The Visualization of Dynamic Memory Management in the Icon Programming Language

Ralph E. Griswold and Gregg M. Townsend

TR 89-30

December 22, 1989

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

1. Introduction

The Icon programming language [1] supports many types of data — strings, records, lists, sets, tables, and so on. Data objects vary in size by type and, for some types, from value to value. Object size ranges from one byte to many thousands of bytes. Storage management is automatic. Space for an object is allocated when the object is created, and unused space is collected as necessary [2].

It is typical for Icon data objects to be created and used in a transient fashion, so that the space they occupy is needed only for a short period of time. Consequently, many Icon programs exhibit a substantial amount of "storage throughput".

The management of storage — allocation and garbage collection — is intended to be transparent to the programmer and user, placing the burden on the implementation. In most situations, the Icon programmer does not need to think about the allocation of storage or its consequences on program performance. Nonetheless, in cases where the amount of memory is limited or where programs require unusually large amounts of space for the data they manipulate, the management of memory may be a concern both to the programmer and user. Furthermore, since storage management is complicated and may require substantial resources, its performance is a significant concern in the implementation of Icon.

Because of the diversity of data types and the varying sizes of data objects, as well as a wide range of applications with differing storage demands, it is difficult to characterize the behavior of storage management. As with many complex implementations, relatively little is known about what actually goes on in Icon's storage-management system or how well it performs in practice. The problem is not so much in collecting data but in interpreting it.

These considerations motivated the memory-monitoring system for Icon that is described in this report. This system emphasizes the visualization of memory management — allocation and garbage collection — dynamically, as it takes place during the execution of Icon programs.

The memory-monitoring system consists of two parts:

• Instrumentation that produces *allocation history files*.

• Programs that convert allocation history files to visual representations of Icon's storage regions.

The production of allocation history files is an option that is available when an Icon program is run. It does not require changes to the program and it does not affect program behavior.

There are several kinds of visualization programs. Some programs produce animated displays on color graphics monitors and allow user interaction. Other programs produce frames — color image files that can be printed or displayed separately.

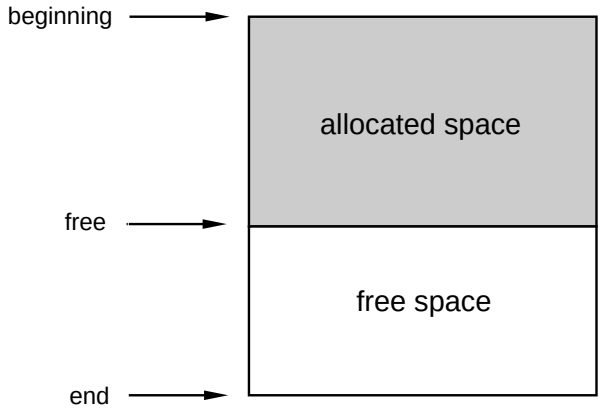
Icon's data regions are shown with different types of data objects displayed in different colors. For the animation programs, a data object appears on the display in its place in memory when it is created and the space for it is allocated. When a garbage collection occurs, the processing of data objects is shown by changing their colors and

eventually redrawing the display after collection. The effect is a color “cinematic” representation of Icon’s allocated data as program execution proceeds. The programs that produce frames provide snapshots of the storage regions at specified times.

This report first briefly describes the implementation of storage management in Icon, then the memory monitoring system, and finally how the information obtained from memory monitoring can be used.

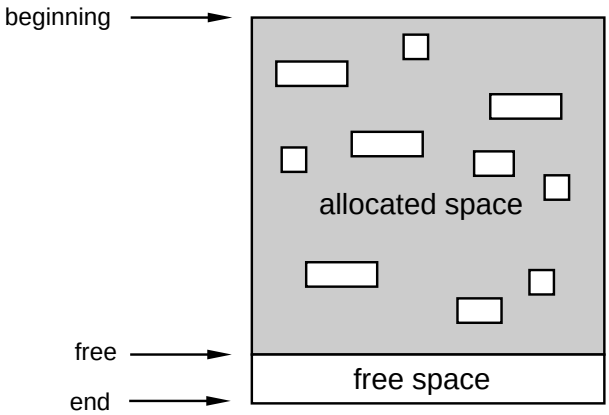
2. Storage Management in Icon

Icon allocates space for objects that are created during program execution in two main regions: a string region and a block region. The string region consists of characters, while the block region contains structures and related



objects. Allocation in the string region is in terms of bytes, while allocation in the block region is in terms of “words”. A word typically is four bytes. Some implementations of Icon also have a static region, which is not described here.

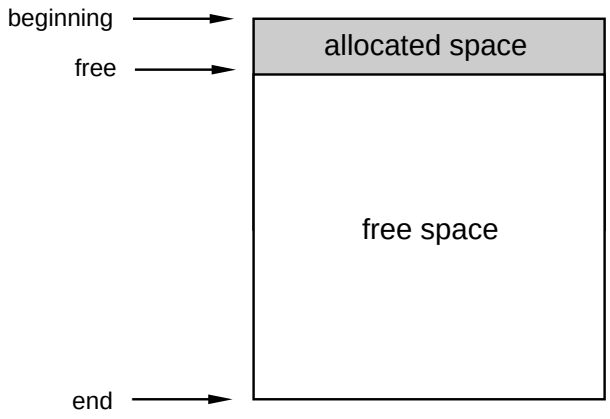
Allocation proceeds in the same manner in both main regions. The regions initially are empty and bounded by pointers. As space is needed, it is provided starting at the beginning of the region. A “free” pointer is incremented to



mark the boundary between *allocated space* and *free space*:

If there is not enough available free space to satisfy an allocation request, a *garbage collection* is performed to reclaim space occupied by objects that are no longer needed.

The garbage collection process is fairly complicated, since it is necessary to locate all objects that may be needed for subsequent program execution. See [2] for details. Objects that need to be saved typically are scattered throughout the string and block regions:



Once the objects to be saved are identified, they are relocated toward the beginning of their region, compressing the allocated space and making more free space available so that allocation can proceed:

There are several interesting issues in such a storage management system. For example:

• How much storage is allocated for different kinds of values?

• How frequently is garbage collection performed? How long does it take?

• How much space is reclaimed by garbage collection?

Answers to these questions lead to others, such as

• What are the effects of different kinds of programming techniques on storage management?

• How do the sizes of the regions affect the performance of storage management?

• How does the performance of storage management vary with different kinds of programs?

3. Instrumentation of Storage Management

There are several approaches to answering such questions. The main ones are analysis, simulation, and instrumentation.

Analysis can be used to characterize the performance of the algorithms used in storage management. Unfortunately, however, analytic techniques have only limited usefulness for systems as complex as storage management in Icon, especially with the wide variety of programs and data that are encountered in use.

Simulation can be used to abstract the essential properties of a complex system without using “invasive” techniques that require modifications to the system itself. Such simulations, however, are difficult to perform and are subject to error, especially if essential aspects of the system are not identified properly.

The third alternative, instrumentation, provides detailed information about what actually goes on. It requires modifying the system itself and developing tools for evaluating the results. Instrumentation is the basis of the results described in this report.

Instrumentation of Icon’s storage management system consists of strategically placed code in the implementation of Icon that produces an *allocation history*. This history contains a record of every object allocated: the region in which it is allocated, its location, its type, and its size. The allocation history also records the details of garbage collection: the objects to be kept and the results of compaction. The production of an allocation history has no harmful effects on running programs.

The instrumentation normally is disabled, but it can be enabled when a program is run without modifying the program or affecting its behavior (except for increasing execution time somewhat). When the instrumentation is enabled, the allocation history is written to a file. This file then can be processed after execution of the program.

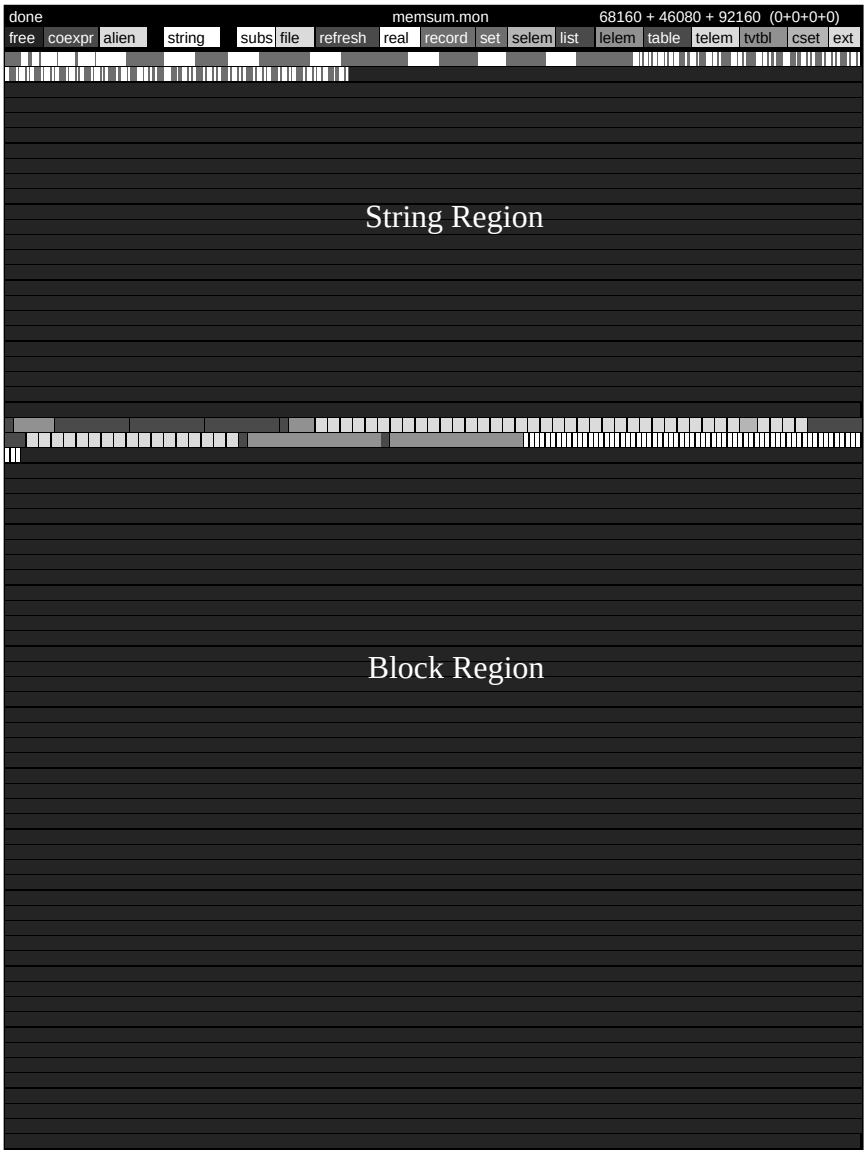
4. Visualizing Storage Management

The results of monitoring program execution using the kind of instrumentation described above are voluminous and detailed. A typical program allocates many thousands of objects and may garbage collect dozens of times.

Usually such instrumentation data is processed by programs that extract salient information, summarize activities of interest, and present this information in a tabular or graphic form. Summarization, which is used to suppress detail that is impossible for the human mind to comprehend in raw form, usually provides only static information and fails to show the dynamic nature of the process that is monitored. It also may fail to report important information simply because it is not anticipated.

The alternative to this conventional method of presenting the results of monitoring is to present the detailed data in a way the human mind can grasp — visualization. This method allows the human mind's extremely powerful abilities for interpreting images to be applied to the data. Patterns and relationships may be detected that never would be observed in data presented in conventional ways.

For the Icon storage management system, visualization consists of a two-dimensional color display of Icon's storage regions. When an object is allocated, it appears on the display in the place corresponding to its location in memory and in a size that corresponds to the amount of memory it occupies. Different colors are used to distinguish objects of different data types. Garbage collection is shown dynamically, with each object to be saved identified as it is found. On systems that support user interaction, the display can be halted temporarily for examination. The visualization programs also can be instructed to pause at significant times, such as when a garbage collection is about to occur. Other programs produce frames — snapshots of the allocated data region at selected times. These snapshots can be used to make slides and printed images for presentations and reports such as this one.



The details of the display vary somewhat depending on the capabilities of the device used. In general, the display consists of a legend with descriptive information, followed by the memory regions. A typical display is shown in black and white at the right.

The top line shows the program status at the left, the name of the allocation history file in the middle, and storage information at the right. The first three figures in the storage information are the sizes, in bytes, of the static, string, and block regions. The figures in parentheses give the number of garbage collections caused by allocation in the static, string, and block regions and as the result of explicit calls to the function collect(), respectively. The second line of the display shows the colors assigned to different kinds of data. This provides a ready reference for interpreting the rest of the display.

The storage regions follow in order. The static region normally is not shown,

since it is relatively uninteresting. The string region immediately follows the legend. The free space at the end is dark, distinguishing it from the allocated space at the beginning of the region. The end of each allocated string is darkened to show the divisions between strings. Since the resolution of most displays is less than the width of a single character, the divisions between individual strings are only approximate. The block region follows the string region, with its free space also darkened to distinguish it from its allocated space.

Plate 1 at the end of this report shows displays of allocation in a program that sorts electronic news articles. Allocation in the string region occurs as news articles are read into the program. Allocation in the block region results from structures used to construct a database.

In this program, the block region fills up first. Display A shows the allocated regions at this point. The resulting garbage collection identifies objects to be saved, which are "marked" by changing their color to black as shown in display B. Next, the colors are reversed, so that marked objects are shown in their proper color and all other objects (which will be "collected") are shown in black. The situation is shown in Display C. Finally the marked objects are compacted as shown display D. The string region at this point is shown without demarcation between allocated strings. Note that garbage collection eliminates a considerable portion of the allocated space. Objects eliminated in this fashion are transient and are the byproduct of temporary computations that are performed by the program — storage throughput. Program execution then continues.

6. Applications

The visualization of memory management in Icon has been valuable in several ways:

- Understanding program behavior.
- Evaluating alternative Icon programming techniques.
- Identifying Icon programming techniques that use storage unnecessarily.
- Identifying and correcting inefficiencies in the implementation of Icon.

Understanding Program Behavior

Once a programmer is familiar with the memory displays, they provide a way to watch program execution. It is easy to see, for example, that a particular program reads in data, makes a set of records, sorts the set, and then builds a table.

While memory displays are no substitute for algorithm animation, they provide a global picture of data structures as opposed to focusing on local effects. Memory displays also are available for any program at any time.

Since data structures are such an important part of many Icon programs, memory displays also give insight into the resources different kinds of applications require and sometimes suggest different approaches to program formulation.

Alternative Programming Techniques

Icon is a rich language with many features. There often are several ways to do the same thing. In particular, there often are several ways of representing the same information within a program.

A program for producing simple concordances provides a good example. The main problem is associating words with the line numbers in which they occur. A table is the natural choice for holding the words, but there are several possibilities for keeping track of the line numbers. In this particular concordance program, a line number for a word is given only once even if the word occurs several times on that line. Consequently, a way is needed to discard duplicate citations. The program, listed in Appendix A, uses tables to hold line numbers. A line number is marked as being associated with a word by assigning a non-default value to the corresponding element. A string that lists all the line numbers is constructed at the end of the program, after all the input has been read.

While this technique uses Icon's features to advantage, it also is expensive in terms of allocated storage. In fact, tables really are used to simulate sets — a common practice, since sets were not available in early versions of Icon. Appendix B lists a revised version of the program using sets. Since a value can be in a set only once, inserting a line number more than once has no effect — the set insertion mechanism automatically eliminates duplicates. This technique also is expensive in terms of allocated storage, although less so than using tables. An alternative technique, shown in the program listed in Appendix C, is to use lists for the line numbers, appending a line number for a citation only if it is not already in the list. Line numbers are appended to the lists as they occur, so it is only necessary to check the last value in a list. Yet another approach is to construct the strings of line numbers as the data is

read. Discarding duplicates is slightly trickier using this approach, since it requires pattern matching. See Appendix D.

Plate 2 shows a display of memory just prior to the first garbage collection for each of these programs. The version of the program that uses tables runs out of space in the block region, as shown in the upper-left display. Using sets instead of tables allows the program to process more data before the first garbage collection, as shown by the larger amount of allocated string storage in the display at the upper right. Using lists allows the program to process even more data before the first garbage collection, as shown by the display at the lower left. In fact, the first garbage collection occurs because the string region is full. Using strings instead of structures to hold line numbers produces a quite different effect, as shown by the display at the lower right. The difference in the string region, compared to the other displays, is a consequence of the fact that this program, unlike the others, builds strings while data is being read.

Despite the differences in handling duplicate line numbers, the four programs basically are the same. It is worth noting that the execution time for the programs decreases as less memory-intensive techniques are used. (This relationship does not always hold, however.) The most serious problem with the table and set approaches is the total amount of memory they require — much so that it is impractical to use them on computers with a small amount of memory.

Unnecessary Allocation

One aspect of Icon that makes it easy to use is automatic type checking and conversion. Type conversion, however, can be expensive in terms of storage allocation (and time). Consequently, unnecessary conversion is a potential source of inefficient programming.

There are two common causes of such inefficiency: specifying string literals where csets are needed and constructing csets unnecessarily in loops. For example,

```
while upto("aeiou") do …
```

causes the string "aeiou" to be converted to a cset every time upto() is evaluated, while a cset literal would not require any conversion. Similarly,

```
while upto(&lcase ++ &digits) do …
```

requires a cset to be constructed every time through the loop, while it could be constructed once outside the loop and referenced by a local identifier. Of course, this problem would not exist if the Icon compiler folded constants.

Another example of unnecessary allocation is illustrated by the following segment of code used in an early version of the concordance program using sets to hold the line numbers:

```
if /uses[word] := set([lineno]) then return       # new set
  else {
    insert(uses[word], lineno)
    return
  }
```

The motivation for the first line of this segment was to create a new set with the line number already in it, rather than creating an empty set and then inserting the line number. The construction of a non-empty set, however, requires a list for the argument — [lineno]. Consequently, a list is created and used transiently for every newly created set. A more efficient approach is

```
/uses[word] := set()
insert(uses[word], lineno)
return
```

Unnecesary allocation often is obvious when a program is monitored. Large numbers of lists or csets make a striking pattern and, when unexpected, alert the viewer that something unintended is occurring.

Problems with the Implementation of Storage Management

In a system as complicated as Icon’s storage management, experience, efficient algorithms, and good coding are not enough to ensure good performance. In fact, even the implementors of such a system usually do not fully understand how it works in practice and the problems it may have. Displays of memory management have proved this.

An example is a heuristic used to accommodate both positional and deque access to lists [2]. Since values can be added to any list, space for possible additional values is provided when a list is created to avoid allocating another block as soon as a value is added to a list. Many lists remain fixed in size, however, and this additional space is unused. Although the potentially wasted space was recognized when the heuristic was devised, its impact on storage allocation was not appreciated until memory displays were examined. The heuristic was changed as a result, and extra space now is allocated only for empty lists, since these are the main candidates for added values [4].

Two aspects of storage management are particularly troublesome: thrashing and garbage collection in the presence of long-lived data. Thrashing occurs when most allocated objects must be saved and only a few can be collected for subsequent allocation. In extreme cases, garbage collection may occur very frequently (perhaps for every newly created object) and program execution may slow to a crawl as most of the execution time is spent in garbage collection. Thrashing is a well-known problem to implementors, but an Icon programmer may not even guess what is wrong. Memory displays show trashing dramatically.

Although Icon's storage management system has mechanisms for reducing thrashing, it remains a serious problem in some programs. Interestingly, thrashing often can be avoided by using larger memory regions — an option available to most Icon programmers. Experience has shown that programmers have to *see* the difference in order to appreciate that they should exercise this option.

There is a related problem for programs that produce a large amount of long-lived data, such as a large in-memory word list that is created at the beginning of program execution and used until the end of the program. Such long-lived data has to be processed at every garbage collection, although it contributes nothing to reclaiming space for subsequent allocation. Even if thrashing does not occur, program performance may be adversely affected because every object that has to be saved must be located and processed during each garbage collection. While the existence of this problem has been known since the original design of Icon's storage management system, it was not until it was observed on memory displays that its significance was fully appreciated.

7. The Implementation of Memory Monitoring

The instrumentation that produces allocation history files consists of strategically placed function calls that record allocation, note garbage collection, trace the garbage-collection process, and so forth. These function calls are produced by macros, which can be defined to produce empty text instead of function calls under the control of conditional compilation. This allows the memory-monitoring code to be omitted, although it adds less than 4% to the size of the Icon run-time system.

Allocation history information is written only if the environment variable MEMMON, which specifies the name of a file, is defined. If this environment variable is not set, the memory-monitoring functions simply return without doing anything. The overhead for calling these functions is undetectable unless allocation history information is being written.

The description of the information contained in allocation history files is given in Reference 3.

The displays contained in this report were produced under Version 7.9 of Icon [4].

8. Visualization Programs

There are several programs for producing displays [5]. These include drivers for the AED and RasterTech color-graphics monitors, color Quickdraw for the Macintosh, and color PostScript (which is used in this report).

The processing needed for visualization programs consists of two parts: the analysis of allocation history files and device-specific actions. Much of the processing is common to all visualization programs and is isolated in device-independent routines [6]. Producing a new visualization program consists largely of providing device-specific code.

9. Comments on Visualization

Visualization of storage management in Icon has provided many new insights into storage management. It also has suggested other possibilities for program visualization and pointed out some inherent difficulties.

In the visualization of physical objects, such as complex molecules, there usually is a natural geometric model. In the visualization of program activity, there may not be a natural geometry. It is necessary to find a model that is both faithful to the activity and that also is easy to comprehend visually.

For Icon's storage management system, that is comparatively straightforward. Each data object is represented by a rectangle whose width is proportional to the amount of space the object occupies in memory. The height is the same for all rectangles and is used to give objects a visual aspect. The two-dimensional layout, which constitutes folding a long "strip" of linear memory, is essential both to the display and to its interpretation. While this folding is natural, it sometimes introduces visual artifacts. For example, some displays show repeating patterns because of the vertical juxtaposition produced by folding. Whether these patterns are visually obvious depends on the dimensions and resolution of the display. Whether or not these patterns are meaningful or not depends on the program and, unfortunately, on the layout of the display.

The use of color to differentiate between different types of objects is intended to make direct use of color discrimination in the human visual system. Color perception is, however, a complex and relatively poorly understood phenomenon with many psychological aspects [7]. Color perception is centrally important in the design of the visualization programs. Different color palettes applied to the same display produce dramatically different reactions from viewers. A program that is "interesting" with one color scheme is "dull" with another. An example of the effect of different palettes is shown in Plate 3. Selecting palettes was, in fact, one of the more difficult aspects of the design of the visualization programs.

Different viewers have considerably different reactions to displays of Icon's storage management system. These reactions depend to a large extent on the degree of understanding the viewer has of the meaning of the display. Perhaps the most bizarre "application" of the memory management displays is a series of paintings by an artist who based his work on color photographs of Icon's memory management, but who had no understanding of what they represented.

Acknowledgement

Mark Emmer wrote the visualization program for the Macintosh. He also provided several helpful suggestions about memory monitoring and the displays.

References

1. *The Icon Programming Language*, Ralph E. Griswold and Madge T. Griswold, Prentice-Hall, Inc., 1983.
2. *The Implementation of the Icon Programming Language*, Ralph E. Griswold and Madge T. Griswold, Princeton University Press, 1986.
3. *Icon Allocation History Files*, Gregg M. Townsend, technical report IPD98a, Department of Computer Science, The University of Arizona, 1989.
4. *Supplementary Information for the Implementation of Version 7.9 of Icon*, Ralph E. Griswold, technical report IPD-51d, Department of Computer Science, The University of Arizona, 1989.
5. *Programs for Visualizing Icon Memory Management*, Gregg M. Townsend, technical report IPD99a, Department of Computer Science, The University of Arizona, 1989.
6. *Notes on MemMon Internals*, Gregg M. Townsend, technical report IPD97a, Department of Computer Science, The University of Arizona, 1989.
7. *The Forms of Color*, Karl Gerstner, MIT Press, 1986.

Appendix A — Concordance Program Using Tables

```
global uses, lineno, width

procedure main(args)
   local word, line

   width := 15                                   # width of word field
   uses := table()
   lineno := 0

   every tabulate(words())                       # tabulate all the citations

   output()                                      # print the citations

end

#  Add line number to citations for word
#
procedure tabulate(word)

   /uses[word] := table()
   uses[word][lineno] := 1
   return

end

#  Generate words
#
procedure words()
   local s, line

   while line := read() do {
      lineno +:= 1
      write(right(lineno,6),"  ",line)
      map(line) ? while tab(upto(&letters)) do {
         s := tab(many(&letters))
         if *s < 3 then next                     # skip short words
         suspend s
         }
      }

end

#  Print the results
#
procedure output()
   local word, line, numbers

   write()

   uses := sort(uses,3)                          # sort citations
   while word := get(uses) do {
      line := ""
      numbers := sort(get(uses),3)
      while line ||:= get(numbers) || ", " do
         get(numbers)                            # skip marking value
      write(left(word,width),line[1:-2])
      }

end
```

Appendix B — Concordance Program Using Sets

```
global uses, lineno, width

procedure main(args)
   local word, line

   width := 15                                   # width of word field
   uses := table()
   lineno := 0

   every tabulate(words())                       # tabulate all the citations

   output()                                      # print the citations

end

#  Add line number to citations for word
#
procedure tabulate(word)

   /uses[word] := set()
   insert(uses[word],lineno)
   return

end

#  Generate words
#
procedure words()
   local s, line

   while line := read() do {
      lineno +:= 1
      write(right(lineno,6),"  ",line)
      map(line) ? while tab(upto(&letters)) do {
         s := tab(many(&letters))
         if *s < 3 then next                     # skip short words
         suspend s
         }
      }

end

#  Print the results
#
procedure output()
   local word, line, numbers

   write()

   uses := sort(uses,3)                          # sort citations
   while word := get(uses) do {
      line := ""
      numbers := sort(get(uses))
      while line ||:= get(numbers) || ", "
      write(left(word,width),line[1:-2])
      }

end
```

Appendix C — Concordance Program Using Lists

```
global uses, lineno, width

procedure main(args)
   width := 15                                   # width of word field
   uses := table()
   lineno := 0

   every tabulate(words())                       # tabulate all the citations

   output()
end

#  Add line number to citations for word
#
procedure tabulate(word)

   if /uses[word] := [lineno] then return
   else {
      if uses[word][–1] ~= lineno then put(uses[word],lineno)
      return
      }
end

#  Generate words
#
procedure words()
   local s, line

   while line := read() do {
      lineno +:= 1
      write(right(lineno,6),"  ",line)
      map(line) ? while tab(upto(&letters)) do {
         s := tab(many(&letters))
         if *s < 3 then next                     # skip short words
         suspend s
         }
      }
end

#  Print the results
#
procedure output()
   local word, line, numbers

   write()

   uses := sort(uses,3)                          # sort citations
   while word := get(uses) do {
      line := ""
      numbers := get(uses)
      while line ||:= get(numbers) || ", "
      write(left(word,width),line[1:–2])
      }
end
```

Appendix D — Concordance Program Using Strings

```
global uses, lineno, width

procedure main(args)
   local word, line

   width := 15                                   # width of word field
   uses := table("")
   lineno := 0

   every tabulate(words())                       # tabulate all the citations

   output()                                      # print the citations

end

#  Add line number to citations for word
#
procedure tabulate(word)

   if uses[word][–2 –: ∗lineno] == lineno then return
   else {
      uses[word] ||:= lineno || ", "             # new line number
      return
      }

end

#  Generate words
#
procedure words()
   local s, line

   while line := read() do {
      lineno +:= 1
      write(right(lineno,6),"  ",line)
      map(line) ? while tab(upto(&letters)) do {
         s := tab(many(&letters))
         if ∗s < 3 then next                     # skip short words
         suspend s
         }
      }

end

#  Print the results
#
procedure output()
   local word

   write()

   uses := sort(uses,3)                          # sort citations
   while word := get(uses) do
      write(left(word,width),get(uses)[1:–2])

end
```

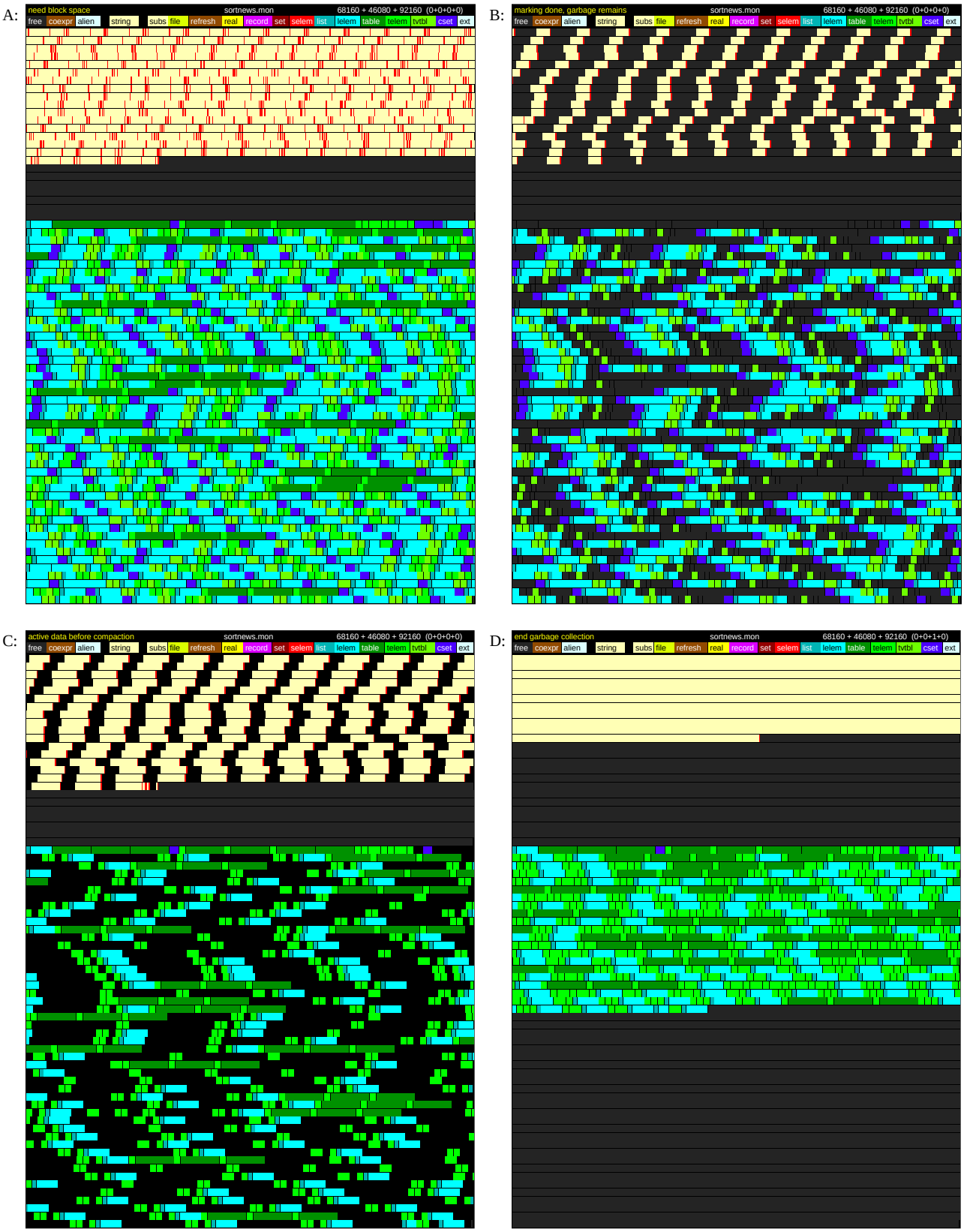


Plate 1. Four Stages of Garbage Collection

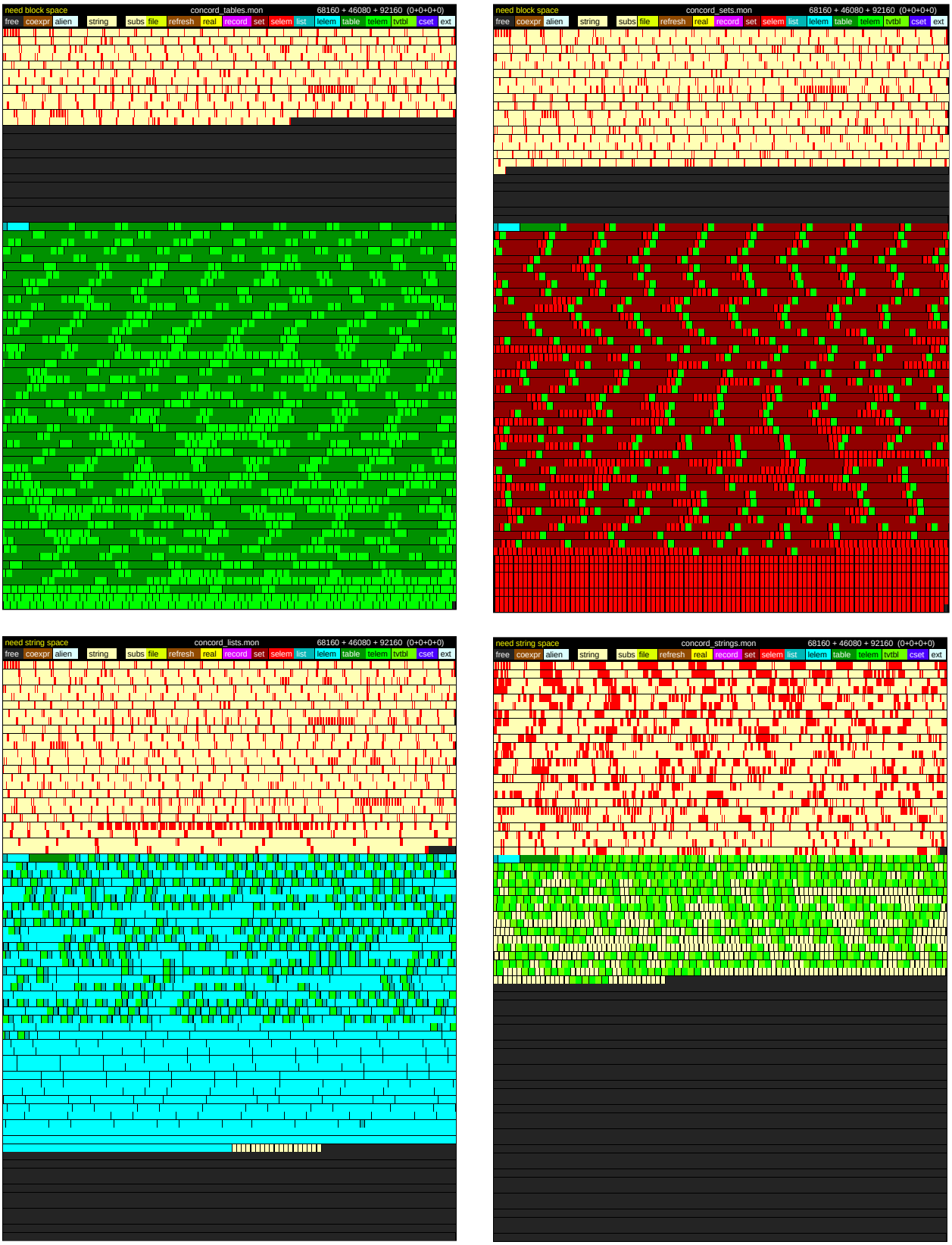


Plate 2. Four Different Concordance Programs Just Prior to First Garbage Collection

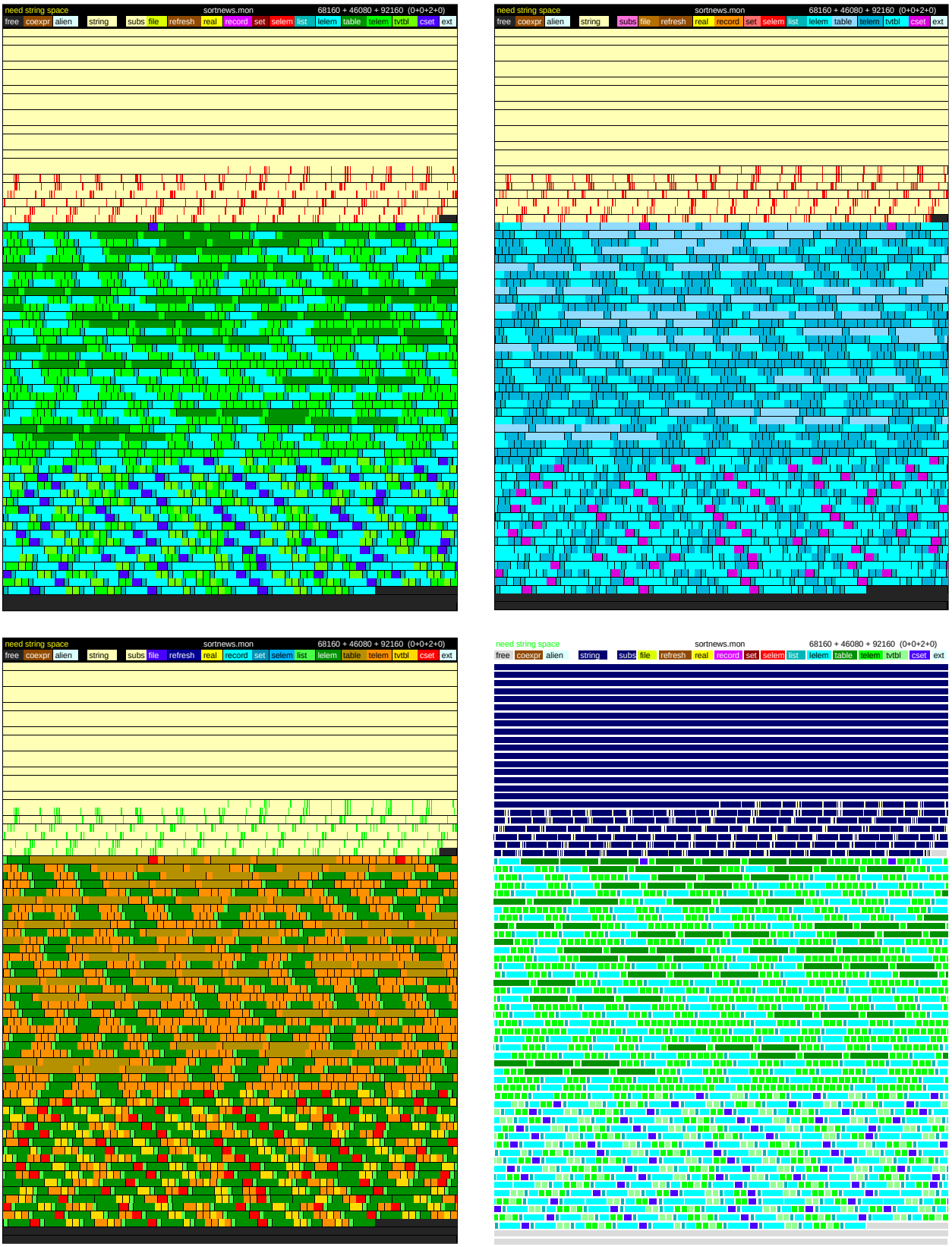


Plate 3. One Display Presented with Four Different Palettes