Real-Time Garbage Collection of Strings and Linked Data Structures*

Kelvin Nilsen

TR 87-5

ABSTRACT

Modern high-level languages such as Rosetta and Icon need to collect garbage not only from regions of linked structures similar to LISP's dotted pairs, but also from string regions where data is organized as an array of characters. Some characteristics of string regions that make garbage collection particularly difficult are: multiple pointers to the same characters within the array are allowed and encouraged, all possible character values are legitimate as data so it is not possible to ''mark'' a string by overwriting with a reserved character, and a character is generally much smaller than a pointer so it is not possible to overwrite a single character value with a forwarding pointer to a new location for a particular string.

This paper describes an algorithm for real-time garbage collection and its implementation as part of Rosetta's run-time support system. Source code for the algorithm is provided in C as an appendix to this document. Analysis of performance is also reported, and comparisons are made with traditional garbage collection and reference counting schemes.

January 30, 1987

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant DCR-8502015.

Real-Time Garbage Collection of Strings and Linked Data Structures

1. Icon's Storage Management System

Icon [1,2] is a high-level programming language conceived in 1976 for the purposes of string and list processing. This language provides a variety of scalar data types such as integer, float, and string; and several structured data types including user-defined records, associative tables, and dynamically sized lists (deques). Each structured data object is constructed of pointers to other objects, and pointer cycles are common. Rosetta[1] provides most of the same data structures as Icon, but has the run-time requirement of executing with real-time response. The Icon storage management system is outlined here to introduce some of the special problems of garbage collecting this assortment of data types.

All Icon-level data is accessed via descriptors. A descriptor consists of a type field holding an enumeration of all the possible data types, and a data field. The data field holds either the actual data or a pointer to the data. The integer type, for example, stores the integer value in the data field. Floats and other data that is too large to be stored in the descriptor's data field are allocated elsewhere, and pointers to these dynamically allocated objects are stored in the descriptor's data field. All dynamically allocated objects are titled in their first field with a code representing the type of the object. This title is used by the garbage collector.

String data is special. The high-order bit of the descriptor's type field is zero only for the string data type. For string data, the remaining bits of the type field are treated as an unsigned integer representing the length of the string. The data field of the string descriptor points at the first character of the string within a large character array. Note that Icon does not give any special significance to the null or any other character. Also important is that the creation of a substring requires no additional allocation within the character array. Since the desired characters are already in the array (pointed to by the parent string), the substring operation simply calculates the address of the first character in the substring and the substring's length.

Dynamic allocation of memory takes place in several regions. New strings are allocated from a large array of characters used only for string data. Blocks of memory for floats, records, and most other data types are allocated from an area of memory called the block region. The allocator assumes that both the string and block regions are initially empty, and proceeds from low to high addresses, satisfying each allocation request with a pointer to the next available memory within the region, and incrementing the internal pointer to the end of the allocated region by the size of the request. A heuristic employed by the string catenation routine has been measured to reduce the size of the string region by approximately 5%. Whenever its first argument extends to the end of the currently allocated string region, the catenate procedure allocates only enough memory to copy the second argument into the string memory that immediately follows the first argument's string data.

Icon's memory management must deal also with a static region. The special characteristic of memory allocated in this region is that these objects are never relocated. Objects that must be allocated in the static region include system input and output buffers and co-expression blocks (Icon's co-expressions are similar to coroutines). Within the static region, free memory is maintained on a linked list similar to that described in [4, pp. 173-177].

Preparatory to allocating memory, the Icon interpreter makes a predictive need request to the storage allocator. The predictive need request specifies the maximum amount of memory that might be required by the operations that

[1] Rosetta is a high-level programming language, similar to Icon, designed to facilitate both prototyping and final development of applications requiring real-time response. Rosetta is still under development and no reports have yet been published describing the language. Readers interested in Rosetta itself may refer to [3], the language reference document for CommSpeak, a direct ancestor of Rosetta. Wherever performance measurements of the real-time garbage collection algorithm are reported in this paper, these are measurements made on CommSpeak. CommSpeak's garbage collection algorithm differs from the algorithm described by this paper in that CommSpeak uses a heap of pointers to string descriptors to locate common string regions. Changes were made to the CommSpeak algorithm in order to bound allocation time by a small constant.

follow. If there is not enough free memory to satisfy the specified predictive need, garbage collection takes place. Icon garbage collection is traditional in the sense that all garbage is collected before program execution proceeds. The task of the garbage collector is to locate all memory that is accessible to the Icon program, possibly expand and relocate each of the regions in which allocation takes place, and relocate the accessible data as the first data allocated in the new regions. In practice, many Icon programs never collect garbage at all. In a large suite of Icon programs assembled for the purpose of monitoring Icon system performance, the cost of memory management is less than 3% of the entire cost of execution.

The first phase of garbage collection locates and marks all accessible objects and finds all pointers to each of these objects. For each accessible data block, a linked list of the descriptors pointing to that block is threaded through the data field of those descriptors. Because it is impossible to detect within the string region the areas of text that are accessed by multiple string descriptors before looking at all valid string descriptors, an array of pointers to these descriptors is constructed. This array is placed at the end of currently allocated memory (using the standard C library sbrk() system call) and can be expanded if the number of string descriptors exceeds its default size.

During the second phase of garbage collection, the array of pointers to string descriptors is first sorted by address within the string data region using the quicksort algorithm. Sorting makes it easy to discover regions of text that are shared by multiple string descriptors. As these accessible string pieces are allocated and copied to the new string region, the descriptors pointing into the string region are modified to point instead to the relocated address of the string. This method of string garbage collection was first used in the implementation of XLP [5].

Two passes are required through the block region to relocate the blocks that were marked by the first phase. The first pass calculates the new address of each accessible block, and adjusts all the pointers to that block accordingly. The second pass relocates the blocks. This work must be done in two passes because the addresses used to link the list of descriptors pointing to each block are invalidated by relocation of the data blocks.

2. Garbage Collection in Real Time

In [6], Baker describes a method of performing garbage collection that distributes the work of garbage collection over several invocations of the storage allocator. In this section, his basic algorithm is outlined and extensions to the algorithm are proposed which permit the garbage collection of objects of different sizes, process stacks, and strings.

The Baker algorithm uses two regions for allocation, a *to* space and a *from* space. While allocation is taking place in the *to* space, garbage collection is performed in the *from* space. Non-garbage in the *from* space is incrementally relocated to the *to* space. Calculations performed at run time guarantee that garbage collection of the *from* space completes before the *to* space exhausts its free pool. When there is no longer enough memory available in *to* space to satisfy an allocation request, the names given to *to* space and *from* space are reversed. Baker provides algebraic justification of his claim that the cost of each allocation is bounded by a small constant.

Baker's work was somewhat simplified by the fact that his target language was LISP, where each data object is the same: a two-element cell containing a CAR and a CDR. Both the algorithm and the analysis are cluttered by additional detail when storage management must deal with a variety of data types and sizes. Of even greater impact on the original Baker algorithm is the addition of the string data type to the repertoire that the storage management system must effectively handle. A description of the Baker algorithm modified to deal with Rosetta's built-in data structures follows. An implementation of this algorithm written in C accompanies the description.

Since Rosetta, unlike Icon, is strongly typed at compile time, there is no need to maintain type information in the standard variable descriptor. Another difference between Rosetta and Icon is that all Rosetta objects are heap allocated, so all Rosetta descriptors are treated as pointers. A title for use by the garbage collector is still the first word of each dynamically allocated block. There are several Rosetta data types with sizes that cannot be determined by their type alone. For these types, information regarding the actual size of the object is stored within the object. For example, since a programmer can specify many flavors of records, the first word following the title of each record block specifies the number of fields within that record.

Conceptually, Rosetta's memory allocation is very similar to Icon's. Memory is allocated starting at the beginning of *to* space. Each allocation request is satisfied by returning the value of an internal pointer representing the next available memory in *to* space and subsequently incrementing that pointer by the size of the allocation. Because the management of multiple regions of allocation introduces the complexity of synchronizing garbage collection phases between each of the regions, both string data and data blocks are allocated from the same pool.

Although similar in concept, allocation is slightly more complicated in Rosetta than in Icon. First, in order not to expand storage regions unless the behavior of a particular Rosetta program requires additional storage, it is necessary[2] to verify at each allocation that the currently allocated data need no more space than the current size of the allocation region to collect garbage. Additionally, if garbage collection of *from* space is still active, then some amount of work proportional to the size of the current allocation request must be contributed to garbage collecting that region.

Rosetta's garbage collector need not execute incrementally. It could just as well run to completion on a single invocation. The key features of this algorithm that make it ideal for real-time execution are that the algorithm is iterative (as opposed to recursive), easily interrupted between iterations, and the forward progress of the work done by the algorithm is directly evident and measurable throughout its execution. This last characteristic is necessary to equitably distribute its execution over many invocations. In order to satisfy real-time constraints, the garbage collector is invoked many times, each invocation specifying the percentage of the old allocated region to be garbage collected by the invocation. The percentage collected by each invocation is proportional to the size of the allocation request. The proportionality constant, which controls an inverse relationship between response time and the amount of memory required for program execution, may be set at compile time or adjusted at run time.

The description of Rosetta's garbage collection that follows ignores for the moment string and stack data types. The strategy of the garbage collector is to relocate every accessible object from the old allocation region into the new allocation region. To bootstrap the algorithm, all objects pointed to by the system's tended descriptors are first relocated. By definition, tended descriptors are the only pointers from the interpreter into Rosetta's data space. Data that can not be reached by following some path of pointers originating at a tended descriptor is garbage, and its space can be reclaimed to satisfy future allocation requests. Because multiple pointers to the same object are common, every time an object is relocated, the old object is overwritten with a special mark and a forwarding address to its new location. The algorithm proceeds by repeatedly relocating objects pointed to by objects that were just relocated themselves. This process, called scanning, results in a breadth-first mark with concurrent compression of accessible memory. After all pointers of all relocated objects have been followed and relocated, garbage collection is done. The old storage region may be reclaimed to serve as a future allocation region.

The amount of garbage collection performed on each allocation request is governed by the constant K. For an allocation request of n bytes, the garbage collector copies or scans a combined total of $n \times K$ bytes. Copying is the process of moving an object from *from* space into *to* space. Scanning consists of updating all pointers contained within the objects just relocated into *to* space.

Suppose that when garbage collection begins, the old *to* space contains N bytes of allocated memory. If that region contains no garbage, then all N bytes need to be copied into the new *to* space and all N bytes need to be scanned. The process of copying this data from *from* space into *to* space and scanning the relocated objects requires that the program allocate $2 \times N / K$ bytes of new memory. So in the worst case, *to* space must be at least $N + 2 \times N / K$ bytes large.

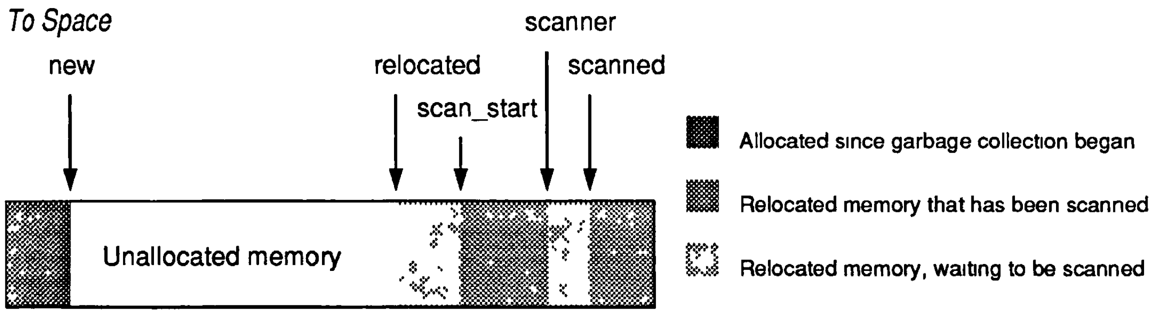
Throughout the remainder of this document, the term "word" refers to the data type representing a machine address for a given architecture and compiler. Using this definition, the size of a word is not necessarily identical to the size of memory or registers. Because Rosetta data objects range in size from two words to ten (or even larger for a record data type with many fields), it is not always convenient to garbage collect exactly the requested amount on each invocation. The garbage collector can be suspended only between the relocation of entire objects. Rosetta retains within the garbage collector's state an integer count representing the balance between garbage collection and allocation. Following a call to the garbage collector that results in excessive garbage collection, the next call to the garbage collector yields less work than normal from the collector because this count shows that collection is ahead of allocation.

During the time that garbage collection is taking place, many data structures are in a kind of limbo. For example, part of a linked list may reside in *to* space and part in *from* space. Rosetta's interpreter accesses these data structures only by means of its tended descriptors. At the moment garbage collection begins, the objects referenced

[2] As described by Baker, this check is not made with each allocation, but instead at the start of each garbage collection pass. Unfortunately, that approach requires that storage regions expand with each pass of the garbage collector, and even the virtual memory of a VAX computer was quickly exhausted by some simple test programs run at the University of Arizona.

by all tended descriptors are relocated to the new allocation region. The interpreter maintains the invariant that only data objects in the current allocation region are directly accessible by preceding each assignment to a tended descriptor with relocation of the object whose pointer value is to be assigned to the descriptor if the object resides in *from* space. This bookkeeping work is one of the major costs of distributing garbage collection over several invocations of the garbage collector. It results in increased code size and slower execution. Since in a typical program, garbage collection is only active a small percentage of its running time, the operation that relocates objects before assignment to tended descriptors is packaged in a C pre-processor macro that checks whether garbage collection is active before attempting to relocate the data object and update the pointer.

The figure below illustrates the layout of the memory region used for new memory allocation.


To Space
new
relocated
scan_start
scanner
scanned
Unallocated memory
Allocated since garbage collection began
Relocated memory that has been scanned
Relocated memory, waiting to be scanned

Following is an outline of the high points of this algorithm written in C. This version of the algorithm does not implement garbage collection of string data.

```
/* Baker algorithm for storage management adapted for variable-sized allocations */

/* constants for title of each object */
#define INTEGER 0
#define RECORD 1
#define MARKED 2

typedef long word,
typedef unsigned long uword,

struct O_int
{
  word title,
  long i,
},

struct O_rec
{
  word title,
  int num_fields,
  union Object *fields[1],
};

struct O_forward
{
  word title,
  union Object *new_address,
},

typedef union Object
{
  struct O_int int_obj,
  struct O_rec rec_obj,
  struct O_forward template,
} *Descriptor,
static int numDesc[] =                /* how many descriptors in an object of this type? */
{
  0,              /* integer */
  -1,             /* record, -1 means determined by 2nd word within object */
}

static int descOffset[] =       /* at what offset does the first descriptor appear? */
{
  sizeof(struct D_int),                       /* integer */
  sizeof(struct D_rec) - sizeof(Descriptor),  /* record */
};

word nxtBlockSize = 0, blockSize = StartSize;

char *new, *relocated, *scanner,       /* scanner == NULL means garbage collection is done */
     *scanned, *scan_start;
char *toSpace, *toBack, *fromSpace, *fromBack;

int scan_balance = 0;

int scan_cnt;                   /* number of descriptors to scan out of current scan object */

Descriptor tendDesc[TendedDescriptors];

/* allocate n bytes */
char *alloc(n)
     unsigned int n;
{
  register char *cp;

  nxtBlockSize += n + (n + K - 1) / (K/2);
  if (scanner)
    scan(n);

  if (new + n > relocated || (scanner == NULL && nxtBlockSize >= blockSize))
  {                                           /* start garbage collection */
    uword used_size;

    /* recalculate nxtBlockSize to eliminate accumulation of roundoff errors */
    used_size = blockSize - (relocated - new);
    nxtBlockSize = used_size + 2 * used_size / K + n;
    if (blockSize < nxtBlockSize)             /* expand region if necessary */
    {                                         /* but don't shrink it */
      blockSize = nxtBlockSize;
      cp = fromSpace;
      fromSpace = realloc(toSpace, blockSize);
      toSpace = realloc(cp, blockSize);
      fromBack = fromSpace + blockSize;
      toBack = toSpace + blockSize;
    }
    else
    {
      cp = fromSpace;
      fromSpace = toSpace;
      toSpace = cp;
      cp = fromBack;
      fromBack = toBack;
      toBack = cp;
    }

    nxtBlockSize = 0;
    scan_cnt = 0;
    relocated = scanner = scan_start = scanned = toBack;
    new = toSpace;
    scan_balance = 0;
    for (i = 0; i < TendedDescriptors; i++)
      tendDesc[i] = copy(tendDesc[i]);
  }
  cp = new;
  new += n;
  return cp;
}

/* garbage collect an amount corresponding to an allocation of size amt */
scan(amt)
     int amt;
{
  int i;

  scan_balance += amt * K;
  while (scan_balance > 0)
  {
    if (scan_cnt)
    {
      scan_cnt—;
      *((Descriptor *) scanner) = copy(*((Descriptor *) scanner));
      scanner += sizeof(Descriptor);
      scan_balance -= sizeof(Descriptor);
    }
    else
    {
      int type;

      if (scanner == scanned)
      {
        if (scan_start == relocated)
        {                       /* garbage collection is done */
          scanner = NULL;
          return;
        }
        else
        {
          scanned = scan_start;
          scan_start = scanner = relocated;
        }
      }

      type = ((Descriptor) scanner)->template.title;
      scan_balance -= descOffset[type];
      scanner += descOffset[type];
      scan_cnt = (numDesc[type] < 0)?
        ((Descriptor) scanner)->rec_obj.num_fields: numDesc[type];
    }
  }
}

/* copy an object into to-space if necessary, returning a pointer to the relocated object */
Descriptor copy(data)
     Descriptor data;
{
  if (data == NULL || (char *) data < fromSpace || (char *) data > fromBack)
    return data;
  else if (data->template.title == MARKED)
    return data->template.new_address;
  else
  {
    register int size;
```

```
            size = descOffset[data->template.title];
            size += sizeof(Descriptor) * ((descOffset[data->template.title] < 0)?
              data->rec_obj.num_fields: descOffset[data->template.title]);

            scan_balance -= size;
            relocated -= size;
            nxtBlockSize += size + (size + K - 1) / (K/2);
            movmem((char *) data, relocated, size);

            data->template.title = MARKED;
            data->template.new_address = (Descriptor) relocated;
            return (Descriptor) relocated;
        }
    }

    /* this macro is used to fix pointers before indirection within the interpreter */
    #define FixPtr(p)         if (scanner) p = copy(p)
```

2.1 Process Stacks

Rosetta supports multi-tasking, each task owning a private stack. Run-time stacks present special problems to real-time garbage collection. Because stacks are theoretically infinite in size, it is not possible to follow all pointers contained on the stack in a finite amount of time, let alone a time bounded by some small finite constant. This difficulty is handled by noting that at the time garbage collection begins, the stack is of some finite size. Any stack growth after garbage collection begins contains only pointers into the new allocation region. The fixed-size stack that exists at the moment garbage collection begins is easily scheduled for incremental garbage collection scanning to be executed with each allocation request.

Because stacks are typically much larger than all other data types, the problems of granularity become especially noticeable here. The time spent copying the 4000 bytes or so of a stack could easily violate real-time response requirements.

It would be possible to simulate *to* and *from* spaces for stacks by placing all stacks on two doubly-linked lists, a *to* list and a *from* list. The work of copying a stack from one space to another would thereby be reduced to the work of unlinking the stack from the *from* list and inserting it on the *to* list, easily bounded by a small constant.

However, the design of Rosetta is such that stacks need not be reclaimed by standard garbage collection techniques. Stacks become inaccessible when the task served by the stack is killed either by another task or by itself. Thus all stacks are either in use, or on a free list. The total number of stacks available to a program is fixed at link time and is under the programmer's control.

Since the interpreter continues to run while garbage collection is taking place, care must be taken to synchronize the garbage collector and the interpreter with respect to stack sizes. Scanning of a stack begins with the most deeply nested function frame. A state variable accompanying each stack points to the next descriptor on the stack to be scanned as scanning progresses toward the stack's bottom. Each pop that shortens a stack beyond its scanned region adjusts this pointer. Any descriptors pushed onto the stack need not be scanned because their values, having passed through tended descriptors, represent data in the current *to* space.

When there are no more descriptors to relocate, either in the data block region, or the list of active stacks, garbage collection is done.

2.2 Strings

In the algorithm described so far, each allocation of size n is accompanied by the copying and scanning of $n \times K$ bytes of previously allocated memory. String data must also be copied to the new region, but there is no need to scan it because string data contains no pointers. Each byte of string data is considered both relocated and scanned at the moment it is copied into *to* space. It is convenient to count string data twice toward the balance between garbage collection and allocation because the copying of string data is slightly more complicated than the copying of linked data structures.

As string data is allocated, it is grouped into common regions. Each region is prefixed with an integer that serves as a traditional mark during the garbage collection phase. A negative value signifies that this particular block has not yet been marked. A positive value represents the index of a string control block within a special array and signifies that the string region has been marked. Every string descriptor contains a pointer both to its string data and to this unsigned integer that precedes the string segment containing the string data.

During garbage collection, each time a string descriptor is relocated into *to* space, the region of string data pointed to by this string descriptor is readied for relocation. If the region has not yet been marked, a region control block is allocated for this string data and the mark field of the region is set to the appropriate index value. Within the region control block is kept a pointer to a list of string descriptors that refer to this region. The newly relocated string descriptor is added to this list.

As string descriptors are installed on the list of descriptors to be adjusted for a given region, a simple search is performed for holes of unreferenced data within the string region. In order that this search be performed in constant time, each garbage collection pass searches for only two possible holes and finds these holes only if they surround predetermined indices within the string region. The indices, which serve as probes for holes, are changed for each string region on each pass of the garbage collector. The sequence of values for these probes covers every possible index within the string exactly once and does so in such a way as to locate large holes in fewer passes of the garbage collector than are required for small holes. The arithmetic required to generate this sequence is somewhat involved, but it is conceptually simple and would benefit greatly from being placed in microcode or hardware. Refer to the source code appendix for further details on how this is done.

When holes are found between referenced string data, the original string region is split into multiple string regions representing the valid string data between the holes. The region is divided on the assumption that having found one hole in the region, other holes are likely to exist. These holes can be found most rapidly by dividing the region. In cases where this assumption is not valid, subsequent passes of the garbage collector join this region with others as described in the following paragraph.

After all probes within a string region have failed to discover any internal holes, the string region is merged with other string regions with the same status under the assumption that these regions will remain relatively stable. The region created out of the merger is then probed for newly formed holes on subsequent passes of the garbage collector.

This algorithm is not particularly quick at eliminating holes within existing string regions. However, the algorithm executes in constant time and guarantees that all holes will eventually be found. In particular, a hole representing $1/\mu$ the size of a complete string region is found within $2^{\lceil \log_2 \mu \rceil}$ passes of the garbage collector. In most cases, this tradeoff is acceptable as the price of computer memory is low and continues to decrease.

It is possible for a string created since the current pass of the garbage collector began to point at string data in *from* space (if the new string is created as a substring of an older one). When this occurs[3], the new string descriptor is also linked onto the list for the given string region. As it is inserted onto the list, an amount of garbage collection equivalent to what will be required to eventually relocate this string descriptor is performed.

In Rosetta, it is possible to modify an existing string descriptor so that the descriptor points to different string data than previously. Whenever a descriptor already on a list of string descriptors scheduled for adjustment is modified in this manner, that descriptor is removed from the list of descriptors associated with its old string data region. To facilitate constant-time removal of this descriptor from the list, these lists are doubly linked.

Garbage collection of strings requires the use of a large region of contiguous memory to hold an array of string region control blocks. The size of this array depends on the number of independent string regions that must be relocated. In the worst case, this includes every string region in existence at the time garbage collection begins. A count is maintained of the number of string regions inserted in *to* space. This count is used to bound the size of this array for the subsequent garbage collection pass. The array is located in the middle of *to* space, *N* bytes[4] from the end of the array so that the array may be overwritten by allocation requests that arrive after garbage collection completes.

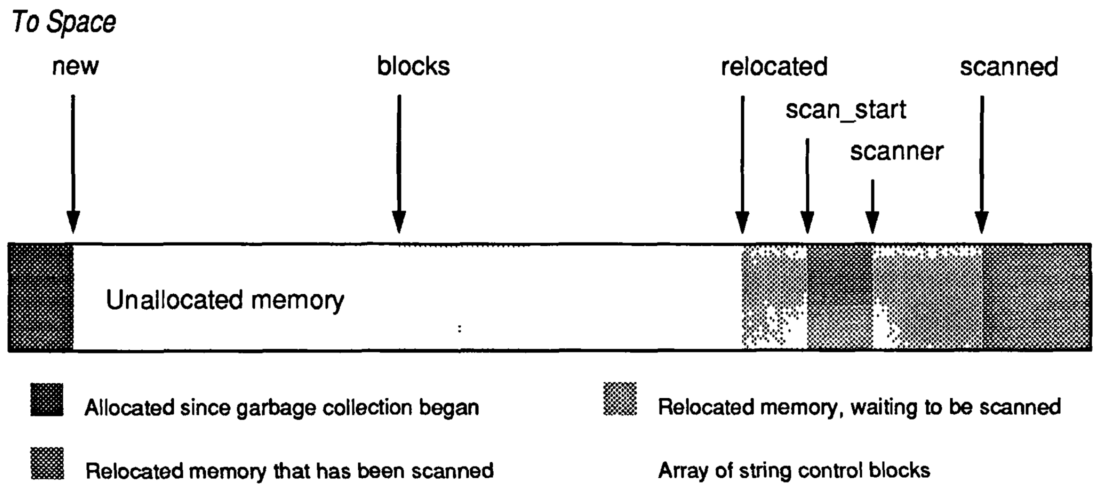
Rosetta uses the same heuristic as Icon to avoid extra copies of string data whenever the first argument to string catenation is the most recent data allocated in *from* space. The actual savings in string data space has not been measured but is probably not as great as the savings earned by Icon because all data, not just strings, is allocated

[3] Range checks on pointer values determine whether objects and string data are in *to* or *from* space.

[4] To be exact, this array is located far enough from the end of *to* space to allow for the copying of all *N* bytes of *from* space and the 3-way splitting of each existing string region.

from the same free pool.

The figure below illustrates the various regions in *to* space for the garbage collection of linked data structures and strings.


To Space
new
blocks
relocated
scan_start
scanner
scanned
Unallocated memory
Allocated since garbage collection began
Relocated memory, waiting to be scanned
Relocated memory that has been scanned
Array of string control blocks

For more information on the details of this algorithm, refer to the source code supplied as an appendix to this document.

3. Analysis of Real-Time Garbage Collection

Performance of this garbage collection algorithm is measured in terms of worst-case response and average throughput. Of greatest interest to real-time programmers is the worst-case response time.

3.1 Worst-case Response Time

In the equations below, K represents the proportionality constant discussed earlier that relates allocation to garbage collection. Each allocation of n bytes is accompanied by scanning and relocating a combined total of $n \times K$ bytes.

Let γ be the size of the largest object referenced by the program. The largest built-in data type contains 10 words. A record with a large number of fields may be larger. Lists are not implemented as single contiguous blocks of memory so they need not be considered in determining the value of γ.

Define Γ to be the cost of copying a byte of data from one area of memory to another. Assume that this cost is the same as the cost of scanning a byte. Remember that scanning consists of looking at a pointer and modifying its value if the pointer points into *from* space. Scanning frequently requires that objects be copied into *to* space but that operation and its costs are charged separately.

An allocation of δ bytes of memory executes in time

$$\alpha + \beta + (\delta \times K + \gamma - 1) \times \Gamma$$

where α is the constant time required to perform the allocation and the rest of the expression above represents the cost of garbage collection. β in the expression above represents the constant costs of garbage collection including the cost of bootstrapping the garbage collection algorithm. The bootstrapping cost includes the cost of updating all tended descriptors, which is simply the number of tended descriptors times $(\gamma \times \Gamma)$. In most real programs, garbage collection is only active a small fraction of the running time, so the actual cost of allocation is usually only α.

Storage Throughput and Requirements

Since each invocation of the garbage collector executes in time proportional to the amount of garbage collection performed, garbage collecting a region of size N executes with time complexity:

$$O(N)$$

An analysis of storage requirements is slightly more involved. The applications that motivated the design of Rosetta typically run for long periods of time. Existing programs serving these application areas generally iterate, responding to asynchronous events as they occur. Most programs reach a steady state, finding a natural balance between allocation and garbage collection. In this steady state, the amount of memory accessed by the program at any time (the non-garbage) remains relatively constant.

Suppose that the steady-state amount of memory referenced by a program is M bytes. Then the combined total of *to* space and *from* space must be at least

$$2 \times (M + \frac{2 \times M}{K}) = 2 \times M \times (1 + \frac{2}{K})$$

K in the above equation is the same constant discussed earlier. Note that memory requirements are $O(\frac{1}{K})$ and worst-case response time is $O(K)$. For large K, response time increases but memory requirements decrease. Decreasing K improves response time at the expense of additional memory.

In the analysis described above, M includes the space occupied by string data regions, which may include holes of unaccessible data that has not yet been reclaimed by the garbage collector. For typical programs, this represents a very small fraction of available memory.

4. Comparison with Icon Algorithm

Since Icon performs all garbage collection with a single invocation of the garbage collector, Icon's performance can be compared only with the analysis of storage throughput described above. Icon's cost of garbage collecting multiple regions with a combined total of N bytes and a total of ν string descriptors is of time complexity

$$O(N + \nu^2)$$

In this expression, the first term represents the cost of relocating objects in order to expand regions and compress holes from in between allocated objects. The second term represents the cost of sorting string descriptors using the quicksort algorithm in order to recognize overlapping string segments.

Although the time complexity of the real-time algorithm compares favorably with Icon's algorithm, the real-time algorithm imposes a greater penalty on standard memory referencing operations. For example, all indirection of tended descriptors must be prefaced by a test to see if garbage collection is active, and if so whether the referenced object has been relocated.

One advantage of the real-time algorithm is that, because it is designed to run concurrently with execution of a program, most of the work of garbage collection could be delegated to an auxiliary processor, and much of the bookkeeping work associated with indirection of pointers could be executed by microcode or even hard wired into a CPU. This assumes, of course, that dedicated hardware might be developed to support this garbage collection algorithm.

In real comparisons with Icon programs, this algorithm performs comparable to Icon's garbage collector. In programs for which the task of garbage collection is small in comparison to overall execution time, programs using the real-time garbage collector executed in approximately half the time of their Icon analogs. This comparison represents differences in the languages and the speed of their respective interpreters. Programs that perform large amounts of difficult garbage collection run in about the same time, or up to 50% slower than Icon. In making these comparisons, it is important to recognize that the work performed by the respective garbage collectors is not exactly the same. Rosetta heap allocates all objects, whereas Icon places small objects such as integers and strings (the descriptor, not the data) on the stack. Because of this, Rosetta's garbage collector executes several times more frequently than Icon's.

5. Comparison with Reference Counting

Reference counting is often considered the preferred storage reclamation method whenever both real-time response and implicitly managed storage allocation and reclamation are required. For a variety of reasons discussed below, this is not in general true.

Reference counting schemes do not work well for the reclamation of string data. To allocate a reference count to each byte of string data at least doubles[5] the amount of required string data space, and significantly increases the amount of work associated with string operations. Allowing a single reference count to serve a region of several bytes of data reduces the time costs mentioned above, but permits fragmentation of memory, which could in the worst case require even more string data space than allocating a reference count to each byte.

Linked data structures that permit pointer cycles cannot be reclaimed by reference counting alone. The task of breaking pointer cycles can be pushed off on the programmer, or can be performed implicitly by some sort of traditional garbage collection algorithm. Forcing programmers to break all pointer cycles they might create does not satisfy Rosetta's goal of providing a high-level programmer interface for real-time applications. Detecting and breaking cycles by adding to the reference counting scheme standard garbage collection capabilities significantly increases the amount of work that must be done by the reference counting mechanisms[7] which, even without this additional complexity, has difficulty competing with the real-time garbage collection algorithm described in this document.

Reference counting does nothing to solve the problem of fragmentation of memory in the presence of differently sized objects. A common approach to this problem is to keep several free lists, one list of objects for each size that may be required. Since the number of lists must be fixed in order to manage these lists efficiently, and because the sizes of Rosetta's objects range from 2 words theoretically to infinity, it is not possible to exactly match each allocation request to an available block using this system of storage management. The approach that would probably be taken is to keep free lists of sizes increasing as the power of 2, e.g. 4, 8, 16, 32, 64, ... Lists would be expanded according to the needs of the program and conversion of objects from one size to another would not be performed. Allocations return objects of the smallest size that is at least as big as the allocation request. Using this strategy, each allocation is in the worst case little more than half used. Furthermore, the free lists contain enough objects of each size to satisfy the needs of the program in its worst moment so far with respect to objects of that size. Assuming that the reference counting scheme is organized as described in this paragraph and counting the reference count field of each object as part of the storage management overhead, reference counting is unable to guarantee better than 50% utilization of available memory.

Another problem with reference counting is that care must be taken when freeing an object to not violate real-time response requirements. If, when the reference count for an object goes to zero, the reference counts for all objects referenced by that object are also decremented and so on, then there is no reasonable bound on the amount of time required to decrement a single reference count. Instead, when a reference count becomes zero, the object must be moved immediately to the free list for objects of that size. When the object is removed from the free list to satisfy an allocation request, all objects pointed to by that object will have their reference counts decremented. Since the free list is most likely threaded through the reference count fields, this requires a type tag on each object in addition to the reference count. This type tag is needed to know how many pointers the object contains and where they are located.

An unfortunate consequence of this arrangement of free lists is that not all free objects of a given size are linked on the list from which they will eventually be allocated. Suppose, for example, that an object of size 10 contains pointers to objects of size 4, and suppose that the only pointers to the 4-byte objects are from a given 10-byte object. When the reference count on the 10-byte object goes to zero, this object will be placed on its free list, and eventually so should each of the objects pointed to by this 10-byte object. But the 4-byte objects are not moved to their free list until the 10-byte object is allocated for re-use. If, before the 10-byte object is allocated, an allocation request arrives for a 4-byte object and the respective free list is empty, the pool of 4-byte objects must be increased, resulting in even poorer space utilization for the reference counting algorithm.

Each allocation of Δ words is accompanied in the worst case by indirecting up to $2^{\lceil \log_2 \Delta \rceil} - 2$ pointers, decrementing the reference counts for each of these objects, and relinking each of these objects onto the corresponding free

[5] This assumes that a reference count occupies at least one byte.

list. The work associated with each allocation corresponds roughly to the work performed with each allocation by the real-time garbage collection algorithm described in this paper. Using the same symbols defined in the worst-case analysis of real-time garbage collection but allowing Γ to represent the cost of indirecting a byte, or of decrementing the reference count of the object referenced by the indirected byte[6], or of linking or unlinking an object onto a free list, the cost of an allocation can be expressed as follows:

$$(2 + 3 \times \max(2^{\lceil \log_2(\Delta) \rceil} - 2, 0)) \times \mathit{sizeof}(\mathit{word}) \times \Gamma$$

The expression above accounts in its first term for the cost of removing the allocated list from the free list and setting its reference count to one. The second term is the cost of decrementing reference counts for each of the objects previously referenced by the newly allocated object. The expression above can be rewritten to represent the cost of allocating δ bytes of memory:

$$(2 + 3 \times \max(2^{\left\lceil \log_2(\frac{\delta}{\mathit{sizeof}(\mathit{word})}) \right\rceil} - 2, 0)) \times \mathit{sizeof}(\mathit{word}) \times \Gamma$$

The cost of allocating memory using the real-time algorithm is simply the cost of making the allocation plus the cost of collecting garbage if garbage collection is active. The allocation itself incurs only the cost of some simple pointer arithmetic to calculate the address of the new object and to adjust the running total for the size of the next *to* space. In typical programs, garbage collection is only active 5 to 25% of the time the program is executing. When garbage collection is active, allocation cost is given by:

$$\alpha + \beta + (\delta \times K + \gamma - 1) \times \Gamma$$

Note that this expression depends strongly on K, which can be adjusted within reason to deliver performance satisfying a wide range of needs. It should not be difficult to set K such that even the worst-case response time for the real-time garbage collection algorithm is better than that of the reference counting system.

One other advantage of real-time garbage collection over reference counting is that the removing of a reference to an object, and the addition of a new pointer to an existing object require no storage management work, whereas both operations require updating of reference counts using the reference counting method of allocation, and decrementing a reference count may incur the additional expense of linking the object onto its appropriate free list. This savings is realized during the 75% or so of time that a typical Rosetta program is not collecting garbage.

6. Conclusions and Future Work

By trading memory for response time, it is possible to perform full-fledged garbage collection of linked data structures and string data without sacrificing real-time response. Further, the tradeoffs between increased memory sizes and quicker response time can be adjusted over a wide range of values by setting the constant K to its desired value.

Further investigation of the behavior of string regions both analytically and through actual performance measurements is needed. A bound of some sort must be placed on the amount of memory required by a program that might create a large number of small holes in string data regions.

A problem that requires additional attention is that the work of allocation and garbage collection is not easily distributed amongst multiple processors. This is because there are many places in the code where mutual exclusion is required, so many that execution of the code is essentially serialized. A possible solution requiring additional study is to create one *to* and one *from* space for each processor that contributes to the job of storage management.

7. Acknowledgements

This research was initiated as a result of Ralph Griswold's expressed dissatisfaction with reference counting methods of storage management. His comments motivated a search for a better way of doing things. Both Ralph Griswold and Janalee O'Bagy have contributed to this work by reading early drafts of the document and suggesting

[6] Since indirection is performed only on word quantities, consider this to be cost of indirecting a word divided by the size of a word in bytes.

a variety of improvements.

References

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
2. R. E. Griswold and M. T. Griswold, *The Implementation of The Icon Programming Language*, Princeton University Press, 1986.
3. K. Nilsen, ''The CommSpeak Language Reference manual'', The Univ. of Arizona Tech. Rep. 87-4, Tucson, Arizona, 1987.
4. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978.
5. W. M. McKeeman, J. J. Horning and D. B. Wortman, *''A compiler generator''*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1970.
6. H. G. Baker Jr., ''List Processing in Real Time on a Serial Computer'', *Comm. ACM 21*, 4 280-293.
7. T. W. Christopher, ''Reference Count Garbage Collection'', *Software—Practice & Experience 14*(1984), 503-507.

Appendix: Implementation of Real-Time Garbage Collection in C

```
/* constants for title of each object */
#define INTEGER 0
#define STRING 1
#define RECORD 2
#define STACK 3
#define MARKED 4

typedef long word;
typedef unsigned long uword;

struct O_int
{
  word title;
  long i;
};

struct O_rec
{
  word title;
  int num_fields;
  union Object *fields[1];
};

struct O_string
{
  word title;
  unsigned int *block;
  char *s;
  unsigned int len;
};

/* string descriptors are linked onto a list associated with each
     string region for updating pointers into that string region.
     the linked list is created by overlaying (struct O_string)
     with (struct t_string)
*/
struct t_string
{
  struct t_string *next, *prev;
  char *s;
  unsigned int len;
};

struct O_forward
{
  word title;
  union Object *new_address;
};

struct O_stack
{
  word title;
  struct O_stack *prev, *next;
  Descriptor *sp, *scan_sp, stack[StackSize];
};

struct stack_head
{
  word title;
  struct O_stack *prev, *next;
}
```

```
typedef union Object
{
  struct O_int int_obj;
  struct O_string str_obj;
  struct O_rec rec_obj;
  struct O_stack stack_obj;
  struct O_forward template;
} *Descriptor;

struct str_block                    /* control block for each string region */
{
  char *start, *stop;                       /* outer bounds on this region */
  char *probes[2];                          /* current probe points */
  int probe;                                /* running probe count */
  union
  {
    char *hole_starts[2];
    struct
    {
      struct str_block *next;       /* next block for merging */
      word offset;                  /* by how much do we adjust string pointers */
    } s;
  } u;
  char *hole_stops[2];                      /* outer bounds on each hole */
  struct t_string fix_list;
};

static int objSize[] =
{
  sizeof(struct O_int),             /* int */
  sizeof(struct O_string),          /* string */
  sizeof(struct O_rec),             /* record */
  sizeof(struct O_stack),           /* stack */
};

static int descOffset[] =           /* offset of first descriptor if there is one, or total size */
{
  sizeof(struct O_int),
  sizeof(struct O_string),
  sizeof(struct O_record) - sizeof(Descriptor),
  5 * sizeof(word),
};

static int numDesc[] =
{
  0,                                /* int */
  0,                                /* string */
  -1,                               /* record */
  0,                                /* stack */
};

uword str_total;            /* total length of merged string data */

int scan_block;             /* string block number currently being scanned */

Descriptor tendDesc[TendedDescriptors];     /* array of tended descriptors */
Descriptor copy();

extern word datasize;               /* number of static variables */
extern Descriptor *dseg;            /* start of static data */

char *fromSpace, *fromBack, *toSpace, *toBack;
char *new, *scanner, *scan_start, *scanned, *relocated;

int numStrings;                     /* number of string regions */
static int numStacks;               /* number of stacks in use */
uword blockSize, nxtBlockSize;      /* size of to-space, next to-space */
```

```
static int scan_cnt;                /* number of descriptors to scan */
struct str_block *blocks;           /* base of string block array */
unsigned int block_no;              /* number of next string block to be allocated */

struct O_stack *stackScan;
struct stack_head freeStack,        /* list of free stacks */
     stacksInUse;                   /* list of stacks being used */

word scan_balance = 0;

#define N  (blockSize - (relocated - new))     /* how much memory might need to be relocated */
#define ScanSize N                             /* how much memory might need to be scanned */

/* NewSize is size of new region after done with garbage collection */
#define NewSize  ((N + ScanSize + (datasize + numStacks * StackSize) * sizeof(Descriptor))/K)

/* StringGSize - storage needed to collect strings */
/*  note that each string region might be split three ways */
#define StringGSize  (numStrings * (sizeof(struct str_block) + 2 * sizeof(int)))

/* string regions that have probed all indices for holes unsuccessfully are linked for merging */
struct str_block *mergers;

int *lastString;

/* allocate n bytes */
char *alloc(n)
     register unsigned n;
{
  register char *cp;

  if (lastString)
  {
    *lastString = init_probe(new - (char *) (lastString + 1));
    lastString = NULL;
  }

  if (!scanner || scan(n))
  {
    nxtBlockSize += n + (n + K - 1)/(K/2);
    if (nxtBlockSize >= blockSize)
      help_alloc(n);
  }
  cp = new;
  new += n;
  return cp;
}

/* allocate string data, merging with existing string region if possible */
char *s_alloc(sdesc, n)
     Descriptor sdesc;
     register unsigned n;
{
  register char *cp;
```

```
    if ((sdesc->str_obj.len = n) == 0)
    {
      sdesc->str_obj.s = NULL;
      sdesc->str_obj.block = NULL;
    }
    else
    {
      n += sizeof(int);                   /* leave room for string block prefix */
      if (!scanner || scan(n))
      {
        nxtBlockSize += n + (n + K - 1)/(K/2);
        if (nxtBlockSize >= blockSize)
          help_alloc(n);
      }
      cp = new;

      if ((sdesc->str_obj.block = lastString) == NULL)
      {
        sdesc->str_obj.block = lastString = (unsigned int *) cp;
        cp += sizeof(int);
        numStrings++;
        nxtBlockSize += sizeof(struct str_block) + 2 * sizeof(int);
      }
      else
      {

        n -= sizeof(int);
        if (scanner)          /* we've scanned more than we had to */
          scan_balance -= sizeof(int) * K;
      }
      sdesc->str_obj.s = cp;

      new += n;
    }
  }

  /* start a new garbage collection pass, as required by an allocation of size n */
  static char *
  help_alloc(n)
       register unsigned n;
  {
    register char *cp;
    int i;
    /*
     * can't charge this allocation to next phase, because we've
     * already collected on its time, so start scan_balance at zero
     */

    if ((toBack - relocated) > blockSize / 4)
      blockSize += blockSize / 2;                /* keep lots of breathing room in to-space */
  /* recalculate nxtBlockSize to correct for roundoff noise */
  nxtBlockSize = NewSize + StringGSize + ScanSize + n;
  if (blockSize < nxtBlockSize)                /* exchange from-space and to-space */
  {
    blockSize = nxtBlockSize;
    cp = fromSpace;
    fromSpace = realloc(toSpace, blockSize);
    toSpace = realloc(cp, blockSize);
    fromBack = fromSpace + blockSize;
    toBack = toSpace + blockSize;
  }
  else
  {
    cp = fromSpace;
    fromSpace = toSpace;
    toSpace = cp;
    cp = fromBack;
    fromBack = toBack;
    toBack = cp;
  }

  nxtBlockSize = 0;                   /* initialize state variables */
  scan_block = -1;

  relocated = scan_start = toBack;
  new = toSpace;

  scan_balance = 0;
  str_total = 0;
  mergers = NULL;
  blocks = (struct str_block *) (toBack - N - StringGSize);
  block_no = 0;
  numStrings = 0;
  lastString = NULL;

  scan_cnt = datasize;                /* scan global data first */
  scanner = (char *) dseg;
  scanned = (char *) &dseg[datasize];

  stackScan = stacksInUse.next;

  for (i = 0; i < TendedDescriptors; i++)
    tendDesc[i] = copy(tendDesc[i]);

  /* make the desired allocation */
  cp = new;
  new += n;
  return cp;
}

/* how many bits are needed to represent i? */
find_bits(i)
     int i;
{
  for (num_bits = 0; i; num_bits++)
    i /= 2;
  return num_bits;
}

/* determine the initial value of probe for a string region of length i
 *  this is all 1's in as many bit positions as are required to represent
 *  the length of the region, negated.
 */
init_probe(i)
     register int i;
{
  register int j;
  i = find_bits(i);
  for (j = 0; i--; )
    j = j << 1 | 1;
  return -j;
}

/* turn integer i inside out */
invert_bits(i, numbits)
     int i, numbits;
{
  int j = 0;

  while (numbits--)
  {
    j = (j << 1) | (i & 0x01);
    i /= 2;
  }
  return j;
}

/* calculate next two probe points, and adjust the running probe value */
inv_decr(ip, cp1, cp2)
     int *ip;
     char **cp1, **cp2;
{
  register int i, numbits;

  i = -*ip;
  numbits = find_bits(i);
  i = invert_bits(i, numbits);
  i--;

  if (i == 0)                   /* no more probes for this string region */
  {
    *ip = 0;
    *cp1 = *cp2 = NULL;
  }
  else
  {
    *cp1 = (char *) (ip + 1) + invert_bits(i, numbits);
    i--;
    *ip = -invert_bits(i, numbits);
    *cp2 = (char *) (ip + 1) - *ip;
  }
}

/* fix a string descriptor as it is relocated into to space.
 * this function prepares the string region pointed to by the
 * string descriptor for relocation
 */
fixString(sp)
     register struct O_string *sp;
{
  register struct str_block *bp;
  register struct t_string *tsp;

  if (sp->len && sp->s >= fromBlock && sp->s < fromBack)       /* only fix if data is in from-space */
  {
    if (*sp->block < 0)                 /* not marked yet */
    {
      bp = &blocks[block_no];
      inv_decr(sp->block, &bp->probes[0], &bp->probes[1]);
      bp->probe = *sp->block;
      *sp->block = block_no++;

      bp->start = sp->s;
      bp->stop = sp->s + sp->len;
```

```
        if (bp->probes[0] == NULL)
        {                                         /* all indices have been probed */
          str_total += sp->len;                   /* ready this region for merging with others */
          bp->u.s.next = mergers;
          mergers = bp;
        }
        else
        {
          bp->u.hole_starts[0] = bp->u.hole_starts[1] = NULL;
          bp->hole_stops[0] = bp->hole_stops[1] = NULL;

          test_probe(sp->s, sp->len, bp, 0);
          test_probe(sp->s, sp->len, bp, 1);
        }

        tsp = (struct t_string *) sp;
        bp->fix_list.next = tsp;
        tsp->next = bp->fix_list.prev = NULL;
        tsp->prev = &(bp->fix_list);
      }
      else
      {
        bp = &blocks[*sp->block];

        if (sp->s < bp->start)
        {
          if (bp->probe == 0)
            str_total += bp->start - sp->s;
          bp->start = sp->s;
        }

        if (sp->s + sp->len > bp->stop)
        {
          if (bp->probe == 0)
            str_total += sp->s + sp->len - bp->stop;
          bp->stop = sp->s + sp->len;
        }

        if (bp->probes[0])
          test_probe(sp->s, sp->len, bp, 0);

        if (bp->probes[1])
          test_probe(sp->s, sp->len, bp, 1);
      }
    }
}

/* test to see if the string data starting at s, of length len contributes
 *  any information regarding the size of the hole at bp->probes[ndx]
 */
static test_probe(s, len, bp, ndx)
     char *s;
     unsigned int len;
     struct str_block *bp;
     int ndx;
{
  if (s > bp->probes[ndx])
  {
    if (bp->hole_stops[ndx] == NULL)
      bp->hole_stops[ndx] = s;
    else if (bp->hole_stops[ndx] > s)
      bp->hole_stops[ndx] = s;
  }
  else if (s + len < bp->probes[ndx])
  {
    if (bp->u.hole_starts[ndx] == NULL)
      bp->u.hole_starts[ndx] = s + len;
    else if (bp->u.hole_starts[ndx] < s + len)
      bp->u.hole_starts[ndx] = s + len;
  }
  else
    bp->probes[ndx] = NULL;          /* no hole at this probe */
}

/* preparatory to scanning a string region, convert the string region
 *  control block to a standard form.
 */
static start_block(bp)
     struct str_block *bp;
{
  int newStrings;

  scan_balance -= sizeof(int);                    /* count header as scanned */
  if (bp->probe)
  {
    if (bp->probes[0] == NULL || bp->probes[0] < bp->start || bp->probes[0] > bp->stop)
    {                                             /* no hole at probes[0] */
      if (bp->probes[1] == NULL || bp->probes[1] < bp->start || bp->probes[1] > bp->stop)
      {                                           /* no hole at probes[1] */
        newStrings = 1;                           /* one region */
        bp->probes[0] = bp->stop;
        bp->u.hole_starts[0] = bp->stop;
        bp->hole_stops[1] = bp->hole_stops[0] = NULL;
      }
      else
      {                                           /* hole at probes[1] */
        newStrings = 2;                           /* two regions */
        bp->probes[0] = bp->probes[1];
        bp->probes[1] = bp->stop;
        bp->u.hole_starts[0] = bp->u.hole_starts[1];
        bp->hole_stops[0] = bp->hole_stops[1];
        bp->u.hole_starts[1] = bp->stop;
        bp->hole_stops[1] = NULL;
      }
    }
    else
    {                                             /* hole at probes[0] */
```

```
      if (bp->probes[1] == NULL || bp->probes[1] < bp->start
          || bp->probes[1] > bp->stop)
      {                                   /* no hole at probes[1] */
        newStrings = 2;                   /* two regions */
        bp->probes[1] = bp->stop;
        bp->u.hole_starts[1] = bp->stop;
        bp->hole_stops[1] = NULL;
      }
      else                                /* hole at probes[1] */
        newStrings = 3;                   /* three regions */
    }
    nxtBlockSize += newStrings * (sizeof(word) * (K + 2) / K + sizeof(struct str_block) + 2 * sizeof(int));
    numStrings += newStrings;
  }
}

/* scan_strings() is called by scan when only string data remains to be scanned */
static scan_strings()
{
  register int i;
  int len;
  static char *merge_ptr;
  static int *merge_base;
  static word offsets[3];             /* for each of three possible sub-regions, */
  static int *bases[3];               /*  keep offsets and base pointers */
  struct t_string *tsp;
  struct O_string *sp;

  if (scan_block == -1)
  {
    /* this is the first invocation of scan_strings() on this pass of the
     *  garbage collector.  initialize things.
     */
    if (str_total)     /* we have regions to merge */
    {
      relocated -= str_total + sizeof(int);
      merge_base = (int *) relocated;
      merge_ptr = (char *) (merge_base + 1);
      *merge_base = init_probe(str_total);
      nxtBlockSize += (str_total + sizeof(int)) * (K + 2) / K + sizeof(struct str_block) + 2 * sizeof(word);
    }

    scan_block = 0;
    if (block_no)
      start_block(&blocks[0]);
    bases[0] = NULL;
  }

  /* process linked list of merge regions first */
  while ((bp = mergers) != NULL && scan_balance > 0)
  {
    i = (scan_balance + 1) / 2;
    if (bp->stop - bp->start <= i)
      i = bp->stop - bp->start;

    movmem(bp->start, merge_ptr, i);
    scan_balance -= 2 * i;
    merge_ptr += i;
    if ((bp->start += i) == bp->stop)
    {
      mergers = bp->u.s.next;
      bp->u.s.offset = merge_ptr - bp->stop;            /* save offset of move */
    }
  }
  bp = &blocks[scan_block];
```

```
while (scan_balance > 0)
{
  if (scan_block < block_no)
  {
    if (bp->probe == 0)                              /* merge this region */
    {
      if (tsp = bp->fix_list.next)
      {
        if (bp->fix_list.next = tsp->next)
          tsp->next->prev = &bp->fix_list;

        sp = (struct O_string *) tsp;
        sp->title = STRING;
        sp->block = merge_base;
        sp->s += bp->u.s.offset;
        scan_balance -= 2 * sizeof(word);
      }
      else
      {
        scan_balance -= sizeof(int);                 /* consider the region prefix copied */
        if (++scan_block < block_no)
          start_block(bp = &blocks[scan_block]);
      }
    }
    else if (bp->start)               /* copy text for first subregion */
    {
      if (bases[0] == NULL)
      {
        relocated -= (len = bp->u.hole_starts[0] - bp->start) + sizeof(int);
        bases[0] = (int *) relocated;
        *bases[0] = (bp->hole_stops[0] || find_bits(len) < find_bits(bp->probe))? init_probe(len): bp->probe;
        offsets[0] = (relocated + sizeof(int)) - bp->start;
        bases[1] = NULL;
      }

      i = (scan_balance + 1) / 2;
      if (bp->u.hole_starts[0] - bp->start <= i)
        i = bp->u.hole_starts[0] - bp->start;

      movmem(bp->start, bp->start + offsets[0], i);
      scan_balance -= 2 * i;
      if ((bp->start += i) == bp->u.hole_starts[0])
        bp->start = NULL;             /* mark first subregion as copied */
    }
    else if (bp->hole_stops[0])       /* copy text for second subregion */
    {
      if (bases[1] == NULL)
      {
        relocated -= (len = bp->u.hole_starts[1] - bp->bp->hole_stops[0]) + sizeof(int);
        bases[1] = (int *) relocated;
        *bases[1] = init_probe(len);
        offsets[1] = (relocated + sizeof(int)) - bp->start;
        bases[2] = NULL;
      }

      i = (scan_balance + 1) / 2;
      if (bp->u.hole_starts[1] - bp->hole_stops[0] <= i)
        i = bp->u.hole_starts[1] - bp->hole_stops[0];
```

```
          movmem(bp->hole_stops[0], bp->hole_stops[0] + offsets[1], i);
          scan_balance -= 2 * i;
          if ((bp->hole_stops[0] += i) == bp->u.hole_starts[1])
            bp->hole_stops[0] = NULL;           /* mark second subregion as copied */
        }
        else if (bp->hole_stops[1])             /* copy text for third subregion */
        {
          if (bases[2] == NULL)
          {
            relocated -= (len = bp->stop - bp->hole_stops[1]) + sizeof(int);
            bases[2] = (int *) relocated;
            *bases[2] = init_probe(len);
            offsets[2] = (relocated + sizeof(int)) - bp->start;
          }

          i = (scan_balance + 1) / 2;
          if (bp->stop - bp->hole_stops[1] <= i)
            i = bp->stop - bp->hole_stops[1];

          movmem(bp->hole_stops[1], bp->hole_stops[1] + offsets[2], i);
          scan_balance -= 2 * i;
          if ((bp->hole_stops[1] += i) == bp->stop)
            bp->hole_stops[1] = NULL;
        }
        else if (tsp = bp->fix_list.next)       /* update string pointers for region */
        {
          if (bp->fix_list.next = tsp->next)
            tsp->next->prev = &bp->fix_list;

          sp = (struct O_string *) tsp;
          sp->title = STRING;

          /* see which subregion string data was copied into */
          if (sp->s < bp->probes[0])
            i = 0
          else if (sp->s < bp->probes[1])
            i = 1;
          else
            i = 2;

          sp->block = bases[i];
          sp->s += offsets[i]

          scan_balance -= 2 * sizeof(word);
        }
        else                                    /* this string region has been entirely relocated */
        {
          scan_balance -= sizeof(int);          /* consider the region prefix copied */
          if (++scan_block < block_no)
          {
            start_block(bp = &blocks[scan_block];
            bases[0] = NULL;
          }
        }
      }
      else
        return gc_cleanup();            /* all out of string descriptors */
    }
    return 0;
  }
```

```
/* cleanup after garbage collection terminates */
static gc_cleanup()
{
  scanner = NULL;
  nxtBlockSize = NewSize + StringGSize + ScanSize;
  return 1;
}

/* allocate a stack from the free list */
Descriptor alloc_stack()
{
  register struct stack_head *newsp;

  if (newsp = freeStack.next)
  {
    numStacks++;
    nxtBlockSize += StackSize * sizeof(Descriptor) / K;
    if (newsp->next)
      newsp->next->prev = (struct O_stack *) &freeStack;
    freeStack.next = newsp->next;

    if (newsp->next = stacksInUse.next)
      stacksInUse.next->prev = newsp;
    newsp->prev = (struct O_stack *) &stacksInUse;
    stacksInUse.next = newsp;
  }
  return newsp;
}

/* return stack to its free pool */
free_stack(sp)
     register struct O_stack *sp;
{
  numStacks--;
  nxtBlockSize -= StackSize * sizeof(Descriptor) / K;

  if (stackScan == sp)
    stackScan = sp->next;

  if (sp->next)
    sp->next->prev = sp->prev;
  sp->prev->next = sp->next;

  if (sp->next = freeStack.next)
    freeStack.next->prev = sp;
  freeStack.next = sp;
}

/* scan stacks, assume stackScan is not NULL */
static scan_stack()
{
  register Descriptor *sp;

  if ((sp = stackScan->scan_sp) == NULL)
    sp = stackScan->sp;
  while (scan_balance > 0)
  {
    (*sp) = copy(*sp);
    sp++;
    if (sp >= &stackScan->stack[StackSize])
    {
      if ((stackScan = stackScan->next) == NULL)                /* no more stacks to scan */
        return;
      if ((sp = stackScan->scan_sp) == NULL)
        sp = sdesc->sp;
    }
  }
  stackScan->scan_sp = sp;
}

/* garbage collect as much as is required for an allocation of size n */
/*  return non-zero when garbage collection completes */
scan(n)
     unsigned int n;
{
  int offset, type;

  scan_balance += n * K;
  while (scan_balance > 0)
  {
    if (stackScan)
      scan_stack();
    else if (scan_cnt)
    {
      scan_cnt--;
      *((Descriptor *) scanner) = copy(*((Descriptor *) scanner));
      scanner += sizeof(Descriptor);
      scan_balance -= sizeof(Descriptor);
    }
    else
    {
      if (scanner == scanned)
      {
        if (scan_start == relocated)    /* no more descriptors to scan */
          return scan_strings();
        else
        {
          scanned = scan_start;
          scan_start = scanner = relocated;
        }
      }

      type = ((Descriptor) scanner)->ab_template.title;
      if (type == STRING)
      {
        fixString((Descriptor) scanner);
        scanner += sizeof(struct O_string);
        /* string descriptor is only halfway scanned at this point */
        scan_balance -= sizeof(int) + sizeof(char *);
      }
      else
      {
        scan_balance -= descOffset[type];
        if ((scan_cnt = numDesc[type]) < 0)
          scan_cnt = ((Descriptor) scanner)->ab_record.num_fields;
        scanner += descOffset[type];
      }
    }
  }
  return 0;
}

/* copy an object into to-space if necessary, returning a pointer to the relocated object */
Descriptor copy(ptr)
     register Descriptor ptr;
{
  register int i, type;

  if (ptr == NULL
      || (ptr < fromBlock || ptr > fromBack)
      || ((type = ptr->ab_template.title) == STACK))
    return ptr;
  else if (type == MARKED)
    return ptr->ab_template.forward;
  else
  {
    if (numDesc[type] != -1)                    /* then fixed size */
      i = objSize[type];
    else                                        /* variable size */
      i = objSize[type] + ptr->ab_record.num_fields * sizeof(Descriptor);

    relocated -= i;
    scan_balance -= i;
    nxtBlockSize += i + (i + K - 1) / (K/2);
    movmem((char *) ptr, relocated, i);
    ptr->ab_template.title = MARKED;
    ptr->ab_template.forward = (Descriptor) relocated;
    return (Descriptor) relocated;
  }
}
```