler with 128 byte pages.

While this is a small sample, it does suggest a few things. For small page sizes, all collectors have similar performance, so the differences in "hint filtering" between the algorithms is not apparent. As page size grows, GC-1's crudeness becomes apparent. However, a small modification of it, GC-3, shows almost as much promise as the most selective algorithms of GC-0 and GC-2.

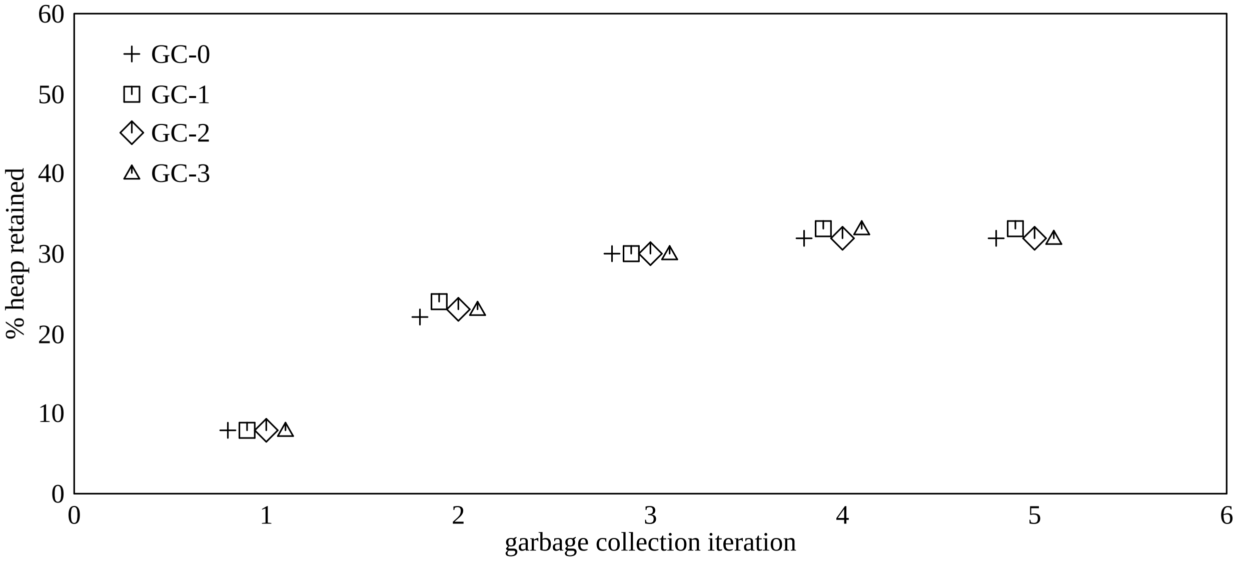

Figure 14: Titan Scheme Compiler with 256 byte pages

Figure 15: Titan Scheme Compiler with 4096 byte pages

5.4. 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. In Figures 16 and 17 we see that the worst case amount of heap being locked by any of the collectors is 2%. It is only by going to an extreme page size of 4096 bytes that one sees suggestions of over-locking. In Figures 18 and 19, GC-1 and GC-3 lock 18% at one collection iteration.

As page size gets larger, a larger amount of the heap must be locked down initially. This increases the amount of work that must be done during the copy back portion of the algorithm and reduces the amount of space that is freed as the storage used by items which were copied back cannot be released.

As before, one can conclude that GC-0 and GC-2 are insensitive to page size. This is as expected as they employ the most selective hint filtering. Collectors GC-1 and GC-3 are sensitive to page size, but page sizes can be selected which exhibit satisfactory performance.

5.5. Observations

The performance observed to date indicates that the algorithms employed by GC-0 and its derivative GC-2 are much more selective (and thus complex) than is needed. The simpler algorithms employed by GC-1 and its derivative GC-3 offer similar performance with an appropriately selected page size. Experience to date suggests that a page size of 512 bytes is a good initial choice.

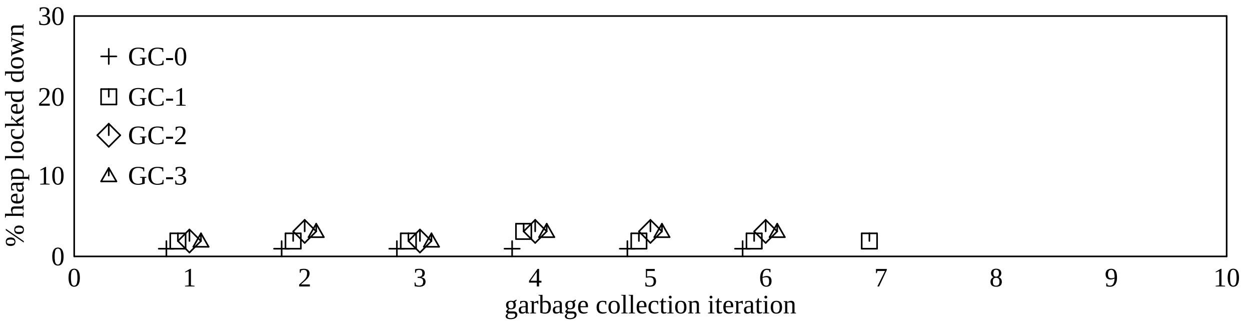

Figure 16: BOYER with 512 byte pages

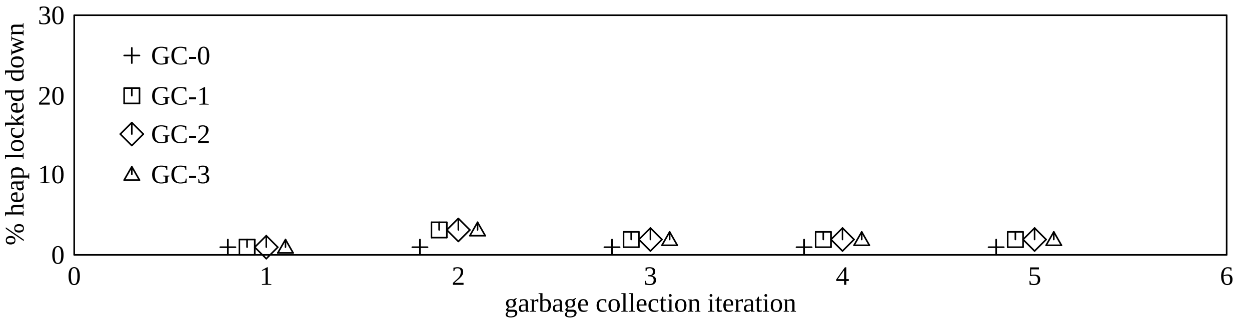

Figure 17: Titan Scheme Compiler with 512 byte pages

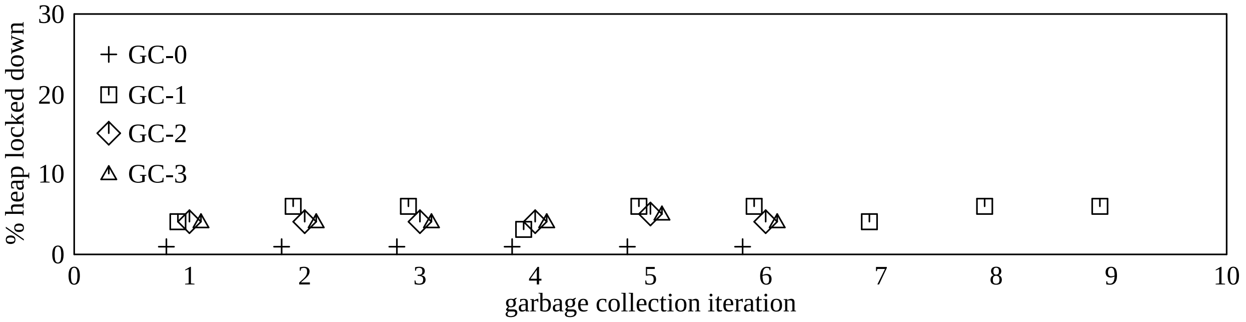

Figure 18: BOYER with 4096 byte pages

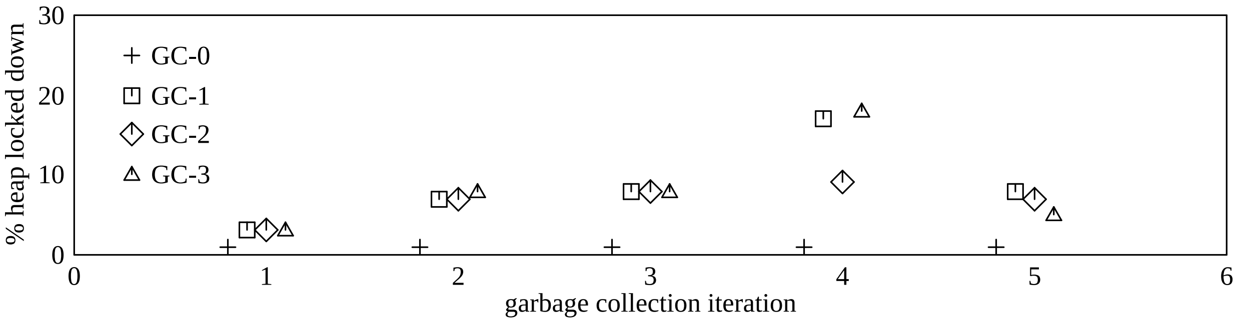

Figure 19: Titan Scheme Compiler with 4096 byte pages

6. Summary

This paper has introduced a garbage collection algorithm which provides efficient compacting collection without requiring well-defined roots. It 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, this does not result in over-retention of storage.

The algorithm has been used within a Scheme system for the Titan, where the object code is an intermediate language. Its applicability to other languages and environments has been shown.

The sample collector in the Appendix has been supplied to aid in understanding the collection algorithm and to encourage the reader to try it in their environment. Garbage collected storage is entirely too useful an idea to be confined to the Lisp community!

Acknowledgements

Wendy Bartlett read and criticized several drafts of this paper, each time substantially improving the presentation. John DeTreville, John Ousterhout, Walter van Roggen, and David Wall read later drafts and suggested additional improvements. I thank you all.

I. A Mostly-Copying Collector for C

```
/* This module implements garbage collected storage for C programs using the
   "mostly-copying" garbage collection algorithm.

   Copyright (c) 1987, Digitial Equipment Corp.

   The module is initialized by calling:

        gcinit( <heap size>, <stack base>, [ <global>, ... ,] NULL )

   where <heap size> is the size of the heap in bytes, and <stack base> is
   the address of the first word of the stack which could contain a pointer
   to a heap allocated object.  Following this are zero or more addresses of
   global cells which will contain pointers to garbage collected objects.
   This list is terminated by NULL.

   Once initialized, storage is allocated by calling:

        gcalloc( <bytes>, <pointers> )

   where <bytes> is size of the object in bytes, and <pointers> is the number
   of pointers into the heap which are contained in the object.  The pointers
   are expected to be at the start of the object.  The function will return a
   pointer to the data structure with its pointer cells initialized to NULL.
   For example, an instance of the structure:

        struct  symbol  {
           struct *symbol  next;
           char name[10];
        }

   could be allocated by:

        sp = (symbol*)gcalloc( sizeof( symbol ), 1 );

   When the garbage collector is invoked, it will search the processor’s
   registers, the stack, and the global pointers for "hints" as to what
   storage is still accessible.  The hints from the registers and stack will
   be used to decide which storage should be left in place.  Note that objects
   which are referenced by global pointers might be relocated, in which case
   the pointer value will be modified.

   N.B.  This code assumes that pointers and integers are 32-bits long.  It
         also handles a variable number of arguments in a machine dependent
         manner.  Define the variable VAX for VAX code, or TITAN for Titan
         code.
*/

/* Exported items. */

typedef int  *GCP;      /* Type definition for a pointer to a garbage
                           collected object. */

extern  gcinit( /* <heap size in bytes>, <address of stack base>,
                   [ <address of global ptr>, ...] NULL  */ );
```

```
extern  GCP gcalloc( /* <bytes> , <# of pointers> */ );

/* External definitions */

#include <stdio.h>
extern  char *malloc( /* <# of bytes required> */ );

/* ******************************** */
/* Processor dependent definitions. */
/* ******************************** */

/* VAX */

#ifdef VAX

/* Assume stack alignment on 32-bit words. */

#define  STACKINC  4

/* No pointers will be in registers during garbage collection, so no register
   definitions need be supplied.
*/

#endif

/* Titan */

#ifdef TITAN

/* Stack is 32-bit word aligned */

#define  STACKINC  4

/* Assume that any register may contain a pointer.*/

#define FIRST_REGISTER 0        /* First register to scan. */
#define LAST_REGISTER  60       /* Last register to scan. */

/* The following function is called to read one of the Titan registers.  It
   must be open-coded using constant register numbers as zzReadRegister is
   actually a Mahler inline function which expects a constant register
   number.
*/

extern  zzReadRegister( /* <constant register #> */ );

unsigned  register_value( regnum )
{
        switch (regnum)  {
                case  0: return( zzReadRegister(  0 ) );
                case  1: return( zzReadRegister(  1 ) );
                case  2: return( zzReadRegister(  2 ) );
                        .
                        .
                        .
                case 62: return( zzReadRegister( 62 ) );
                case 63: return( zzReadRegister( 63 ) );
```

```
                default: return( 0 );
        }
}

#endif

/* The heap consists of a contiguous set of pages of memory. */

int     firstheappage,  /* Page # of first heap page */
        lastheappage,   /* Page # of last heap page */
        heappages,      /* # of pages in the heap */
        freewords,      /* # words left on the current page */
        *freep,         /* Ptr to the first free word on the current page */
        allocatedpages, /* # of pages currently allocated for storage */
        freepage,       /* First possible free page */
        *space,         /* Space number for each page */
        *link,          /* Page link for each page */
        *type,          /* Type of object allocated on the page */
        queue_head,     /* Head of list of pages */
        queue_tail,     /* Tail of list of pages */
        current_space,  /* Current space number */
        next_space,     /* Next space number */
        globals;        /* # of global ptr’s at globalp */

unsigned *stackbase;    /* Current base of the stack */

GCP     *globalp;       /* Ptr to global area containing pointers */

/* Page type definitions */

#define OBJECT 0
#define CONTINUED 1

/* PAGEBYTES controls the number of bytes/page */

#define PAGEBYTES 512
#define PAGEWORDS (PAGEBYTES/sizeof(int))
#define WORDBYTES (sizeof(int))

/* Page number <--> pointer conversion is done by the following defines */

#define PAGE_to_GCP( p ) ((GCP)((p)*PAGEBYTES))
#define GCP_to_PAGE( p ) (((int)p)/PAGEBYTES)

/* Objects which are allocated in the heap have a one word header.  The
   form of the header is:

         31           17 16             1 0
        +---------------+----------------+-+
        | # ptrs in obj | # words in obj |1|
        +---------------+----------------+-+
        |          user data               |  <-- user data starts here. GCP
                     .                            ptrs come first
                     .
                     .
        |                                  |
        +----------------------------------+
```

```
   The number of words in the object count INCLUDES one word for the header
   and INCLUDES the words occupied by pointers.

   When an object is forwarded, the header will be replaced by the pointer to
   the new object which will have bit 0 equal to 0.
*/

#define MAKE_HEADER( words, ptrs ) ((ptrs)<<17 | (words)<<1 | 1)
#define FORWARDED( header ) (((header) & 1) == 0)
#define HEADER_PTRS( header ) ((header)>>17 & 0x7FFF)
#define HEADER_WORDS( header ) ((header)>>1 & 0xFFFF)
#define HEADER_BYTES( header ) (((header)>>1 & 0xFFFF)*WORDBYTES)

/* Garbage collector */

/* A page index is advanced by the following function */

int next_page( page )
        int  page;      /* Page number */
{
        if  (page == lastheappage)  return( firstheappage );
        return( page+1 );
}

/* A page is added to the page queue by the following function. */

queue( page )
        int  page;      /* Page number */
{
        if  (queue_head != 0)
           link[ queue_tail ] = page;
        else
           queue_head = page;
        link[ page ] = 0;
        queue_tail = page;
}

/* A pointer is moved by the following function. */

GCP move( cp )
        GCP  cp;        /* Pointer to an object */
{
        int  cnt,       /* Word count for moving object */
             header;    /* Object header */
        GCP  np,        /* Pointer to the new object */
             from, to;  /* Pointers for copying old object */

        /* If NULL, or points to next space, then ok */
        if  (cp == NULL  ||
             space[ GCP_to_PAGE( cp ) ] == next_space)
           return( cp );

        /* If cell is already forwarded, return forwarding pointer */
        header = cp[-1];
        if  (FORWARDED( header ))  return( (GCP)header );

        /* Forward cell, leave forwarding pointer in old header */
```

```
        np = gcalloc( HEADER_BYTES( header )-4, 0 );
        to = np-1;
        from = cp-1;
        cnt = HEADER_WORDS( header );
        while  (cnt--)  *to++ = *from++;
        cp[-1] = (int)np;
        return( np );
}

/* Pages which have might have references in the stack or the registers are
   promoted to the next space by the following function.  A list of
   promoted pages is formed through the link cells for each page.
*/

promote_page( page )
        int  page;      /* Page number */
{
        if  (page >= firstheappage  &&  page <= lastheappage  &&
             space[ page ] == current_space)  {
           while  (type[ page ] == CONTINUED)  {
              allocatedpages = allocatedpages+1;
              space[ page ] = next_space;
              page = page-1;
           }
           space[ page ] = next_space;
           allocatedpages = allocatedpages+1;
           queue( page );
        }
}

collect()
{
        unsigned  *fp;  /* Pointer for checking the stack */
        int  reg,       /* Register number */
             cnt;       /* Counter */
        GCP  cp,        /* Pointer to sweep across a page */
             pp;        /* Pointer to move constituent objects */

        /* Check for out of space during collection */
        if  (next_space != current_space)  {
           fprintf( stderr, "gcalloc - Out of space during collect\n" );
           exit( 1 );
        }
        /* Allocate current page on a direct call */
        if  (freewords != 0)  {
           *freep = MAKE_HEADER( freewords, 0 );
           freewords = 0;
        }
        /* Advance space */
        next_space = (current_space+1) & 077777;
        allocatedpages = 0;

        /* Examine stack and registers for possible pointers */
        queue_head = 0;
        for (fp = (unsigned*)(&fp) ;
             fp <= stackbase ;
             fp = (unsigned*)(((char*)fp)+STACKINC) )  {
```

```
            promote_page( GCP_to_PAGE( *fp ) );
        }
#ifdef FIRST_REGISTER
        for (reg = FIRST_REGISTER ; reg <= LAST_REGISTER ; reg++)  {
            promote_page( GCP_to_PAGE( register_value( reg ) ) );
        }
#endif

        /* Move global objects */
        cnt = globals;
        while  (cnt--)
            *globalp[ cnt ] = (int)move( *globalp[ cnt ] );

        /* Sweep across promoted pages and move their constituent items */
        while  (queue_head != 0)  {
            cp = PAGE_to_GCP( queue_head );
            while (GCP_to_PAGE( cp ) == queue_head  &&  cp != freep)  {
                cnt = HEADER_PTRS( *cp );
                pp = cp+1;
                while  (cnt--)  {
                    *pp = (int)move( *pp );
                    pp = pp+1;
                }
                cp = cp+HEADER_WORDS( *cp );
            }
            queue_head = link[ queue_head ];
        }

        /* Finished */
        current_space = next_space;
}

/* When gcalloc is unable to allocate storage, it calls this routine to
   allocate one or more pages.  If space is not available then the garbage
   collector will be called.
*/

allocatepage( pages )
        int  pages;             /* # of pages to allocate */
{
        int  free,              /* # contiguous free pages */
             firstpage,         /* Page # of first free page */
             allpages;          /* # of pages in the heap */

        if  (allocatedpages+pages >= heappages/2)  {
            collect();
            return;
        }
        free = 0;
        allpages = heappages;
        while  (allpages--)  {
            if  (space[ freepage ] != current_space  &&
                 space[ freepage ] != next_space)  {
                if  (free++ == 0)  firstpage = freepage;
                if  (free == pages)  {
                    freep = PAGE_to_GCP( firstpage );
                    if  (current_space != next_space)  queue( firstpage );
```

```
            freewords = pages*PAGEWORDS;
            allocatedpages = allocatedpages+pages;
            freepage = next_page( freepage );
            space[ firstpage ] = next_space;
            type[ firstpage ] = OBJECT;
            while (--pages)  {
               space[ ++firstpage ] = next_space;
               type[ firstpage ] = CONTINUED;
            }
            return;
         }
      }
      else  free = 0;
      freepage = next_page( freepage );
      if  (freepage == firstheappage)  free = 0;
   }
   fprintf( stderr,
            "gcalloc - Unable to allocate %d pages in a %d page heap\n",
            pages, heappages );
   exit( 1 );
}

/* The heap is allocated and the appropriate data structures are initialized
   by the following function.
*/

gcinit( heap_size, stack_base, global_ptr )
        int  heap_size;
        unsigned  *stack_base;
        GCP  global_ptr;
{
        char *heap;
        int  i;
        GCP  *gp;

        heappages = heap_size/PAGEBYTES;
        heap = malloc( heap_size+PAGEBYTES-1 );
        if ((unsigned)heap & (PAGEBYTES-1))
           heap = heap+(PAGEBYTES-((unsigned)heap & (PAGEBYTES-1)));
        firstheappage = GCP_to_PAGE( heap );
        lastheappage = firstheappage+heappages-1;
        space = ((int*)malloc( heappages*sizeof(int) ))-firstheappage;
        for (i = firstheappage ; i <= lastheappage ; i++)  space[ i ] = 0;
        link = ((int*)malloc( heappages*sizeof(int) ))-firstheappage;
        type = ((int*)malloc( heappages*sizeof(int) ))-firstheappage;
        globals = 0;
        gp = &global_ptr;
        while  (*gp++ != NULL)  globals = globals+1;
        if  (globals)  {
           globalp = (GCP*)malloc( globals*sizeof(GCP) );
           i = globals;
           gp = &global_ptr;
           while  (i--)  {
              globalp[i] = *gp;
              **gp = NULL;
              gp = gp+1;
           }
```

```
        }
        stackbase = stack_base;
        current_space = 1;
        next_space = 1;
        freepage = firstheappage;
        allocatedpages = 0;
        queue_head = 0;
}

/* Storage is allocated by the following function.  It will return a pointer
   to the object.  All pointer slots will be initialized to NULL.
*/

GCP  gcalloc( bytes, pointers )
        int  bytes,     /* # of bytes in the object */
             pointers;  /* # of pointers in the object */
{
        int  words,     /* # of words to allocate */
             i;         /* Loop index */
        GCP  object;    /* Pointer to the object */

        words = (bytes+WORDBYTES-1)/WORDBYTES+1;
        while  (words > freewords)  {
           if  (freewords != 0)  *freep = MAKE_HEADER( freewords, 0 );
           freewords = 0;
           allocatepage( (words+PAGEWORDS-1)/PAGEWORDS );
        }
        *freep = MAKE_HEADER( words, pointers );
        for  (i = 1; i <= pointers; i++)  freep[i] = NULL;
        object = freep+1;
        if  (words < PAGEWORDS)  {
           freewords = freewords-words;
           freep = freep+words;
        }
        else  {
           freewords = 0;
        }
        return( object );
}
```

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] D. G. Bobrow. Managing Reentrant Structures Using Reference Counts. *ACM Transactions on Programming Languages and Systems* 2(3):269-273, July, 1980.

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

[4] John DeTreville, Digitial Systems Research Center.
Private communication, February 12, 1988.

[5] L. Peter Deutsch, Daniel G. Bobrow. An Efficient, Incremental, Automatic Collector. *Communications of the ACM* 19(9):522-526, September, 1976.

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

[7] Brian W. Kernighan, Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall Inc., 1978.

[8] Brian W. Kernighan, Rob Pike. *The UNIX Programming Environment.* Prentice-Hall Inc., 1984, pages 227.

[9] Donald E. Knuth. *The Art of Computer Programming.* Volume 1.*Fundamental Algorithms.* Addison-Wesley, 1968, pages 406-422.

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

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

[12] 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.

[13] Guy L. Steele Jr. *Common LISP: The Language.* Digital Press, 1984.

[14] David W. Wall. Global Register Allocation at Link Time. *SIGPLAN Notices* 21(7):264-275, July, 1986. SIGPLAN'86 Symposium on Compiler Construction.

[15] 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.

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

Table of Contents

List of Figures