Real-Time Garbage Collection on General-Purpose Machines

Taiichi Yuasa
Research Institute for Mathematical Sciences, Kyoto University, Kyoto, Japan

An algorithm for real-time garbage collection is presented, proved correct, and evaluated. This algorithm is intended for list-processing systems on general-purpose machines, i.e., Von Neumann style serial computers with a single processor. On these machines, real-time garbage collection inevitably causes some overhead on the overall execution of the list-processing system, because some of the primitive list-processing operations must check the status of garbage collection. By removing such overhead from frequently used primitives such as pointer references (e.g., Lisp car and cdr) and stack manipulations, the presented algorithm reduces the execution overhead to a great extent. Although the algorithm does not support compaction of the whole data space, it efficiently supports partial compaction such as array relocation.

1. INTRODUCTION

Garbage collection is the most popular method in list-processing systems such as Lisp to reclaim discarded list cells (cons cells in Lisp terminology). Although there are several variations of garbage collection, they share essentially the same scheme: Available cells are collected together to form a *freelist*, from which one cell is removed each time the application program requires a new cell. When the freelist is exhausted, the application program is temporarily suspended and the *garbage collector* program begins to run. The whole process of the garbage collector consists of two major phases. The *mark phase* determines which cells are in use (or *accessible*), by traversing list structures in use. The *sweep phase* then puts all inaccessible cells into the freelist. In addition, some list-processing systems have the compaction phase (or relocation phase) in which all accessible cells are moved into a contiguous memory area and pointer references to the relocated cells are updated appropriately. After the execution of the garbage collector, the execution of the application program is resumed. This kind of a garbage collector, which suspends the application program, is called a *stop garbage collector*.

The primary disadvantage of garbage collection is that it periodically suspends the application program. Roughly speaking, the time for a garbage collection is estimated as $\alpha A + \beta N$, where A is the number of accessible cells, N is the total number of cells, and α and β are some positive factors. As the application program uses more cells, garbage collection takes longer and thus the application program is suspended in its execution for a longer time. As reported in Steele [14], for typical applications, garbage collection takes several seconds to several tens of seconds. It is difficult for an interactive or real-time list-processing system to provide adequate service when it suspends execution for such a long time. For example, if an application program were to control a robot in a product line of a car factory, the robot would stop periodically while the line keeps moving, thus yielding faulty cars.

In order to avoid these problems, several algorithms for "real-time" garbage collection have been proposed [1, 2, 5, 6, 9, 10, 13, 14]. A real-time garbage collector runs in parallel with the application program so that the time for each list-processing primitive is bounded by some small constant. Most of these real-time algorithms are intended for multiprocessor machines. The basic idea is to use one processor for garbage collection while another processor is responsible for the execution of the application program. Unfortunately, many list-processing systems are running on general-purpose machines and thus these algorithms cannot be applied to these systems. Here, *general-purpose machines* refer to Von Neumann style computers with a single processor, such as the MC68000 and the VAX. For these machines, no support is expected from the underlying hardware and from the firmware. Although the algorithms

Address correspondence to Taiichi Yuasa, Department of Information and Computer Science, Toyohashi University of Technology, Tempaka-Cho, Toyohashi, 440 Japan.

0164-1212/90/$3.50

could be simulated on general-purpose machines, the overall system efficiency would be reduced to a great extent. Nowadays, Lisp machines are available as commercial products, but there are many more Lisp systems (or list-processing systems in general) in use on general-purpose machines. It seems that many other Lisp systems will become available on general-purpose machines in the future.

On the other hand, Baker's real-time algorithm [1] is inherently serial and has been implemented on single-processor machines [12], However, this algorithm puts an extra burden on operations of pointer references (i.e., car and cdr in Lisp). Implementations of this algorithm, therefore, use special-purpose hardware [10] to achieve moderate performance. The overhead of implementing them on general-purpose machines is regarded extremely high [3], even with the support of the firmware [17].

This paper presents and discusses an algorithm for real-time garbage collection suitable for list-processing systems on general-purpose machines. This algorithm was designed with two principles kept in mind. A real-time garbage collector on general-purpose machines inevitably causes some overhead against the execution efficiency of the list-processing system. The primary reason is that some of the primitive list-processing operations must check the current status of garbage collection. Such a check is unnecessary in conventional list-processing systems with a stop garbage collector. In order to reduce the total overhead due to the real-time behavior of the garbage collector, our first design principle is that overhead on frequently used primitives should be small or even nonexistent. In particular, the presented algorithm was designed so that it does not put any overhead on the commonly used operations of pointer references, assignments, and stack manipulations, while it puts an extra burden on the rarely used destructive list operations such as *rplaca* and *rplacd* in Lisp.

The other design principle is that it should be easy to apply the algorithm to conventional systems with a stop garbage collector. Not all list-processing applications require real-time behavior, but rather the total execution time may be more important for many applications. Thus, we would like to have two versions of the same list-processing system, one with a stop garbage collector and the other with a real-time one, and to have the chance to select the best of them for each application.

In the next section, we present the basic idea of our real-time algorithm, first by introducing a simple list-processing system with a conventional stop garbage collector, and then by applying our algorithm to this system to make it a real-time system. The correctness proof of the algorithm is given in Section 3, and the dynamic behavior of the real-time system is discussed in Section 4. The algorithm is then applied to more realistic list-processing systems: to a system with stack mechanism in Section 5 and to a system with multiple kinds of cells in Section 6. Finally, in Section 7, we will show that the algorithm is easily applied to those systems with some sort of compactifying garbage collector.

Throughout this paper, we use a Pascal-like language as the implementation language of list-processing systems. Deviations from Pascal are as follows:

1. Type declarations are omitted if there is no fear of ambiguity. We make it a rule that variables x, y, z, and p are pointer variables and variables i and j are *integer* variables.
2. We allow underscores "_" in identifiers.
3. We use pointer arithmetic and pointer comparison, similar to those in the C language [8]. When a pointer p points to the ith element of an array, then the expression $p + 1$ represents a pointer to the $(i + 1)$th element. Similarly, comparisons on two pointers are defined in terms of the array index, on condition that both pointers point to elements of the same array. For instance, when p and q point to the ith and jth elements, respectively, of an array, then $p < q$ holds if and only if $i < j$.

2. THE ALGORITHM

In this section, we explain the basic idea of our real-time algorithm. We first present a simple list-processing system with a conventional stop garbage collector. This system is simple in that it supports only a single type of cell, *cons cells*, and that it does not have the stack mechanism for program execution.

A *cons cell*, or simply a *cell*, is a record object consisting of three fields: *mark*, *car*, and *cdr*.

```
type cell = record
              mark : Boolean;
              car : pointer;
              cdr : pointer;
            end
```

A pointer may be *nil*, or else it points to either a cell or an atom. Cells are allocated in the heap H, which is an array of $N + 1$ cells.

```
var H : array[0..N] of cell
```

The system actually uses $H[0]$ to $H[N-1]$ as the heap, and thus a maximum of N cells are available in the system. The last location of H ($H[N]$) is reserved so that the expression "$p + 1$" is meaningful whenever p points to a cell. We introduce two constant pointers *Hbtm* and *Htop*, which point to the first location of the heap (i.e., $H[0]$) and the last location of the heap (i.e.,

$H[N-1]$), respectively. Atoms are allocated somewhere not in the heap. To distinguish pointers to cells from those to atoms, we use the predicate *consp*. *consp*(p) is *true* if and only if the pointer p points to a cell.

The system has an array R consisting of NR pointers, where NR is a small constant. The application program can access and modify the contents of R by primitive operations *Lgetr* and *Lsetr*. *Lgetr*(i) returns the ith element of R and *Lsetr*(i, p) replaces the ith element of R with p. Pointers in R are called *root pointers*. Only those cells reachable directly or indirectly from the root pointers are regarded to be *accessible*. Other cells are inaccessible. The purpose of garbage collection is to arrange inaccessible cells so that they may be recycled for further use. It is the responsibility of the application program to prevent all cells in use from being garbage-collected unexpectedly. In particular, we assume that only accessible pointers can be passed as arguments to operations.

The primitive list-processing operations in this system are *Lcons*, *Lcar*, *Lcdr*, *Lrplaca*, *Lrplacd*, and *Leq*, each of which corresponds to a Lisp function in the obvious way. Unlike Lisp, *Lcons* does not return a value, but is a procedure that causes a side effect. *Lcons*(i, x, y) allocates a new cell with x and y in its *car* and *cdr* fields and, in addition, stores the cell pointer into $R[i]$. For example, in order to obtain a new list consisting of two *nils*, we write

Lcons(1, *nil*, *nil*);

Lcons(1, *nil*, *Lgetr*(1));

The list is then stored in $R[1]$.

Figure 1 illustrates an implementation of our simple list-processing system with a conventional stop garbage collector. The system is initialized by *init*() which is invoked once when the system begins to run. All available cells are linked through their *car* fields to form the freelist whose first element is pointed to by the global variable *free_list*. This initialization of the freelist is rather a convention to simplify the explanation. Available cells could be allocated directly from the heap until the first garbage collection occurs.

The garbage collector is invoked when the freelist is exhausted. During the mark phase, the garbage collector marks, i.e., turns to *true* the *mark* fields of, all accessible cells by recursively traversing list structures by the use of the garbage collection stack *gcs*. During the sweep phase, the heap is scanned sequentially so that all inaccessible cells, i.e., all non-marked cells, are collected into the freelist. Note the use of the stack operations *gcs_push* and *gcs_pop*. The body of the mark phase loop simply repeats the push and pop operations and the actual marking is done by *gcs_push*(x), which

```
var free_list : pointer;
procedure init();
  begin free_list := nil;
        for p := Hbtm to Htop do
          begin p^.mark := false;
                p^.car := free_list;
                free_list := p
          end;
        for i := 1 to NR do R[i] := nil
  end {of init};
procedure Lcons(i, x, y);
  begin if free_list = nil then
          begin gc();
                if free_list = nil then error("no storage")
          end;
        {cell allocation}
        p := free_list;
        free_list := p^.car;

        p^.car := x;
        p^.cdr := y;

        R[i] := p
  end {of Lcons};
function Lcar(x) : pointer; Lcar := x^.car;
function Lcdr(x) : pointer; Lcdr := x^.cdr;
procedure Lrplaca(x, y); x^.car := y;
procedure Lrplacd(x, y); x^.cdr := y;
function Leq(x, y); Boolean; Leq := x = y;
procedure Lsetr(i, x); R[i] := x;
function Lgetr(i): pointer; Lgetr := R[i];
procedure gc();
  begin {initialization}
        gcs_init();
        for i := 1 to NR do gcs_push(R[i]);
        {mark phase}
        while not gcs_empty() do
          begin p := gcs_pop();
                gcs_push(p^.car);
                gcs_push(p^.cdr)
          end;
        {sweep phase}
        for p := Hbtm to Htop do
          if p^.mark then
            p^.mark := false
          else begin p^.car := free_list;
                     free list := p
               end
          end {of gc};
{primitive operations on gcs}
var gcs : array[1..NGCS] of pointer;
var gcs_top : integer;
procedure gcs_init(); gcs_top := 1;
procedure gcs_push(x);
  if consp(x) and (not x^.mark) then
    begin x^.mark := true;
          gcs[gcs_top] := x;
          gcs_top := gcs_top + 1
    end;
function gcs_pop : pointer;
  begin gcs_top := gcs_top - 1;
        gcs_pop := gcs[gcs_top]
  end;
function gcs_empty : Boolean;
    gcs_empty := (gcs_top = 1);
```

Figure 1. The conventional system with a stop garbage collector.

```
procedure gc();
  begin {initialization}
        gcs_init();
        for i := 1 to NR do gcs_push(R[i]);

        {mark phase}
        while not gcs_empty() do
          begin p := gcs_pop();
                while consp(p) and (not p^.mark) do
                  begin p^.mark := true;
                        gcs_push(p^.cdr);
                        p := p^.car
                  end
          end;

        {sweep phase}
        for p := Hbtm to Htop do
          if p^.mark then p^.mark := false;
          else begin p^.car := free_list;
                     free_list := p
               end
  end {of gc};
```

Figure 2. An alternative definition of *gc*().

pushes x onto *gcs* only if x points to a non-marked cell. This mechanism may seem inefficient since a pointer may be poped from *gcs* immediately after it is pushed. However, a slight change will overcome this inefficiency as illustrated in Figure 2. Our intention is to keep the algorithm simple and clear.

Now we modify the above system so that it becomes a real-time system (see Figure 3).

In the real-time version, the garbage collection proceeds while the application program keeps running. Since we assume only a single processor, the two processes cannot proceed really in parallel. Instead, the

```
var free_count : integer;
type phases = (idling, mark_phase, sweep_phase);
var phase : phases;
var sweeper : pointer;

procedure init();
  begin free_list := nil;
        for p := Hbtm to Htop do
          begin p^.mark := false;
                p^.car := free_list;
                p^.cdr := leave_me;
                free_list := p
          end;
        free_count := N;
        phase := idling;
        sweeper := Htop + 1;
        for i := 1 to NR do R[i] := nil;
        gcs_init()
  end {of init};

procedure Lcons(i, x, y);
  begin {garbage collection dispatcher}
        if phase = mark_phase then
          begin mark();
                if gcs_empty() then phase := sweep_phase
          end
        elseif phase = sweep_phase then
          begin sweep();
                if sweeper > Htop then phase := idling
          end
        elseif free_count ≦ M then
          begin phase := mark_phase;
                sweeper := Hbtm;
                for i := 1 to NR do gcs_push(R[i])
          end;

        if free_count ≦ 0 then error("no storage");

        {cell allocation}
        p := free_list;
        free_list := p^.car;
        free_count := free_count - 1;

        p^.car := x;
        p^.cdr := y;
        p^.mark := (p ≧ sweeper);

        R[i] := p
  end {of Lcons};

procedure mark()
  begin i := 1;
        while i ≦ K1 and (not gcs_empty()) do
          begin p := gcs_pop();
                gcs_push(p^.car);
                gcs_push(p^.cdr);
                i := i + 1
          end
  end {of mark};

procedure sweep();
  begin i := 1;
        while i ≦ K2 and sweeper ≦ Htop do
          begin if sweeper^.mark then
                  sweeper^.mark := false;
                elseif not sweeper^.cdr = leave_me then
                  begin sweeper^.car := free_list;
                        sweeper^.cdr := leave_me;
                        free_list := sweeper;
                        free_count := free_count + 1
                  end;
                sweeper := sweeper + 1
                i := i + 1
          end
  end {of sweep};

procedure Lrplaca(x, y);
  begin if phase = mark_phase then gcs_push(x^.car);
        x^.car := y
  end;

procedure Lrplacd(x, y);
  begin if phase = mark_phase then gcs_push(x^.cdr);
        x^.cdr := y
  end;
```

Figure 3. The real-time system (operations not described here are same as in Figure 1).

whole process of garbage collection is divided into chunks, each of which can be executed in less than some constant time, and is executed only in certain situations. The situation we chose is when the application program requires a new cell, i.e., when *Lcons* is called. This idea is credited to Baker [1]. Since cells are requested while garbage collection is in progress, we cannot wait until the freelist is exhausted. Instead, a garbage collection begins when the number of the freelist cells becomes less than a certain number M. To keep the num-

ber of freelist cells, we introduce an *integer* variable *free_count*.

With the real-time system, however, the application program keeps requiring cells even during the sweep phase. If the freelist is set empty at the beginning of the sweep phase, as was in the stop garbage collector, then during the next call of *Lcons*, the sweeping steps must be repeated until a non-marked cell is found in the heap. Since there is no small upper bound for the time required to find a non-marked cell, that call of *Lcons* could take a long time. Thus, our real-time system should leave the freelist alone and collect only those non-marked cells that are not in the freelist yet. To determine whether a cell is in the freelist, we use the *cdr* fields of freelist cells, which are left unused in the stop collector. The rule of thumb is that a non-marked cell is in the freelist if and only if its *cdr* field has a distinguished pointer *leave_me*, which is not accessed by the application program.

The procedure *mark*( ) implements a single chunk of the mark phase. It marks $k1$ cells each time it is called, with $k1$ being a small constant. If there are less than $k1$ cells to mark, that is, if *gcs* becomes empty before the body of the **while** statement is repeated $k1$ times, then *mark*( ) returns immediately. Precisely speaking, therefore, *mark*( ) marks $k1$ cells as long as it is possible. Similarly, the procedure *sweep*( ) implements a chunk of the mark phase and takes care of $k2$ cells as long as it is possible. The **for** control variable p used in the sweep phase loop of Figure 1 is replaced by the global variable *sweeper* so that calls to *sweep*( ) can continually process the whole heap. The global variable *phase* keeps track of the current state of garbage collection. Its value is *mark_phase* during mark phase, *sweep_phase* during sweep phase, and *idling* otherwise.

The most important feature of our real-time system is that, during a sweep phase, it collects those and only those cells that are inaccessible and are not in the freelist at the beginning of the mark phase. Although some cells may become garbage during the garbage collection, they are not collected until the next garbage collection. This feature is realized by the following properties of the system.

1. All accessible cells at the beginning of a mark phase are eventually marked during the phase.
2. Newly allocated cells during a garbage collection are never collected during the sweep phase.

The **if** statement

if *phase* = *mark_phase* **then** *gcs_push*(x^.*car*);

in the definition of *Lrplaca* in Figure 3 and the similar **if** statement of *Lrplacd* are added to achieve the first property. If the list structures used in the application program are never changed during the mark phase, then repeated calls to *mark*( ) eventually mark all accessible cells. However, *Lrplaca* and *Lrplacd* modify the list structures and thus, without these **if** statements, there is the possibility that some accessible cells are left unmarked. This is mainly because *mark*( ) cannot mark a cell once the cell becomes nonreachable from (pointers on) *gcs*. For example, suppose that the following statements are executed immediately after a garbage collection begins.

Lsetr(2, *Lcar*(*Lgetr*(1)));

Lrplaca(*Lgetr*(1), *nil*);

Figure 4 illustrates the status of the relevant cells when *Lrplaca* is invoked. Assume that the cell α is the only cell that points to the cell β. If *Lrplaca* simply replaced the *car* field of α, then β would not be reachable from *gcs* any more and thus would loose the chance to get marked. As the result, β might be regarded as a garbage while it is still accessible. This situation cannot occur in our system since the pointer to β is pushed onto *gcs* by the statement *gcs_push*(x^.*car*) at the beginning of *Lrplaca*(x, y). In order to achieve the second property above, we added the statement

p^.*mark* := ($p \geqq$ *sweeper*);

into *Lcons*(i, x, y). During execution of the mark phase, this statement effectively marks all newly allocated cells. During execution of the sweep phase, this statement marks only those newly allocated cells that are not yet processed by *sweep*( ). Other cells allocated during the sweep phase need not be marked since *sweep*( ) processes each cell only once. Of course, these discussions do not entirely justify that the system has the above properties. The proof is left to the next section.

Note that the real-time system causes no execution overhead on the primitives *Lcar*, *Lcdr*, *Lsetr*, *Lgetr*, and *Leq*. As for *Lcons*, calls to *mark*( ) and *sweep*( ) can be expanded inline in the body of *Lcons*. Moreover, because $k1$ and $k2$ are constants, the loops in these procedures can be expanded into straight-line code. Thus, the essential overhead of *Lcons* will be relatively small. *Lrplaca* and *Lrplacd* suffer from the overhead, but these operations are used less frequently than other primitives in actual list-processing applications. They may be used internally for assignments to variables in those Lisp systems that implement variable bindings by association lists [11]. Even in such Lisp systems, local variables in compiled programs are usually allocated on the system stack and, as will be shown in Section 5, our algorithm causes no execution overhead on the operation to replace the contents of the system stack. In order to estimate the efficiency of our real-time system, we

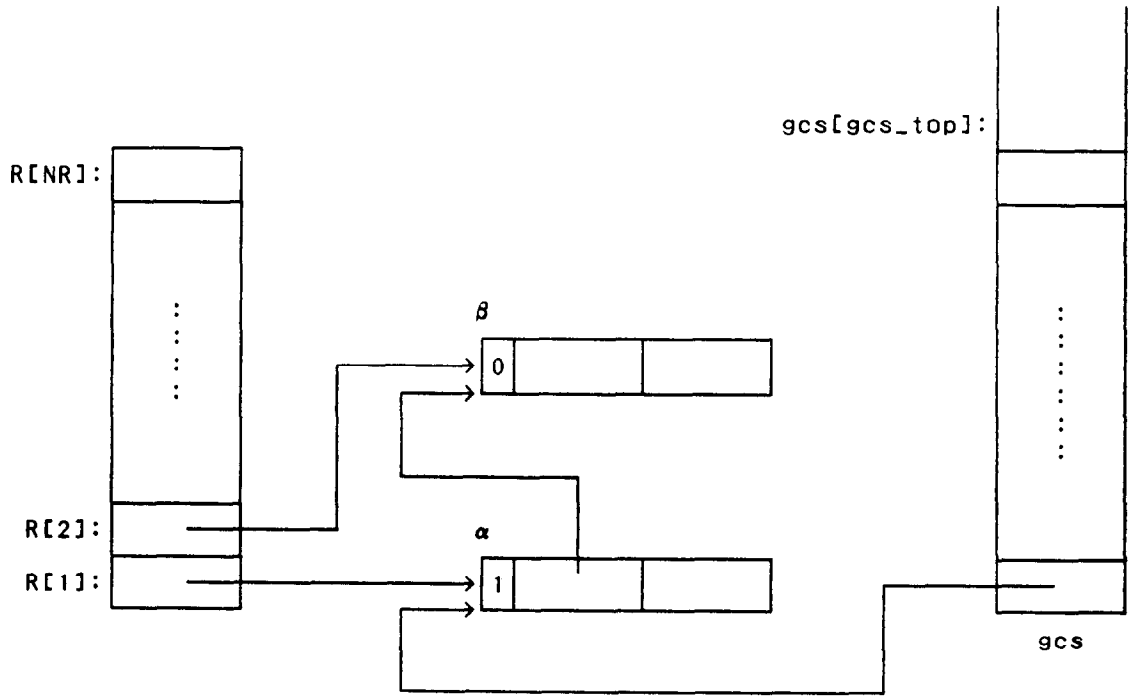

Figure 4. The status of cells (The arrows ' → ' represent pointer references. Truth values of *mark* fields are represented by 1 (for *true*) and 0 (for *false*).)

still need to analyze how frequently garbage collection occurs. This analysis is left to Section 4.

What is as important as execution efficiency is the size of primitive operations, since in many list-processing systems some primitive operations are expanded inline in the compiled code of application programs. Usually, calls to *Lcons* are not expanded inline because the body of *Lcons* is relatively large. Calls to *Lrplaca* and *Lrplacd* are rare in compiled programs. Since each of the other primitives is of the same size in the real-time system as in the conventional system, the real-time system promises that the size of compiled code is kept small.

Now we need to show the memory overhead of the real-time system. First of all, *gcs* needs no extra space. As for the size of the heap, we need to analyze the dynamic behavior of the system, which is the main issue in Section 4.

3. PROOF OF CORRECTNESS

Our major concern of this section is to prove that the following theorem holds in the real-time system presented in the previous section.

Theorem 3.1. All and only those cells that are not accessible and are not in the freelist at the beginning of a garbage collection are put into the freelist during the sweep phase.

For this purpose, we first postulate some *system invariants* which characterize our real-time system. Here, a *system invariant* (or invariants, for short) is a property of the system that holds between calls to primitive operations. Naturally, each invariant is proved by induction on calls to primitive operations. More precisely, an invariant is proved first by showing that the property holds immediately after the system is initialized by *init*( ), and then by showing that each primitive operation preserves the property, assuming that *all* invariants (including the invariant to prove) hold before the primitive is called. We also assume that pointer arguments to primitive operations are all accessible. Since the complete proof of each invariant is too long to fit this paper, we only give a rough sketch of the proof. However, the reader should keep it in mind that the proof is essentially by induction.

The first four invariants are simple properties concerned with *gcs* and *sweeper*.

Invariant 3.1. *gcs* is empty during idling and sweep phases.

Invariant 3.2. *gcs* consists only of pointers to marked cells.

Invariant 3.3. A single pointer appears on *gcs* at most once.

Invariant 3.4. The value of the variable *sweeper* is a cell pointer or is equal to *Htop*+1. Its value is *Htop*+1 during idling phase, and *Hbtm* during mark phase.

Proof. Invariant 3.1 holds because *gcs* is augmented only by *gcs_push* which is invoked only in mark phase and *gcs* is empty at the end of mark phase. Invariant 3.2 holds because *gcs_push* marks a cell before it pushes the cell pointer onto *gcs* and no marked cell is unmarked during mark phase. Invariant 3.3 holds because *gcs_push* pushes a cell pointer onto *gcs* only when the cell pointed to by the pointer is not marked yet, whereas pointers already on *gcs* point to marked cells (Invariant 3.2). Invariant 3.4 holds because *sweeper* is set to *Hbtm* at the beginning of mark phase, incremented in sweep phase, and is *Htop*+1 at the end of sweep phase.

Let us introduce some terms used in the next group of invariants. When *sweeper* points to the *j*th cell $H[j]$, the *i*th cell in the heap (i.e., $H[i]$) is said to be *above* the *sweeper* if $i \geqq j$. Otherwise, the cell is said to be *below* the *sweeper*. Note that, during the sweep phase, "below the *sweeper*" and "above the *sweeper*" mean "already swept" and "not yet swept," respectively. Also note that, by Invariant 3.4, every cell is below the *sweeper* during the idling phase, and every cell is above the *sweeper* during the mark phase.

Let us call a cell *m markable* if it is not marked and there is a path from *gcs* to *m* traversing only non-marked cells except the first cell, which is directly pointed to from *gcs* and therefore is marked. More precisely, *m* is markable if and only if there is a sequence of distinct cell pointers $p_0, p_1, \cdots, p_n$ $(n > 0)$ such that

1. p_0 is on *gcs*,
2. either $p_i\text{^}.car = p_{i+1}$ or $p_i\text{^}.cdr = p_{i+1}$ $(0 \leqq i < n)$,
3. $p_i\text{^}.mark = false$ $(0 < i \leqq n)$, and
4. p_n points to *m*.

The intention is that markable cells are eventually marked during the mark phase.

The following invariants are listed together because they depend on each other in that the proof of each invariant uses the other invariants as induction hypotheses.

Invariant 3.5. *leave_me* is inaccessible.

Invariant 3.6. *leave_me* is not reachable from *gcs*.

Invariant 3.7. A cell is a freelist cell if and only if its *cdr* field is *leave_me*.

Invariant 3.8. The freelist is loop-free.

Invariant 3.9. No cell below the *sweeper* is marked.

Invariant 3.10. Each accessible cell above the *sweeper* is either marked or markable.

Proof.

(*Invariant 3.5*). An inaccessible cell becomes accessible only when it is allocated by *Lcons*, but *Lcons* replaces the *car* and *cdr* fields of the allocated cell with accessible cells. The allocation operation, therefore, does not make *leave_me* accessible. On the other hand, *leave_me* may be assigned to the *cdr* part of a cell by *sweep*( ), but that cell is inaccessible (Invariant 3.10). Thus, *leave_me* never becomes accessible.

(*Invariant 3.6*). *gcs_push* does not make *leave_me* reachable from *gcs* because, when it is called from *mark*( ), it pushes pointers pointing only to already reachable cells, and, when called from elsewhere, it pushes pointers pointing only to accessible cells but *leave_me* is never accessible (Invariant 3.5). Reachability from *gcs* may be affected by the destructive operations of *Lrplaca* and *Lrplacd* but only accessible cells become reachable by these operations. Thus, *leave_me* never becomes reachable from *gcs*.

(*Invariant 3.7*). Invariant 3.8 makes sure that the allocation operation of *Lcons*(i, x, y) removes a cell from the freelist. By that operation, the *cdr* field of the removed cell is replaced by an accessible cell that is distinct from *leave_me*. Invariants 3.5–3.10 make sure that the freelist is augmented only when *sweep*( ) encounters a cell that is not in the freelist. In that event, *sweep*( ) replaces the *cdr* field of the cell with *leave_me*.

(*Invariant 3.8*). It is clear from Invariant 3.7 that the operation of *sweep*( ) to augment the freelist does not cause circularity in the freelist.

(*Invariant 3.9*). The system never turns *mark* fields of cells below the *sweeper* to *true*: Whenever *gcs_push* is called, *sweeper* = *Hbtm* and thus no cell is below the *sweeper*. When *sweeper* is incremented by *sweep*( ), the *mark* field of the cell pointed to by *sweeper* is turned *false* if it is *true* before.

(*Invariant 3.10*). Invariant 3.10 is proved in a way similar to the proof of Theorem 3.1 below and is left to the reader.

Now we are ready to prove Theorem 3.1. Figure 5 illustrates the transfer of state during a garbage collection. Each arrow represents a call to a primitive operation and each box represents a state. The garbage collection is triggered by a call to *Lcons* in state A, which transfers the state to B. In state B, primitive operations are called a certain number of times, and finally, a call to *Lcons* transfers the state to C and starts the sweep phase. Similarly, primitive operations are called a certain number of times in state C. Finally, a call to *Lcons* transfers the state to D and this ends the garbage collection. Let us call those cells that are either accessible or in the freelist in state A *red* cells, and let us call the other cells *black* cells. What we have to prove is that all and only black cells are added to the freelist during the sweep phase. To do this, we postulate two loop invariants.

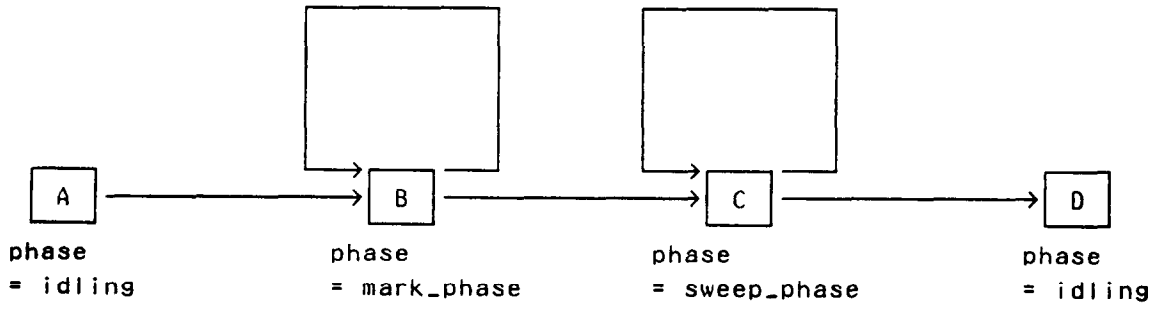

Figure 5. Transfer of the state during garbage collection.

Loop Invariant B. In state B, a cell is red if and only if it is marked, markable, or in the freelist.

Loop Invariant C. In state C, a cell above the *sweeper* is red if and only if it is marked or in the freelist.

By the definition of *sweep*( ), it adds a cell to the freelist if and only if it encounters a cell m that satisfies the following conditions.

1. m is above the *sweeper*.
2. m is not marked.
3. m is not in the freelist.

Loop Invariant C says that m is black, and we conclude that *sweep*( ) eventually puts all and only black cells to the freelist. What remains to do for the proof of Theorem 3.1, therefore, is to prove the two loop invariants.

Proof of Loop Invariant B. The proof is split into two cases.

A → B: When *Lcons* transfers the state from A to B, it pushes root pointers onto *gcs*. Those cells directly pointed to by root pointers are marked, the other accessible cells become markable, and no other cells are marked nor markable, since no cell was marked before (Invariants 3.9 and 3.4). *Lcons* then removes one cell from the freelist and marks it. Thus, Loop Invariant B holds after the cell of *Lcons*.

B → B: We need only to show that *Lcons*, *Lrplaca*, and *Lrplacd* preserve Loop Invariant B, since the other primitive operations do not modify list structures, mark fields, and *gcs*. Let us begin with *Lcons*. Consider the loop body

p := *gcs_pop*( );

gcs_push($p^{\wedge}$.*car*);

gcs_push($p^{\wedge}$.*cdr*);

of *mark*( ). Let m be an arbitrary cell that is markable before execution of the body, and let $q_0, q_1, \cdots, q_n$ $(n > 0)$ be a sequence of distinct cell pointers such that

1. q_0 is on *gcs*,
2. either $q_i{}^{\wedge}.car = q_{i+1}$ or $q_i{}^{\wedge}.cdr = q_{i+1}$ $(0 \leqq i < n)$,
3. $q_i{}^{\wedge}.mark = false$ $(0 < i \leqq n)$, and
4. q_n points to m.

We will show that m remains markable or is marked after execution of the loop body.

Case 1: $p^{\wedge}.car, p^{\wedge}.cdr \neq q_1, \cdots, q_n$. The loop body preserves the above properties and thus m remains markable.

Case 2: $p^{\wedge}.car = q_h$ for some h $(0 < h \leqq n)$ but $p^{\wedge}.cdr \neq q_1, \cdots, q_n$ or, conversely, $p^{\wedge}.cdr = q_h$ for some h $(0 < h \leqq n)$ but $p^{\wedge}.car \neq q_1, \cdots, q_n$. After the execution of the loop body,

1. q_h is on *gcs*,
2. either $q_i{}^{\wedge}.car = q_{i+1}$ or $q_i{}^{\wedge}.cdr = q_{i+1}$ $(h \leqq i < n)$,
3. $q_h{}^{\wedge}.mark = true$,
4. $q_i{}^{\wedge}.mark = false$ $(h < i \leqq n)$, and
5. q_n points to m.

hold. That is, m remains markable if $h = n$ and m gets marked otherwise.

Case 3: $p^{\wedge}.car = q_{h1}$ and $p^{\wedge}.cdr = q_{h2}$ for some $h1$ and $h2$ $(0 < h1 \leqq n, 0 < h2 \leqq n)$. Let $h = \max(h1, h2)$. Then the five properties of $q_h, \cdots, q_n$ given in case 2 hold and thus, after the execution of the loop body, either m remains markable or m is marked.

Since no cell is unmarked during the mark phase and the loop body of *mark*( ) does not affect the freelist, we conclude that *mark* preserves Loop Invariant B, and that *Lcons* preserves it.

Similarly, to show that *Lrplaca* preserves Loop Invariant B, it suffices to show that, for each markable cell m, m remains markable or is marked after a call to *Lrplaca*. Let $q_0, q_1, \cdots, q_n$ $(n > 0)$ be a sequence of distinct cell pointers as above.

Case 1: $x^{\wedge}.car \neq q_0, q_1, \cdots, q_n$. *Lrplaca* preserves the four properties of this pointer sequence and therefore m remains markable.

Case 2: $x\hat{}.car = q_0$. *gcs_push*($x\hat{}.car$) does nothing because $x\hat{}.car\hat{}.mark = q_0\hat{}.mark = true$ (Invariant 3.2).

Case 3: $x\hat{}.car = q_n$. Since $x\hat{}.car\hat{}$ points to m, *gcs_push*($x\hat{}.car$) turns the *mark* field of m *true*. Thus, m gets marked after *Lrplaca*.

Case 4: $x\hat{}.car = q_j$ for some j ($0 < j < n$). After the call of *gcs_push*($x\hat{}.car$),

1. q_j is on *gcs*,
2. either $q_i\hat{}.car = q_{i+1}$ or $q_i\hat{}.cdr = q_{i+1}$ ($j \leqq i < n$),
3. $q_i\hat{}.mark = false$ ($j < i \leqq n$), and
4. q_n points to m

hold. Thus, m remains markable after *gcs_push*($x\hat{}.car$). If x is distinct from any q_i ($j \leqq i < n$), then m obviously remains markable after the statement "$x\hat{}.car := y$". Suppose $q_h = x$ for some h ($j \leqq h < n$). If $q_h\hat{}.cdr$ is distinct from q_{h+1}, then $q_h\hat{}.car = q_{h+1}$, i.e., $x\hat{}.car = q_{h+1}$. Thus, $q_j = q_{h+1}$ ($=x\hat{}.car$) for $0 < j < h + 1 < n$, but this contradicts the assumption that $q_0, q_1, \cdots, q_n$ are distinct. Therefore, $q_h\hat{}.cdr = q_{h+1}$. This means that the above properties of the sequence $q_j, \cdots, q_n$ hold and thus m remains markable, even after the statement "$x\hat{}.car := y$".

The similar discussion applies to *Lrplacd* and we conclude that the primitive operations preserve Loop Invariant B.

Proof of Loop Invariant C. The proof is also split into two cases.

B → C: The discussion in the proof of Loop Invariant B above applies also to the final call to *Lcons* in state B, and thus Loop Invariant B holds when the state becomes C. Since Loop Invariant B implies Loop Invariant C, Loop Invariant C holds when the state becomes C.

C → C: We need only to show that *Lcons* preserves Loop Invariant C, since the other primitive operations, when called in state C, do not affect the freelist, mark fields of cells, and the value of *sweeper* during the sweep phase. The allocation operation of *Lcons* preserves Loop Invariant C because it removes one cell from the freelist but marks it if it is above the *sweeper*. Also, *sweep*( ) preserves Loop Invariant C because none of those cells put into the freelist and none of the unmarked cells are above the *sweeper* after the call of *sweeper*( ). Therefore, we conclude that *Lcons* does preserve Loop Invariant C.

4. THE DYNAMIC BEHAVIOR

In this section, we analyze the dynamic behavior of our real-time system. In particular, we are interested in the status of the freelist during the execution of a given application program. From the analysis, we postulate a sufficient condition on the system parameters N, M, $k1$, and $k2$ to avoid *starvation*, i.e., the situation that the freelist becomes empty while the application program requires more cells. We then use this condition to estimate the memory overhead of our real-time system. Also in this section, we estimate the number of times the garbage collector is invoked.

In order to measure the course of computation, we use the number of times that *Lcons* is invoked: "at time t" means "at the tth call to *Lcons*." Given an application program, let $F(t)$ and $A(t)$ be the number of freelist cells and the number of accessible cells, respectively, at time t. Note that $A(t)$ depends on the program, but not on the system parameters N, M, $k1$, and $k2$.

Let us trace $F(t)$. Clearly, $F(1) = N$ since every cell is in the freelist initially. Then, during the idling phase, one cell is removed from the freelist each time *Lcons* is called, but no cell is added into the freelist. Thus, $F(t) = F(t-1) - 1$. When the number of the freelist cells becomes equal to M, the first garbage collection begins. Suppose this happens at time $t = a$. Then

$$F(a) = M$$

Suppose that, during the first garbage collection, *phase* is turned from *mark_phase* to *sweep_phase* at time $t = b$ and the garbage collection ends at time $t = c$. During the mark phase, *gcs_pop*( ) is called $k1$ times each time *Lcons* is called. The only exception is the last call of *Lcons* which may call *gcs_pop*( ) less than $k1$ times. As discussed in the previous section, cell pointers that are accessible at $t = a$ are pushed onto *gcs* exactly once, but pointers to other cells are never pushed onto *gcs*. Therefore, *gcs_pop*( ) is called totally $A(a)$ times during the mark phase. Thus,

$$b - a = \lceil A(a)/k1 \rceil$$

During the sweep phase, the value of *sweeper* is incremented by $k2$ each time *Lcons* is called. The only exception is the last call of *Lcons* which may increment *sweeper* by less than $k2$. Since *sweeper* is totally incremented by N,

$$c - b = \lceil N/k2 \rceil$$

To simplify the calculation, let us assume that N is a multiple of $k2$.

$$c - b = N/k2$$

Since no cell is added into the freelist during the mark phase,

$$F(t) = F(a) - (t - a), \qquad \text{for } a \leqq t \leqq b$$

On the other hand, the value of $F(t)$ during the sweep

phase depends on the distribution of the black cells (i.e., those cells that are not accessible nor in the freelist at $t = a$) over the heap. For $i = 0, 1, \cdots, N/k2$, let $B(i)$ be the number of black cells among the first $i * k2$ cells $H[0], \cdots, H[i * k2 - 1]$ in the heap. $B(i)$ gives the number of black cells that are put into the freelist by the first i calls of *sweep*( ). The first call to *sweep*( ) occurs at time $t = b + 1$ and thus, at time t ($b + 1 \leqq t \leqq c + 1$), $B(t - b - 1)$ black cells have been put into the freelist. Since one cell is removed from the freelist by each call of *Lcons*, we obtain

$$\begin{aligned} F(t) &= F(b) + B(t - b - 1) - (t - b), \\ &= F(a) + B(t - b - 1) - (t - a), \\ &\quad \text{for } b + 1 \leqq t \leqq c + 1 \end{aligned}$$

Thus, $B(i)$, together with $F(a)$ (=M), N, $k1$, and $k2$, completely defines $F(t)$ for $a \leqq t \leqq c + 1$. Although the function $F(t)$ thus defined may possibly have negative values, the system causes the "*no storage*" error when the number of freelist cells (i.e., the value of *free_count*) is going to be negative. In order for the first garbage collection to proceed successfully, it is necessary and sufficient that $F(t) \geqq 0$ for all t ($a \leqq t \leqq c + 1$).

The distribution function $B(i)$ can be an arbitrary function that satisfies the following conditions.

1. $B(0) = 0$
2. $B(i - 1) \leqq B(i) \leqq B(i - 1) + k2$, for $i = 1, 2, \cdots, N/k2$
3. $B(N/k2) = N - A(a) - F(a)$ (={number of black cells})

Since

$$\begin{aligned} B(i) &= B(N/k2) \\ &\quad - (B(N/k2) - B(N/k2 - 1)) \\ &\quad \cdots \\ &\quad - (B(i + 1) - B(i)) \\ &\geqq N - A(a) - F(a) - (N/k2 - i) * k2 \\ &= i * k2 - A(a) - F(a) \end{aligned}$$

and since $B(i)$ is nonnegative,

$$B(i) \geqq \begin{cases} 0, & 1 \leqq i \leqq d \\ i * k2 - A(a) - F(a), & d + 1 \leqq i \leqq N/k2 \end{cases}$$

where $d = \lfloor (A(a) + F(a))/k2 \rfloor$. From this, we obtain the lower bound $F_{lb}(t)$ of $F(t)$.

$$F_{lb}(t) = \begin{cases} F(a) - (t - a), \\ \qquad a \leqq t \leqq b + d + 1 \\ F(a) + ((t - b - 1) * k2 - A(a) - F(a)) \\ \quad -(t - a), \\ \qquad b + d + 2 \leqq t \leqq c + 1 \end{cases}$$

Note that in the extreme case that all black cells are located at the higher part of the heap, $F(t)$ is identical to $F_{lb}(t)$. This means that $F_{lb}(t)$ is the best possible lower bound of $F(t)$. $F_{lb}(t)$ decreases monotonically when $a \leqq t \leqq b + d + 1$, but increases monotonically when $b + d + 2 \leqq t \leqq c + 1$. A simple calculation says that $F_{lb}(b + d + 1) < F_{lb}(b + d + 2)$. Thus, $F_{lb}(t)$ takes the smallest value when $t = b + d + 1$, and a sufficient condition for $F(t) \geqq 0$ ($a \geqq t \geqq c + 1$) is

$$F(a) - (b + \lfloor (A(a) - F(a))/k2 \rfloor + 1 - a) \geqq 0 \quad (4.1)$$

which is equivalent to

$$F(a) \geqq (A(a) * (1/k1 + 1/k2) + 1)/(1 - 1/k2)$$

Usually, it is difficult to estimate the value of $A(a)$ for a given program, but the maximum number of accessible cells A_{max} is usually relatively easy to estimate. By using A_{max}, we obtain the following sufficient condition for $F(t) \geqq 0$.

$$M \geqq (A_{max} * (1/k1 + 1/k2) + 1)/(1 - 1/k2) \quad (4.2)$$

The above discussion applies also for the second garbage collection if $F(t) = M$ at the beginning of the second garbage collection. This condition is satisfied if $F(t) \geqq M$ immediately after the first garbage collection, i.e., if $F(c + 1) \geqq M$. Since

$$\begin{aligned} F(c + 1) &= F(a) + B(c - b) - (c - a + 1) \\ &= F(a) + (N - A(a) - F(a)) - \lceil A(a)/k1 \rceil \\ &\quad - N/k2 - 1 \\ &\geqq N - A_{max} - \lceil A_{max}/k1 \rceil - N/k2 - 1 \\ &> N * (1 - 1/k2) - A_{max} * (1 - 1/k1) - 2 \end{aligned}$$

a sufficient condition for $F(c + 1) \geqq M$ is

$$N * (1 - 1/k2) - A_{max} * (1 + 1/k1) - 2 \geqq M \quad (4.3)$$

The same discussion holds for successive garbage collections. Therefore, we conclude the following.

Theorem 4.1. Given an application program, if our real-time system satisfies both (4.2) and (4.3), then all garbage collection proceeds successfully.

Theorem 4.1 is useful to find appropriate values of M and N for a given program. Practically, we can ignore "+1" in (4.2) and "−2" in (4.3), since N, M, and A_{max} are much larger and, moreover, (4.2) and (4.3) are derived from the worst-case analysis. The practically safe values are, therefore,

$$M = A_{max} * (1/k1 + 1/k2)/(1 - 1/k2)$$

$$N = A_{max} * (1 + 2/k1 - 1/(k1 * k2))/(1 - 1/k2)^2$$

For instance, if $k1 = k2 = 20$, then $M = 0.105A_{max}$ and $N = 1.216A_{max}$. In comparison, for the conventional system with a stop garbage collector in Figure 1, the smallest safe value for N is A_{max}. In this case, therefore, the real-time system needs 21.6% more memory for the heap.

Now let us estimate the number of times the garbage collector is invoked, assuming that N and M satisfy both (4.2) and (4.3). Suppose that the ith garbage collection begins at time $t = a_i$ and it ends at $t = c_i$. As already shown,

$$c_i = \lceil A(a_i)/k1 \rceil + N/k2 + a_i$$

The number of freelist cells after the ith garbage collection is

$$F(c_i + 1) = N * (1 - 1/k2) - A(a_i) - \lceil A(a_i)/k1 \rceil - 1$$

Then, the system is in idling phase until $t = a_{i+1}$ and thus,

$$F(t) = F(c_i + 1) - (t - c_i - 1),$$

$$\text{for } c_i + 1 \geqq t \geqq a_{i+1}$$

Since $F(a_{i+1}) = M$,

$$F(c_i + 1) - (a_{i+1} - c_i - 1) = M$$

From this, we obtain

$$a_{i+1} - a_i = N - A(a_i) - M, \qquad \text{for } i = 1, 2, \cdots$$

This formula, together with the initial value of $a_1 = N - M + 1$, defines the sequence $\{a_1, a_2, \cdots\}$. If we assume that $A(t)$ is identical to a constant A_{mean}, then

$$a_i = i * (N - A_{mean} - M) + A_{mean} + 1$$

and we obtain a very rough estimation of the number of garbage collections as

$$T/(N - A_{mean} - M)$$

where T is the number of times *Lcons* is called during the execution of the given program. For the conventional system with a stop garbage collector given in Figure 1, the sequence $\{a_1, a_2, \cdots\}$ is defined by

1. $a_1 = N + 1$
2. $a_{i+1} - a_i = N - A(a_i), \qquad \text{for } i = 1, 2 \cdots$

Again, under the assumption $A(t) = A_{mean}$, the number of garbage collections is estimated as

$$T/(N - A_{mean})$$

This expression supports the widely believed rule that the larger N is, the fewer times the garbage collector is invoked. Although this rule does not apply in some cases (indeed, it is not difficult to find a counter example), this rule seems to apply in most cases. Similarly, the rough estimate for the real-time system above suggests that the smaller M is and the larger N is, the fewer times the garbage collector is invoked.

We have already seen that the safe values for N and M are, respectively, $1.216A_{max}$ and $0.105A_{max}$, in case $k1 = k2 = 20$. If we assume $A_{max} = 2A_{mean}$, then for these values of N and M, the number of garbage collection is about $0.82T/A_{mean}$ for the real-time system. In contrast, with the safe value of $N = A_{max}$ for the conventional system, the number of garbage collection is about T/A_{mean}. Thus, with these safe values of N and M, the real-time system causes *less* garbage collection than the conventional system. On the other hand, it is clear that, with the same size of heap, the real-time system causes more garbage collection than the conventional system. For instance, with the heap size $N = 1.216A_{max}$, the conventional system calls the garbage collector about $0.7T/A_{mean}$ times. Thus, the real-time system causes 17% more garbage collection than the conventional system.

5. SYSTEM STACK

In this section, we extend our real-time system so that the application program can handle the *system stack*. The system stack contains pointers and is typically used for argument passing and variable allocation. Just like the root array R, the system stack consists of root pointers. Unlike R, however, the maximum size of the system stack NSS is assumed much larger than the size of R and thus the system cannot take care of the pointers on the system stack all at once when a garbage collection begins, as will be discussed below. The primitive operations on the system stack are *ss_empty*(), *ss_push*(x), and *ss_pop*(). *ss_empty*() returns *true* if the system stack is empty, and returns *false* otherwise. *ss_push*(x) pushes the pointer x onto the system stack. *ss_pop*() pops up the system stack and returns the pointer previously at the top of the system stack.

```
procedure ss_push(x);
  begin SStop := SStop + 1;
        R[SStop] := x
  end;

function sspop : pointer;
  begin ss_pop := R[SStop];
        SStop := SStop - 1
  end;

function ss_empty : Boolean; ss_empty := (SStop = NR);
```

Figure 6. Primitive operations on the system stack.

To simplify the discussions on the system stack, we simply expand R so that it can contain up to $NR+NSS$ pointers.

var R : **array**[1..$(NR + NSS)$] **of** *pointer*;

By this convention, *Lsetr*(i, x) and *Lgetr*(i) are used also to access the system stack. A new variable *SStop* keeps the index of the top of the system stack within R. Initially, *SStop* is set to NR. The three primitive operations on the system stack is defined as in Figure 6. Now, the pointers $R[1], \cdots, R[SStop]$ are the only root pointers and those and only those cells that are reachable directly or indirectly from these root pointers are accessible.

For this model, the algorithm presented in Figure 3 correctly works, if we rewrite the **for** loop to initialize garbage collection as follows.

for $i := 1$ **to** *SStop* **do** *gcs_push*($R[i]$)

However, if *SStop* is relatively large, then the execution of *Lcons* will take a long time when phase is switched from *idling* to *mark_phase*. This violates the real-time property of the system. Instead of processing the root pointers all at once, our revised system processes at most a fixed number of root pointers each time *Lcons* is called (see Figure 7). Since the contents of the system stack will change as computation proceeds, we need to save the contents of the system of the stack when a garbage collection begins. Or else, we cannot make sure that all accessible cells at the beginning of garbage collection are eventually marked during the mark phase. For this purpose, we introduce another stack, called the *save stack*, which is implemented by an array SV and a global variable *SVtop*.

var SV : **array**[1..$(NR + NSS)$] **of** *pointer*;

var *SVtop* : *integer*;

When *Lcons* is called in idling phase, if the length of the freelist becomes too short, then all pointers in the system stack are copied into SV by *copy_system_stack*(*SStop*) and the value of *SStop* is saved into *SVtop*. Then, during the mark phase, *Lcons* processes at most *k3* pointers on the save stack each time *gcs* becomes empty after the call of *mark*( ). Here, $k3$ is a small constant, like $k1$ and $k2$. The copying operation *copy_system_stack*(*SStop*) can be directly implemented by the underlying hardware, using the so-called block transfer mechanism which almost all general-purpose machines support. Since the size of the system stack is at most $NR+NSS$ and since *NSS* is bounded by tens of kilobytes in most list-processing systems, we can assume that the copying operation takes only a short time.

```
procedure Lcons(i, x, y);
  begin { garbage collection dispatcher }
        if phase = mark_phase then
          begin mark();
                if gcs_empty() then
                  if SVtop > 0
                    { processing the save stack }
                    then for i := SVtop downto max(SVtop-k3, 1)
                           do gcs_push(SV[i])
                    else phase := sweep_phase
          end
        elseif phase = sweep_phase then
          begin sweep();
                if sweeper > Htop then phase := idling
          end
        elseif free_count ≦ M then
          begin phase := mark_phase;
                sweeper := Hbtm;
                { save contents of system stack }
                copy_system_stack(SStop);
                SVtop := SStop
          end;

        if free_count ≦ 0 then error("no storage");

        { cell allocation }
        p := free_list;
        free_list := p^.car;
        free_count := free_count - 1;

        p^.car := x;
        p^.cdr := y;
        p^.mark := (p ≧ sweeper);

        R[i] := p
  end { of Lcons };
```

Figure 7. *Lcons* of the real-time system with the system stack.

Primitive operations other than *Lcons* are the same as those in the real-time system without the system stack. In particular, the revised real-time system causes no extra burden on the most frequently used operations *Lcar* and *Lcdr*. As already seen in Section 3, by expanding *mark*( ) and *sweep*( ) inline and by expanding the loops in these procedures into straight-line code, the essential overhead on *Lcons* is relatively small. In addition, since $k3$ is a constant, the **for** loop to process the save stack can be also expanded into straight-line code. Moreover, the revised system causes no execution overhead on the stack operations including direct access to the stack entities by *Lsetr* and *Lgetr*.

The correctness discussions in Section 3 apply, with

minor changes, to the revised system. First of all, we add an invariant.

Invariant 5.1. The save stack is empty during idling and sweep phases.

Then, we redefine the notion of markable so that, in addition to *gcs*, the save stack can be regarded as the origin of markable cells. That is, a cell m is *markable* if and only if there is a sequence of distinct cell pointers $q_0, q_1, \cdots, q_n$ $(n > 0)$ such that

1. q_0 is either on *gcs* or on the save stack,
2. either $q_i\text{^}.car = q_{i+1}$ or $q_i\text{^}.cdr = q_{i+1}$ $(0 \geqq i < n)$,
3. $q_i\text{^}.mark = false$ $(0 < i \geqq n)$, and
4. q_n points to m.

Another change is to replace Invariant 3.2 with a stronger condition.

Invariant 5.2. *leave_me* is reachable neither from *gcs* nor from the save stack.

As for the analysis in Section 4, the mark phase may take more time than $\lceil A(a)/k1 \rceil$, since not only the last but also other calls to *mark*() may invoke *gcs_pop*() less than $k1$ times. Let *SStop0* be the value of *SStop* at $t = a$. Assume that, during the mark phase, *gcs* becomes empty (and thus some pointers on the save stack are processed) at $t = d_1, \cdots, d_{n+1}$ $(a < d_1 < \cdots < d_n < d_{n+1} = b)$. Clearly, $n = \lceil SStop0/k3 \rceil$. Also assume that, after $t = d_i$ $(i = 1, 2, \cdots, n)$, *gcs_pop*() is called e_i times until the save stack is processed next time. Then we have

$$d_{i+1} = \max(\lceil e_i/k1 \rceil, 1) + d_i, \qquad \text{for } i = 1, 2, \cdots, n$$

When *phase* is turned to *mark_phase* at $t = a$, *gcs* is empty. Thus, the next time *Lcons* is called, the save stack is certainly processed. Therefore, $d_1 = a + 1$. Let q_i and r_i be, respectively, the quotient and the remainder of e_i divided by $k1$. Then, the time for the mark phase is calculated as follows.

$$\begin{aligned}
&b - a \\
&= (d_1 - a) + \Sigma\{i \mid 1 \leqq i \leqq n\}[d_{i+1} - d_i] \\
&= 1 + \Sigma\{i \mid 1 \leqq i \leqq n\}[\max(\lceil e_i/k1 \rceil, 1)] \\
&= 1 + \Sigma\{i \mid r_i \neq 0\}[q_i + 1] + \Sigma\{i \mid r_i = 0 \,\&\, q_i \neq 0\}[q_i] \\
&\quad + \Sigma\{i \mid r_i = q_i = 0\}[1] \\
&= 1 + \Sigma\{i \mid 1 \leqq i \leqq n\}[q_i] + \Sigma\{i \mid r_i \neq 0\}[1] \\
&\quad + \Sigma\{i \mid r_i = q_i = 0\}[1] \\
&= 1 + \Sigma\{i \mid 1 \leqq i \leqq n\}[(e_i + r_i)/k1] + n \\
&\quad - \Sigma\{i \mid r_i = 0 \,\&\, q_i \neq 0\}[1] \\
&\leqq 1 + \lfloor A(a)/k1 \rfloor + \lceil SStop0/k3 \rceil \\
&\leqq 1 + \lfloor A(a)/k1 \rfloor + (NR + NSS)/k3
\end{aligned}$$

$(\Sigma\{i \mid P(i)\}[f(i)]$ means the sum of $f(i)$ for all integer i that satisfies $P(i)$.) Here, to simplify the calculation, we have assumed that $(NR + NSS)/k3$ is an integer. On the other hand, the time for the sweep phase (i.e., $c - b$) is same as in Section 4, and (4.1) is still sufficient for $F(t) \geqq 0$ $(a \geqq t \geqq c + 1)$. By replacing "$b - a$" in (4.1) with the above upper bound, we obtain a sufficient condition for (4.1).

$$M \geqq (A_{\max} * (1/k1 + 1/k2) + (NR + NSS)/k3)/(1 - 1/k2) \tag{5.1}$$

Since

$$\begin{aligned}
F(c+1) &= F(a) + (N - A(a) - F(a)) \\
&\quad - (b - a) - N/k2 - 1 \geqq N * (1 - 1/k2) \\
&\quad - A_{\max} * (1 + 1/k1) \\
&\quad - (NR + NSS)/k3 - 1
\end{aligned}$$

the sufficient condition for $F(c + 1) \geqq M$ is

$$N * (1 - 1/k2) - A_{\max} * (1 + 1/k1) - (NR + NSS)/k3 - 1 \geqq M \tag{5.2}$$

Theorem 5.1. Given an application program for our real-time system with the system stack, if both (5.1) and (5.2) hold, then all garbage collection proceeds successfully.

Let us ignore "-1" in (5.2). Then the practically safe value of N is

$$A_{\max} * (1 + 1/k1 - 1/(k1 * k2))/(1 - 1/k2)^2$$

plus the constant

$$(2 * k2 - 1) * (NR + NSS)/(k3 * (k2 - 1))$$

In case $k1 = k2 = k3 = 20$, the real-time system with the system stack needs $1.216A_{\max} + 0.102(NR + NSS)$ cells in the heap. In addition, the system needs the space for the save stack which should contain up to $NR + NSS$ cells. Thus, the real-time system needs $0.216A_{\max} + 1.102(NR + NSS)$ more space than the conventional system with the system stack.

6. MULTIPLE KINDS OF CELLS

So far, we have assumed that only a single type of cells (i.e., cons cells) are available. In this section, we extend our real-time system so that it supports other kinds of cells as well, such as symbol cells in Lisp. Usually, cells of a same type occupy a fixed size of memory and, therefore, if freelists are used to maintain available cells, each cell type α has its own freelist α*free_list*. The system keeps track of the maximum number of allocatable

cells N_α for each type α. A pointer can points to a cell of any type and, given a pointer, the system can determine the type of the cell pointed to by the pointer. In order to simplify our discussion, we assume that cells have two common fields *type* and *mark*: The *type* field determines the type of the cell, and *mark* is used by the garbage collector as before. The other fields contain pointers, and the number of these pointer fields is fixed for each type. For each type α, let $f_1, \cdots, f_{n_\alpha}$ be the names of the pointer fields. Then we have the following primitive operations on each type α.

1. The consing procedure $L\alpha cons(i, x_1, \cdots, x_{n_\alpha})$, which allocates an α cell from α*free_list*, assigns x_j to the f_j field of the cell, and sets the pointer to the cell into $R[i]$.
2. The retrieval functions $L\alpha f_1(x), \cdots, L\alpha f_{n_\alpha}(x)$, similar to $Lcar(x)$. Each $L\alpha f_j(x)$ receives a pointer to an α cell and returns the pointer in the f_j field of the cell.
3. The update procedures $L\alpha rplacf_1(x, y), \cdots, L\alpha rplacf_{n_\alpha}(x, y)$, similar to $Lrplaca(x, y)$. Each $L\alpha rplacf_j(x, y)$ receives a pointer x to an α cell and replaces the f_j field of the cell with y.

As in Section 2, the system initialization procedure *init*( ) prepares freelists so that each α*free_list* consists of N_α cells of type α. Without loss of generality, we can assume that cells in the freelist for type α are linked through their f_1 fields.

For this model with multiple cell types, our real-time system in Figure 3 is extended as follows. Each consing procedure $L\alpha cons(i, x_1, \cdots, x_{n_\alpha})$ begins with the same garbage collection dispatcher as that in *Lcons* in Figure 3, except that the dispatcher in $L\alpha cons(i, x_1, \cdots, x_{n_\alpha})$ uses an α-specific number M_α (see Figure 8). That is, a garbage collection starts when the size of α*free_list* becomes less than or equal to M_α for some type α. The rest of each consing procedure is similar to that of *Lcons*. Like *Lcar*, each retrieval function $L\alpha f_j(x)$ simply returns the value of $x.\hat{}f_j$. Like *Lrplaca*, each update procedure $L\alpha rplacf_j(x, y)$ checks the current phase before replacing $x\hat{}.f_j$ with y. If the system is currently in a mark phase, then it executes *gcs_push*($x\hat{}.f_j$).

```
procedure Lαrplacf_j(x, y);
  begin if phase = mark_phase then gcs_push(x^.f_j);
        x^.f_j := y
  end;
```

The loop body of *mark*( ) must be modified so that it pushes all of the pointers in the pointer fields. The procedure *sweep*( ), when it encounters a non-marked α cell not in the freelist, puts the cell into α*free_list*. Both *mark*( ) and *sweep*( ) check the type of a cell by

```
procedure Lαcons(i, x_1, x_2, ···, x_nα);
  begin { garbage collection dispatcher }
        if phase = mark_phase then
          begin mark();
                if gcs_empty() then phase := sweep_phase
          end
        elseif phase = sweep_phase then
          begin sweep();
                if sweeper > Htop then phase := idling
          end
        elseif αfree_count ≦ M_α then
          begin phase := mark_phase;
                sweeper := Hbtm;
                for i := 1 to NR do gcs_push(R[i])
          end;

        if αfree_count ≦ 0 then error( no storage for type α");

        { α cell allocation }
        p := αfree_list;
        αfree_list := p^.f_1;
        αfree_count := αfree_count - 1;

        p^.f_1 := x_1;
        p^.f_2 := x_2;
          . . . .
        p^.f_nα := x_nα;
        p^.mark := (p ≧ sweeper);

        R[i] := p
  end { of Lcons };
```

Figure 8. The consing procedure for type α.

the type field of the cell. The call to *consp*(x) must be replaced by a call to the boolean function that returns *true* if and only if its argument is a cell pointer. Other primitive operations such as *Lsetr*(i, x) and *Leq*(x, y) need not be changed.

The proof in Section 3 applies also to this system. The only change we have to make is to replace Invariant 3.7 with the following:

Invariant 6.1. The freelist for each type is loop-free.

In order to make sure that each consing procedure be executed successfully, we have to prove

Invariant 6.2. For each type α, the freelist of type α consists only of α cells.

But this is obvious because *sweep*( ) adds a non-marked cell into the freelist of the type of the cell.

Let us determine the sufficient condition for successful garbage collections. As in Section 4, we measure the course of computation by the total number of times that the consing procedures are called. Let $A(t)$ be the total number of accessible cells. For each cell type α, let $D_\alpha(t)$ be the number of times that $L\alpha cons$ is called, until time t. Also, let $F_\alpha(t)$ and $A_\alpha(t)$ be the number of freelist cells and the number of accessible cells, respectively, of type α at time t. Obviously, at any time t, $0 \leqq D_\alpha(t) \leqq t$ and the sum of $D_\alpha(t)$ for all types is equal to t. As in Section 4, assume that a garbage col-

J. SYSTEMS SOFTWARE
1990; 11: 181-198

lection begins at $t = a$ and ends at $t = c$. Also assume that *phase* is turned from *mark_phase* to *sweep_phase* at $t = b$ during the garbage collection. $F_\alpha(a)$ may be larger than M_α, since the garbage collection may be triggered by the consing procedure of the cell type other than α.

$$F_\alpha(a) \geqq M_\alpha$$

The time for the mark phase and the time for the sweep phase are the same as in Section 4.

$$b - a = \lceil A(a)/k1 \rceil$$

$$c - b = N/k2$$

Here, N is the sum of N_α for all type α. As in Section 4, we assume that $N/k2$ is an integer. In addition, we assume that $N_\alpha/k2$ is also an integer. By the same calculations as in Section 4, we can see that during the garbage collection, $F_\alpha(t)$ takes the smallest value

$$F_\alpha(a) - (D_\alpha(b + d + 1) - D_\alpha(a))$$

at $t = b + d + 1$, where $d = (N - N_\alpha)/k2 + \lfloor (A_\alpha(a) + F_\alpha(a))/k2 \rfloor$. $D_\alpha(b + d + 1) - D_\alpha(a)$, which represents the number of times that *Lαcons* is called between $t = a$ and $t = b + d + 1$, is bounded by $(b + d + 1) - a$. Thus, the sufficient condition for $F_\alpha(t) \geqq 0$ for $a \leqq t \leqq c + 1$ is

$$F_\alpha(a) - (b + (N - N_\alpha)/k2 + \lfloor (A_\alpha(a) - F_\alpha(a))/k2 \rfloor + 1 - a) \geqq 0$$

which is equivalent to

$$F_\alpha(a) \geqq ((N - N_\alpha)/k2 + A_\alpha(a)/k1 + A(a)/k2 + 1)/(1 - 1/k2)$$

By using $A_{\alpha\,\max}$ (maximum value of $A_\alpha(t)$), and by using $A_{\max}$ (maximum value of $A(t)$), we obtain the following sufficient condition for $F_\alpha(t) \geqq 0$ $(a \leqq t \leqq c + 1)$.

$$M_\alpha \geqq ((N - N_\alpha)/k2 + A_{\alpha\,\max}/k1 + A_{\max}/k2 + 1)/(1 - 1/k2) \tag{6.1}$$

Since

$$\begin{aligned} F_\alpha(c + 1) &= F_\alpha(a) + (N_\alpha - A_\alpha(a) - F_\alpha(a)) \\ &\quad - (D_\alpha(c + 1) - D_\alpha(a)) \geqq N_\alpha - A_\alpha(a) \\ &\quad - (c - a - 1) > N_\alpha - N/k2 - A_{\alpha\,\max} \\ &\quad - A_{\max}/k1 - 2 \end{aligned}$$

the sufficient condition for $F_\alpha(c + 1) \geqq M_\alpha$ is

$$N_\alpha - N/k2 - A_{\alpha\,\max} - A_{\max}/k1 - 2 \geqq M_\alpha \tag{6.2}$$

Theorem 6.1. Given an application program on our real-time system with multiple kinds of cells, if both (6.1) and (6.2) hold for each type α, then all garbage collection proceeds successfully.

Extension similar to those in Section 5 enable the real-time system with multiple kinds of cells to support the system stack and we obtain the following theorem. Here, *NSS* and $k3$ are those introduced in Section 5.

Theorem 6.2. Given an application program on our real-time system with multiple kinds of cells and with the system stack, if both

$$M_\alpha \geqq ((N - N_\alpha)/k2 + A_{\alpha\,\max}/k1 + A_{\max}/k2 + (NR + NSS)/k3)/(1 - 1/k2)$$

and

$$N_\alpha - N/k2 - A_{\alpha\,\max} - A_{\max}/k1 - (NR + NSS)/k3 - 1 \geqq M_\alpha$$

hold for each type α, then all garbage collection ends successfully.

Unfortunately, the conditions of Theorems 6.1 and 6.2 are too strong. According to Theorem 6.1, it is safe to set

$$M_\alpha = ((N - N_\alpha)/k2 + A_{\alpha\,\max}/k1 + A_{\max}/k2 + 1)/(1 - 1/k2)$$

for each type α, but this value of M_α seems too large if $A_{\alpha\,\max}$ is much smaller than $A_{\max}$. The primary reason for this is that, in the above calculation, we replaced $D_\alpha(t) - D_\alpha(t')$ by $t - t'$. The difference between these two values is quite large for those cell types that are scarcely used by the given program. In order to obtain a more practical estimation, we assume that the given program "proportionally" uses cell types. That is, we assume that there is a nonnegative number C_α for each type α such that

1. $D_\alpha(t) = C_\alpha * t$
2. $A_\alpha(t) = C_\alpha * A(t)$
3. the sum of C_α for all type α is 1

Under this assumption, (6.1) and (6.2) are respectively replaced by

$$M_\alpha \geqq C_\alpha * ((N - N_\alpha)/k2 + A_{\max} * (1/k1 + C_\alpha/k2) + 1)/(1 - C_\alpha/k2)$$

and

$$N_\alpha - C_\alpha * N/k2 - A_{\max} * (C_\alpha + C_\alpha/k1) - 2 * C_\alpha \geqq M_\alpha$$

Thus, a sufficient condition on N_α and N for $F_\alpha(t) \geqq 0$ is

$$N_\alpha - N*(2*C_\alpha/k2 - C_\alpha^2/k2^2) \geqq A_{\max}*(C_\alpha + 2*C_\alpha/k1 - C_\alpha^2/(k1*k2)) + (3*C_\alpha - 2*C_\alpha^2/k2)$$

Let us ignore $(3*C_\alpha - 2*C_\alpha^2/k2)$ in this inequality. Then, by adding this inequality for all type α, we obtain a sufficient condition on N.

$$N*(1 - 2/k2 + 1/(m*k2^2)) \geqq A_{\max}*(1 + 2/k1 - 1/(m*k1*k2))$$

where m is the number of cell types. Now a practically safe value of N is

$$N = A_{\max}*(1 + 2/k1 - 1/(m*k1*k2))/(1 - 2/k2 + 1/(m*k2^2))$$

Note that, if $m = 1$, then the safe value is identical to that given in Section 4. As m increases, this safe value of N also increases. For example, $N = 1.220A_{\max}$ in case $k1 = k2 = 20$ and $m = 3$. In this case, the real-time system needs 22.0% more space as the heap than the conventional system.

7. ARRAYS AND RELOCATION

Our real-time system can support arrays simply by regarding them as variable-length cells. The array allocation procedure *Lmake_array*(i, j, x), which allocates an array of j elements with all initial elements x and assigns (the pointer to) it to $R[i]$, may be defined similarly to the consing procedures in Section 6. *Laref*(x, j), which returns the jth element of (the array pointed to by) x, and *Laset*(x, j, y), which replaces the jth element of x by y, may be defined similarly to the retrieval functions and the update procedures, respectively, presented in Section 6. In order for the execution time of *mark*( ) to be bounded by a constant, we need special treatment when the pointer poped by

$p := gcs\_pop();$

points to an "array cell." If *mark*( ) pushed all of the elements of the array at once, then the real-time nature of the system would be lost, since the number of elements in an array is not bounded by a reasonably small constant. If we assume that the elements of an array are allocated in consecutive locations, which is usually the case, then the use of the save stack in Section 5 will overcome this difficulty. That is, when *mark*( ) recognizes that p points to an array, *mark*( ) copies the elements into the save stack so that they may later be taken care of. By using block transfer, the time for this copying will be negligible. In case that the application program uses many short arrays, it may be more efficient if *mark*( ) itself takes care of those arrays whose sizes are smaller than some small constant, immediately when the pointers to them are poped from *gcs*.

In many modern Lisps, arrays are treated as "first-class" data types. They are objects that can be assigned to variables, consed into list structures, and so on. There, it is expected that the storage occupied by arrays that are not used any more be recycled for further use. Unlike fixed-sized cells, simple linking of free arrays may cause the situation that there is no consecutive space large enough for a new array, while the total size of recycled space is large enough for the array. To avoid such a situation, the garbage collector should relocate or compact arrays in use so that they may be packed into a consecutive memory area.

In order to discuss how our real-time algorithm can be applied for array relocation, we use the following model, which is based on Minsky garbage collection [1,4,7] restricted on arrays. Each array is represented by a fixed-sized header and a body. The header contains useful information on the array, such as the length of the array. The elements of the array are stored in the body. Since the size of array headers is fixed, the system can treat array headers in the similar way as other fixed-sized cells. In particular, array headers are allocated in the heap and headers of non-used arrays may be linked together to form a freelist of array headers. Array bodies, on the other hand, are allocated in a separate space. The body of an array occupies consecutive locations in that space and the header of the array holds the first such location. Reference to an array is performed via the header; no pointer can directly point to array elements. The space for array bodies is divided into two *semispaces*. During execution of the application program, all array bodies are allocated in one of the semispace. During the mark phase, when the garbage collector is going to mark an array header, the array body is copied into the other semispace and, at the same time, the old location of the body stored in the header is updated. By copying array bodies into successive locations, bodies of accessible arrays are compacted in the "to" semispace (*tospace*) at the end of the garbage collection. The contents in the old semispace (*fromspace*) are then discarded and bodies of new arrays are allocated in the tospace. Next time the garbage collector is invoked, the role of the two semispaces is interchanged. The previous tospace is used as the fromspace and the previous fromspace is used as the tospace. Note that, since the location of an array body is stored only in the header, this system need not leave the so-called "forwarding address" [1,3] in array bodies.

Application of our real-time algorithm to this model is quite straightforward. The procedure *mark*( ) now copies array bodies into two places: to the tospace and to the save stack. The copying processes can be done in a short time, by using block transfer. During garbage collection, bodies of new arrays are allocated in the tospace, not in the fromspace. Thus, the tospace consists of copies of accessible array bodies and new array bodies, *sweep*( ) collects non-marked (i.e., inaccessible) array headers into a freelist, but does nothing with array bodies.

Actually, *mark*( ) needs to copy array bodies only into the tospace, if the system takes care of pointers in the tospace as well as pointers in the save stack. By making only one copy for each accessible array, the size of the save stack can remain small, and thus we can save memory space. As already stated, the tospace contains newly allocated bodies as well, which need not be taken care of. It is not difficult to distinguish copied bodies from newly allocated bodies. One method is to add an extra datum into the tospace, when a body is copied from the fromspace or a body is newly allocated. Each such datum should contain two kinds of information: whether the following body is copied or newly allocated and how long the body is. With this information, the system can easily and efficiently ignore newly allocated bodies in the tospace.

8. CONCLUSIONS AND FUTURE PLANS

We presented an algorithm for real-time garbage collection in list-processing systems running on general-purpose machines. This algorithm enables the list-processing system to execute each list-processing primitive within a small constant time and thus not to suspend execution of application programs during garbage collection. Although the execution efficiency decreases with the real-time garbage collection, the overhead is kept small because the algorithm puts no overhead on frequently used primitives such as pointer references, variable references and assignments, and stack manipulations. In order to see the memory overhead of the algorithm, we have shown sufficient conditions on the size of the heap to keep the program running without exhausting the freelist. These conditions are too strong in that the size of the heap can be much smaller in actual situations. Nevertheless, they have proved that the memory overhead of our real-time algorithm is relatively small. Application of the algorithm to already existing list-processing systems is easy since it does not require modification on the data representation. The primary disadvantage of the algorithm is that they do not support compaction of the whole data space. However, we have seen that the algorithm efficiently supports array relocation which is highly desired in modern list-processing systems.

The algorithm presented in this paper are planned to be implemented in a portable Common Lisp [16] system, called Kyoto Common Lisp [18], which is already running under several operating systems on several general-purpose machines, including the VAX and the MC68000. The kernel of this system is written in the C language and with the use of preprocessor macros of C, all versions of the system share the same source programs. This system allocates data cells in the heap and essentially uses the so-called BIBOP (BIg Bag Of Pages) method [15] to manage them. Variable-length data such as arrays and hash tables (in terms of Common Lisp) are allocated in another space and are garbage-collected by copying compaction. Implementation of the real-time algorithm in this system is straightforward and we expect that the same source programs can be shared also by the future versions of the system with the real-time garbage collector.

ACKNOWLEDGMENT

The author wishes to acknowledge the help of Reiji Nakajima who patiently supervised this research. This research was motivated by the implementation discussions of Kyoto Common Lisp with Masami Hagiya. The author wishes to thank him. The author also thanks Daniel Berry for his numerous comments on drafts of this paper.

REFERENCES

1. H. G. Baker, List Processing in Real Time on a Serial Computer, *Commun. ACM* 21(4), 280–294 (1978).
2. M. Ben-ari, Algorithms for On-the-fly Garbage Collection, *ACM Trans. Program. Lang. Syst.* 6(3), 333–344, (1984).
3. R. A. Brooks, Trading Data Space to Reduce Time and Code Space in Real-Time Garbage Collection on Stock Hardware, in *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming,* pp. 256–262, 1984.
4. C. J. Cheney, A Nonrecursive List Compacting Algorithm, *Commun. ACM,* 13(11), 677–678 (1970).
5. P. L. Deutsch and D. G. Bobrow, An Efficient, Incremental, Automatic Garbage Collector, *Commun. ACM*19(9), 522–526 (1976).
6. E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, On-the-fly Garbage Collection: An Exercise in Cooperation, *Commun. ACM* 21(11), 966–975 (1978).
7. R. R. Fenichel and J. C. Yochelson, A LISP Garbage-Collector for Virtual-Memory Computer Systems, *Commun. ACM* 12(11), 611–612 (1969).
8. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Inc., New Jersey, 1978.

9. H. T. Kung and S. W. Song, An Efficient Parallel Garbage Collection System, in *18th Annual IEEE Symposium on Foundations of Computer Science,* Providence, Rhode Island, pp. 120–131, 1977.
10. H. Lieberman and C. Hewitt, A Real-Time Garbage Collector That Can Recover Temporary Storage Quickly, MIT A.I.Memo, No. 569, 1980.
11. J. McCarthy, Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I, *Commun. ACM* 3(4), 184–195 (1960).
12. D. A. Moon, Garbage Collection in a Large Lisp System, in *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming,* pp. 235–246, 1984.
13. I. A. Newman, R. P. Stallard, and M. C. Woodward, A Parallel Compaction Algorithm for Multiprocessor Garbage Collection, in *Parallel Computing 83,* North-Holland Publishing Company, pp. 455–462, 1983.
14. G. L. Steele, Multiprocessing Compactifying Garbage Collection, *Commun. ACM* 18(9), 495–508 (1975).
15. G. L. Steele, Data Representations in PDP-10 MacLisp, in *Proceedings of the 1977 MACSYMA User's Conference,* Washington, D.C., 215–224, 1977.
16. G. L. Steele, et al., *Common Lisp: The Language,* Digital Press, 1984.
17. S. Wholey and S. E. Fahlman, The Design of an Instruction Set for Common Lisp, in *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming,* pp. 150–158, 1984.
18. T. Yuasa and M. Hagiya, *Kyoto Common Lisp Report,* Teikoku Insatsu Publishing, Kyoto, 1985.