学位申請論文

湯淺太一

学位審査報告

氏名	湯淺太一
学位の種類	理学博士
学位記番号	論理博第　　　号
学位授与の日付	昭和　　年　　月　　日
学位授与の要件	学位規則第5条第2項該当

（学位論文題目）

Realtime Garbage Collection on General－purpose Machines.

（汎用コンピュータ上の実時間ごみ集め）

論文調査委員	主査　一松　信 高須　達　　松浦重武

理学研究科

氏　名	湯　淺　太　一

（論文内容の要旨）

人工知能研究などに広く使われるLispなどの言語では，データをリスト構造によって表現する。すなわちデータをその一部（car）と，以後の情報を含むセルの位置を示すポインタ（cdr）を合せたセルの列で表現する。これによりデータの加除・変更が容易にでき，不定長データの処理に便利だが，反面多量の記憶を要する。記憶容量は有限であり，使用ずみのセルの適切な再利用を計らないと，飢餓状態になって，コンピュータでの処理が不可能になる。一度使ったセルのうち不要になったものを回収して再利用する処理が，標題にいうごみ集めである。

従来のリスト処理システムでは，ごみ集めの処理の間はプログラムの実行を中断する一括型が普通だった。この処理は，ときとしてプログラムの実行以上の時間を要し，目的によっては非常な危険性をはらんでいた。

そのため計算処理と並行して行う実時間ごみ集めの方法が研究され，いくつかの手法が提案されている。しかしその多くはごみ集め専用の中央処理装置を付加するもので，通例の汎用コンピュータには不向きであった。

以上のような背景の下で，申請者は一般のコンピュータに適用可能な効率よい実時間ごみ集めの算法を初めて考案した。その算法は，ごみ集め処理を個々の処理時間は無視してよいほどの細かい単位に分割し，基本機能中で一度に一単位ずつ実行することを基本原理とする。具体的には，セルを新に使ったときには格別の処理をせず，リストセルをヒープ上にアロケートするとき（Lisp でいえば，cons 関数が呼び出された際）と，リストセルを破壊的に操作するとき（Lispでいえばrplacaとrplacdが呼び出された際）に，ごみ集め処理を部分的に実行する。しかも全体として記憶の余裕がある限度以下になったとき，初めてごみ集め処理を実行開始する。

申請者は優れた算法を提唱しただけでなく，その正当性を数学的に厳密に証明した。ごみ集め処理中にも，プログラムは実行されていて状況の変化が生じ得るために，これはかなり困難な作業である。申請者は，まずこの算法が対象

とするリスト処理システムのモデルを与え，これに対する具体的なプログラムの形で算法を記述した。そしてリスト処理機能の実行の前後につねに保存される不変特性（invariamt）をいくつか挙げ，それらを帰納法によって証明する方法をとった。ここで不変特性全体を，複数のレベルに分けて証明する手法により，困難な証明をかなり簡易化することに成功している。

この方法は，今後計算機科学における多くの実時間算法の正当性の証明に有効に活用できることが期待される。

最後に申請者は，実際にコンピュータに実現した場合の具体的な評価を行い，最悪の場合でも，20％程度の記憶容量の余裕があれば，飢餓状態は生じないことを示している。このためにも証明のために導入した不変特性を活用し，必要とする記憶容量やごみ集め回数がどのように増減するかを詳しく解析している。これはプログラムの正当性検証技法が性能評価にも有効であることを示した例として，興味深い。

氏　名	湯　淺　太　一

（論文審査の結果の要旨）

リスト処理システムにおいて，実時間ごみ集めが重要であることは，論文内容の要旨中に述べた通りである。特にロボットのような実時間制御を要する応用においては，実行中断の一括型ごみ処理では，その間に大事故を招く危険性さえある。

申諸者の提案した実時間ごみ集めの算法は，特別の付加装置のない汎用計算機に適用できること，現存のリスト処理システムへ容易に応用できること，システム全体の処理効率の低下が僅かであること，余分に要する記憶が比較的少いことなどで，実用上に極めて高い価値をもつ。

さらに重大な成果は，提唱した算法に正当性の証明をつけたことである。これまで提唱された多くの算法では，しばしば算法そのものがあいまいで，適用限界が明確でなく，経験的に正しいようだと判断される例が多かった。ごみ集め処理中にもプログラムの実行は進んで状況が変化するので，各セルを正しく不要か否か判断するのは容易でないことを考えても，この証明が簡単なものではないことが推案される。じっさいこのような厳密な証明に成功したごみ集め算法は，初めてであり，この点でも申請者の独創性が高く評価される。申請者のとつた不変特性を挙げて帰納法で証明する手法は，計算機科学において標準的なものであるが，不変特性を複数のレベルに分けて証明するなど，巧妙な技法を駆使している。この方法は類似の諸問題の証明にも有効と思われる。

さらにこの不変特性をリスト処理システムの動作解析に適用し，正確な性能評価を与え，プログラムの正当性検証技法が性能評価にも有用なことを示している。

本論文は一言でいえば，計算機科学の永年の懸案の難問に，ほぼ決定的といってもよい解決策を提案し，しかもその正当性を厳密に証明した上，性能評価までを行ったものであり，計算機科学の数学的基礎として，極めて優れた成果である。

この論文はすでに内外から高い評価を受けている。例えばUCLAのBerry 教

授は「自分はこの問題を14年間研究して多くの論文を書いたが，これはnovel algorithmの他，その正統性の証明にunusual techniqueを駆使しており．重要な寄与である」と評しているほどである。

参考論文は16篇ある。その内には申請者の作成したKyoto Common Lispのマニュアルや，開発したシステムの解説書も含まれている。参考論文の多くは，所属研究機関において共同で開発してきたプログラム検証系イオタ言語およびそのプログラム作成支援システムの学術報告・ソフトウェア開発手法の研究である。Kyoto Common Lispはすでに国際的にも高く評価され，日本のソフトウェア業界にも大きな刺激を与えた製品である。このように，申請者は，計算機科学の理論的研究に優れた成果を挙げただけでなく，ソフトウェアの作成・開発の実際面にも抜群の能力を発揮している。

よって本論文は，理学博士の学位論文として価値あるものと認める。

なお，主論文および参考論文の内容を中心として，これに関連した研究分野について試問した結果，合格と認めた。

主論文

February 1986
Revised September 1986

Realtime Garbage Collection on General-purpose Machines

·Taiichi Yuasa
Research Institute for Mathematical Sciences, Kyoto University

Abstract

Algorithms of realtime garbage collection are presented, which are intended for list processing systems on general-purpose machines, i.e., Von Neumann style serial computers with a single processor. On these machines, realtime garbage collection inevitably puts some overhead on the overall execution efficiency of the list processing system, because some of the list processing primitives must check the status of the garbage collection. By avoiding such a check during execution of frequently used operations such as pointer references (e.g., Lisp car and cdr) and stack manipulations, the presented algorithms reduce the execution overhead to a great extent. Although these algorithms do not support compaction of the whole data space, they efficiently support "partial compaction" such as array relocation. A rigorous proof of the algorithms is given, first by postulating several invariants that hold between calls to the list processing primitives, and then by proving the invariants by induction. The proved invariants are then used for the evaluation of the presented algorithms.

1. Introduction

Garbage collection is the most popular method to reclaim discarded list cells in list processing systems such as Lisp. Although there are several variations of garbage collection, they share essentially the same scheme: Available cells are collected together to form a *freelist* and one cell is removed from the freelist each time the list processing program requires a new cell. When the freelist exhausts or becomes too short, the list processing program is temporarily suspended and the program of garbage collection (the garbage collector) 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 list processing program is resumed.

The primary disadvantage of garbage collection is that it periodically suspends the main list processing program. Roughly speaking, the time for a garbage collection is estimated as $\alpha A + \beta N$, where A is the number of accessible cells when the garbage collector is invoked and N is the size of the whole memory space. α and β are some positive factors. As the list processing program uses more cells, the time for each garbage collection and thus the suspension time of the main list processing program becomes longer. As reported in [15], for typical list processing programs, each garbage collection takes from several seconds to several ten seconds. It is difficult for an interactive or realtime list processing system to provide adequate service when it frequently suspends the execution of the main program for such a long time. This is typically the case of AI (Artificial Intelligence) applications for which list processing systems are mostly used. For example, if the program controls a robot in a product line of a car factory. the robot will stop its movement periodically while the line keeps moving. As the result, most cars from the factory cannot be sold because some parts may be missing or the bodies may be partially painted, depending on the role of the robot. In the worst case, the movement of the robot asynchronous with other facilities may cause a disastrous accident.

In order to remove this suspension of list processing programs, several algorithms of "realtime" garbage collection have been proposed [1,2,5,6,10,11,14,15]. A realtime garbage collector runs in parallel with the main program so that the time for each list processing primitive is bounded by some small constant. Most of these realtime algorithms are intended for multi-processor machines. The basic idea is to use one processor for garbage collection while another processor is responsible for the execution of the main program. Unfortunately. those algorithms are not appropriate for general-purpose machines on which many list processing systems are running. Here, *general-purpose machines* refer to Von Neumann style computers with a single processor, such as MC68000 and VAX. For these machines, no supports are expected from the underlying hardware nor from the firmware. If the algorithms were simulated on general-purpose machines, the overall system efficiency would be reduced to a great extent because of the frequent switching between the main process and the garbage collection process. Now a days, some Lisp machines are available as commercial products, but there are much more Lisp systems (or list processing systems in general) in use on general-purpose machines. It seems that many further Lisp systems will become available on general-purpose machines in the future. Hence the need for an efficient realtime algorithm for general-purpose machines.

On the other hand, Baker's realtime algorithm [1] is inherently serial and has been implemented on single-processor machines [13]. 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 [11] 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 [18].

This paper presents and discusses algorithms of realtime garbage collection on general-purpose machines. These algorithms were designed with two principles kept in mind. A realtime garbage collection on general-purpose machines inevitably puts some overhead on the execution efficiency of the list processing system. The primary reason is that some of the list processing primitives must check the current status of garbage collection. Such a check is unnecessary in conventional list processing systems with stop garbage collector. In order to reduce the total overhead due to the realtime-ness of garbage collection, our first design principle is that the possible overhead on frequently used primitives should be small or even none. In particular, the presented algorithms were designed so that they put no overhead on the operations of pointer references, assignments, and stack manipulations, while they put an extra burden on destructive list operations such as `rplaca` and `rplacd` in Lisp.

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

In the next section, we explain the basic idea of our realtime algorithms, first by introducing a simple list processing system with no stack mechanism and with only a single kind of cells, and then by extending it into a realtime system. The correctness proof of this simple realtime system is given in Section 3. Section 4 discusses the dynamic behavior of the realtime system. The algorithms are then applied 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 algorithms are easily applied to a system with array compaction. The discussions in Sections 3 and 4 are also applicable to these extensions, with minor modifications.

In order to provide a rigorous proof and evaluation of the algorithms, we obviously have to present the algorithms rigorously. For this purpose, we use slightly changed version of Pascal [8] in this paper. Deviations from Pascal are:

1. Type declarations for parameters and variables are omitted if there is no fear of ambiguity. Namely, we use the convention that `x`, `y`, `z`, and `p` are variables of the `pointer` type and `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 [9]. When a pointer p points to the i-th 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 of the pointers point to elements of a same array. For instance, when p and q point to the i-th and j-th elements, respectively, of an array, then $p<q$ is true if and only if i is strictly less than j.

2. The method

In this section, we explain the basic idea of our realtime garbage collection algorithms. Before going to the explanation, we first present a simple, Lisp-like list processing system with conventional stop garbage collection. This system is simple in that it supports only a single type of cells (*cons cells*) and that it does not have the stack mechanism. This system, however, has the essential aspects of real list processing system.

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 maximum of N cells are available. 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 function 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 fixed number. The user program can access and modify the contents of R by primitive operations Lgetr and Lsetr Lgetr(i) returns the i-th element of R and Lsetr(i,p) replaces the i-th element of R with p. Pointers in R are called *root pointers* and only those cells reachable directly or indirectly from the root pointers are accessible. Other cells are inaccessible and thus are garbage. 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 user program to prevent all cells in use from being garbage-collected unexpectedly. In particular, the system assumes that only accessible pointers are passed as pointer arguments to primitive operations.

In addition to Lgetr and Lsetr, the system provides primitive operations Lcons, Lcar, Lcdr, Lrplaca, Lrplacd, and Leq, each of which corresponds to a Lisp function in the obvious manner. Unlike Lisp, Lcons in the system 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 set a new list consisting of two nils into R[1], the user program is written as

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

Fig.1 illustrates an implementation of our simple list processing system with conventional stop garbage collection. 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 a convention to simplify the explanation. Available cells could be allocated directly from the heap until the first garbage collection occurs, as in most Lisp systems.

The garbage collector is invoked when the freelist exhausts. During the mark phase, the garbage collector marks (i.e., turns true the mark fields of) all accessible cells by recursively traversing list structures by the use of the garbage collection stack gcs. Then during the sweep phase, the heap is sequentially parsed so that all inaccessible cells, i.e. all non-marked cells, are collected into the freelist. Note the use of the push operation gcs_push. 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 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 Fig.2. Our intention is to keep the algorithms simple and clear.

Now we extend the above system so that it be a realtime system (see Fig.3).

In the realtime version, the garbage collection proceeds while the user program keeps running. Since we assume only a single processor, the two processes cannot proceed in really parallel. Instead, the whole process of garbage collection is divided into "chunks", each of which can be executed in time less than some constant time, and is executed in certain "situation" The situation we chose is when a cell is required by the user program, i.e., when Lcons is invoked. This idea is credited to Baker [1]. Since cells are requested while garbage collection is in progress, we cannot wait until the freelist exhausts. Instead, a garbage collection begins when the number of the freelist cells becomes less than a certain number M. To keep the current number of freelist cells, we introduce an integer variable free_count.

Another problem arises on the freelist during the sweep phase. In the stop collector, the freelist is set empty right before the sweep phase. With the realtime collector, the main program keeps requiring cells even during the sweep phase. If the system sets the freelist empty at the beginning of the sweep phase, then the sweeping steps must be repeated until a non-marked cell is found in the heap during the next call of Lcons. This method destroys the realtime-ness of the collector, since there is no small upper bound for the time required to find a non-marked cell. Rather, our realtime system leaves the freelist but collects only those non-marked cells that are not in the freelist yet. To determine whether a non-marked cell is in the freelist, we use the cdr fields of freelist cells, which are not used 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 pointer leave_me, which is distinguished from any meaningful pointers and which the user program cannot access directly.

The procedure mark() in Fig.3 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 be marked, that is, if gcs becomes empty before the body of the **while** statement is repeated k1 times, then the execution of mark( ) ends 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 Fig.1 is replaced by the global variable sweeper so that calls to sweep( ) can continuously 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 realtime system is that, during a sweep phase, it collects those and only those cells that are inaccessible nor in the freelist at the beginning of the garbage collection. Although some cells may become garbage during the garbage collection, they are not collected until the next garbage collection. In order to realize this feature of the system, the system needs to have the following properties.

1. All accessible cells at the beginning of the garbage collection are eventually marked during the mark phase.

2. Newly allocated cells during the garbage collection are never collected during the sweep phase of that garbage collection.

The **if** statement

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

in the definition of Lrplaca in Fig.3 and the **if** statement of Lrplacd are added for the first property. If the list structures used in the main program are never changed during the mark phase, then repeated calls of mark( ) certainly mark all accessible cells as in the case of the stop collector. However, Lrplaca may possibly modify the list structures and thus, without the **if** statement, there is the possibility that an accessible cell is not marked during mark phase. This is mainly because mark( ) cannot mark a cell once the cell becomes non-reachable 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);
```

Fig.4 illustrates the status of the relevant cells when Lrplaca is invoked. Assume that the cell β in the figure is pointed to only by the car field of the cell α. 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 collected as a garbage during the sweep phase while it still remains accessible. This situation cannot occur since the pointer to β is pushed onto gcs by the statement gcs_push(x^.car) at the beginning of Lrplaca(x,y). As of the second property, the statement

```
        p^.mark := (p ≥ sweeper);
```

in Lcons(i,x,y) is added for this property. During the mark phase, this statement effectively marks all newly allocated cells. During the sweep phase, this statement marks only those newly allocated cells that are not yet processed by sweep(). Other newly allocated cells during the sweep phase need not be marked since sweep() processes each cell only once. Of course, these discussions do not make sure that the system certainly has the above feature. We need a rigorous proof, which we will give in the next section.

The realtime system puts no extra burden on the primitives Lcar, Lcdr, Lsetr, Lgetr, and Leq. About Lcons, the 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 compared with the overall execution time of Lcons. Lrplaca and Lrplacd suffer from the overhead due to realtime-ness, but these operations are used less frequently than other primitives in actual list processing programs. They may be used internally for assignments to variables in those Lisp systems that implement variable bindings by association lists [12]. Even in such Lisp systems, local variables in compiled programs are usually allocated on the stack and assignments to variables can be implemented by Lsetr as will be shown in Section 5. In order to finish the efficiency discussions on our realtime system, we still need to analyze how many times the garbage collector is invoked. This analysis requires the understanding of some properties of the garbage collector, which we will present in the next section in terms of invariants. Thus we leave the analysis 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 compiled code. Usually, calls to Lcons (or its equivalent) are not expanded inline because the body of Lcons is too large even in conventional systems. Calls to Lrplaca and Lrplacd are rare in compiled programs. Since each of the other primitives is of the same size in the realtime system as in the conventional system, the realtime system promises that the size of compiled code is kept small.

Now we need to show the memory overhead of the realtime system. First of all, gcs needs no extra space. As of the size of the heap, we need to analyze the dynamic behavior of the system, which is the main topic of Section 4.

3. Proof

One of the reasons that the conventional stop garbage collection has been so widely accepted is its simplicity: The process of garbage collection proceeds without intervention of the user program. Even the allocation of a free cell is done independently of the garbage collector. Thus it is clear that the mark phase marks all and only accessible cells, and that the sweep phase collects all and only non-marked (garbage) cells. This is not the case of our realtime algorithm, however. The whole process of garbage collection is divided into chunks, each of which is executed while the execution of the user program proceeds. Even during the garbage collection, the user program continues to require new cells and make garbage cells. The correctness is thus beyond our intuition and we need a rigorous proof for the algorithm.

As already mentioned in the previous section, the most important feature of our realtime system is the following.

Theorem 3.1. Those and only those cells that are not accessible nor in the freelist at the beginning of a garbage collection will be put into the freelist during the sweep phase.

In order to prove this, we postulate several *system invariants* that characterize the realtime system. Here, a *system invariant* (or invariants, for short) refers to a property of the system that holds between calls to primitive operations. Except for those invariants that are direct consequences of other invariants, we prove each invariant by induction on the call of primitive operations. That is, in order to prove an invariant, we need to show that the property holds initially, i.e., immediately after the system initialization procedure init(), and to show that the property holds after execution of each primitive operation, by assuming that *all* invariants (including the invariant to prove) hold before the execution. We also assume that pointer arguments to the primitive operation are all accessible. Since the complete proof 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 proceeded by induction.

The first four invariants are concerned with gcs and sweeper

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

Proof : Initially, gcs is empty and the system is in idling phase. When phase is turned from mark_phase to sweep_phase, gcs is certainly empty. Since the procedure gcs_push, which is the only operation to augment gcs, is invoked only during mark phase, gcs remains empty until phase becomes sweep_phase. Q.E.D.

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

Proof : When a cell pointer is pushed on gcs, the mark field of the pointed cell is set to true. This mark field remains true as long as the cell pointer is on gcs, since mark fields of cells are turned false only by the procedure sweep(), which is never called during mark phase. Q.E.D.

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

Proof : A cell pointer is pushed onto gcs only when the pointed cell is not yet marked, but any pointer already on gcs points to a marked cell. Q.E.D.

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

Proof : Initially, sweeper = Htop+1. sweeper is not changed during idling and mark phases. When phase is turned to mark_phase, sweeper is set to Hbtm. During sweep phase, sweeper is incremented as long as it points to a cell. When phase is turned to idling, sweeper Htop+1 again. Q.E.D.

Definition 3.1. 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* sweeper if $i \geq j$. Otherwise, the cell is said to be *below* sweeper

Note that, by Invariant 3.4, any cell is below sweeper during idling phase, and any cell is above sweeper during mark phase.

Definition 3.2. A cell *m* is said to be *markable* iff there is a sequence of distinct cell pointers q_0, q_1, ..., q_n ($n>0$) such that

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

Note that, since *m*.mark = q_n^.mark = false, a marked cell cannot be markable.

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 iff its cdr field is leave_me.

Invariant 3.8. The freelist is loop-free. That is, for each freelist cell *m*, there is no sequence of cell pointers q_0, ..., q_n ($n \geq 0$) such that

1. q_0 = *m*.car,
2. q_i^.car = q_{i+1} ($0 \leq i < n$), and
3. q_n points to *m*.

Invariant 3.9. No cell below sweeper is marked.

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

Before going to the proofs of Invariants 3.5 to 3.10, we postulate three invariants which are direct consequences of these invariants.

Invariant 3.11. No freelist cell is accessible (by Invariants 3.5 and 3.7).

Invariant 3.12. No freelist cell is reachable from gcs (by Invariants 3.6 and 3.7).

Invariant 3.13. During sweep phase, any accessible cell above sweeper is marked (by Invariants 3.10 and 3.1).

Proof of Invariant 3.5 : An inaccessible pointer may become accessible only by Lcon. By Lcons(i,x,y), only the pointer to the first freelist cell becomes accessible, because both the car and the cdr fields of the cell are replaced by already accessible pointers x and y. Since leave_me is not a cell pointer, it cannot be identical to the pointer to the first freelist cell and thus leave_me never becomes accessible. Q.E.D.

Proof of Invariant 3.6 : Lcons, Lrplaca, and Lrplacd are the only primitive operations that affect the reachability from gcs. The procedures mark() and sweep(), which are called from Lcons, do not augment the set of pointers reachable from gcs because the pointers pushed on gcs by mark() are already reachable from gcs and sweep() is called only in sweep phase during which gcs remains empty (Invariant 3.1). Although Lcons replaces the car and the cdr fields of the first freelist cell, this does not affect the reachability from gcs, because the first freelist cell itself is not reachable from gcs (Invariant 3.12). When phase is turned from idling to mark_phase, only the accessible pointers become newly reachable from gcs, but leave_me is not accessible then (Invariant 3.5). Thus, leave_me cannot become reachable from gcs by Lcons. By Lrplaca(x,y) (or Lrplacd(x,y)), only those pointers that are reachable from x^.car (or x^.cdr) or from y may become reachable from gcs. Since these pointers are accessible, none of them is identical to leave_me. Q.E.D.

Proof of Invariant 3.7 : By induction hypothesis of Invariants 3.11 and 3.12, the set of freelist cells is modified only by sweep() and Lcons. When sweep() adds a cell into the freelist, the cdr field of the cell is replaced by leave_me. When a cell is removed from the freelist by Lcons(i,x,y), the cdr field of the cell is replaced by the accessible pointer y which is distinct from leave_me. Note that the induction hypothesis of Invariant 3.8 makes sure that Lcons(i,x,y) really removes a cell from the freelist.Q.E.D.

Proof of Invariant 3.8 : sweep() adds a cell into the freelist only when the cdr field of the cell is distinct from leave_me. By the induction hypothesis of Invariant 3.7, this cell is not already in the freelist. Q.E.D.

Proof of Invariant 3.9 : The system never turns true the mark fields of cells below sweeper: Whenever gcs_push(x) is called, sweeper = Hbtm and thus no cell is below 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. Q.E.D.

Proof of Invariant 3.10 : When a freelist cell becomes accessible, its mark field is set true if it is above sweeper Immediately after the root pointers are pushed onto gcs by Lcons, those cells directly pointed to by root pointers are marked and other accessible cells become markable since no cell was marked before (Invariants 3.9 and 3.4). sweep( ) obviously preserves Invariant 3.10. What remains to show is that Invariant 3.10 holds after the execution of mark( ), Lrplaca, and Lrplacd.

To show that Invariant 3.10 holds after the execution of mark( ), it suffices to prove that, for each cell *m* that is markable before the execution of the loop body

```
p := gcs_pop();
gcs_push(p^.car);
gcs_push(p^.cdr);
```

of mark( ), either *m* remains markable or the mark field of *m* becomes true after the execution. To prove this, let q_0, q_1, ..., q_n ($n>0$) be a sequence of distinct cell pointers such that

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

If both p^.car and p^.cdr are distinct from q_1, ..., q_n, then these properties are preserved by the loop body and thus *m* remains markable. If p^.car = q_h for some h ($0 < h \leqq n$) but p^.cdr is distinct from q_1, ..., q_n, or, conversely, if p^.cdr = q_h for some h ($0 < h \leqq n$) but p^.car is distinct from q_1, ..., q_n, then after the loop body,

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

That is, *m* remains markable if h n and *m* is marked otherwise. Finally, if p^.car = q_{h1} and p^.cdr = q_{h2} for some h1 and h2 ($0 < h1 \leqq n$, $0 < h2 \leqq n$), then let h = max(h1,h2). Then the above properties of q_h, ..., q_n hold and thus, after the loop body, either *m* remains markable or *m* is marked.

Similarly, to show that Invariant 3.10 holds after the execution of Lrplaca and Lrplacd, it suffices to prove that, for each cell *m* that is markable before the execution, either *m* remains markable or the mark field of *m* becomes true after the execution. Since the proof is almost the same for both Lrplaca and Lrplacd, we only prove this for

Lrplaca. Let q_0, q_1, ..., q_n (n>0) be a sequence of distinct cell pointers as above. If x^.car is distinct from q_0, q_1, ..., q_n, then the properties of this sequence are preserved by Lrplaca and therefore *m* remains markable. If x^.car = q_0, then gcs_push(x^.car) does nothing because x^.car^.mark = q_0^.mark true (Invariant 3.2). If x^.car = q_n, i.e., if x^.car points to *m*, then gcs_push(x^.car) turns the mark field of *m* true. Thus *m* is marked after Lrplaca. If x^.car = q_j for some j (0<j<n), then after gcs_push(x^.car),

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

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

Invariant 3.14. Each cell reachable from gcs is either marked or markable.

Proof : Immediately after the root pointers are pushed onto gcs by Lcons, only accessible cells are reachable from gcs. As already shown in the proof of Invariant 3.10, accessible cells are either marked or markable at that time. No cell becomes newly reachable from gcs by mark( ) and, as shown in the proof of Invariant 3.10, cells that are either marked or markable before the execution of mark( ) are either marked or markable after the execution. No cell is reachable from gcs when sweep( ) is called during sweep phase. Thus Lcons preserves Invariant 3.14. By Lrplaca(x,y) (or Lrplacd(x,y)), only those cells that are reachable from x^.car (or x^.cdr) or from y may become newly reachable from gcs. Since these cells are accessible, they are either marked or markable already (Invariant 3.10). Q.E.D.

In order to simplify the discussions that follow, we "paint" cells with two colors when phase is turned from idling to mark_phase. Those cells that are then either accessible or in the freelist are painted in *red* and other cells are painted in *black*. In addition, at any moment during the execution of the system, we call a cell as an *active* cell if it is then either marked, reachable from gcs, or in the freelist. Cells not active are called *inactive*. Note that, by Invariant 3.10, accessible cells are active and, equivalently, inactive cells are inaccessible. Before proving Theorem 3.1, we reclaim one more system invariant.

Invariant 3.15. For each cell *m* above sweeper, *m* is active if and only if *m* is red.

Proof : Primitive operations other than Lcons do not change sweeper Thus, in order to show that these operations preserve Invariant 3.15, it suffices to show that they preserve the activeness of cells. This is obvious except for Lrplaca and Lrplacd, since other primitive operations do not affect mark fields, the reachability from gcs, nor the freelist. Since Lrplaca does not turn mark fields false, marked cells remain marked. The freelist is not changed by Lrplaca. As shown in the proof of Invariant 3.10, markable cells either remain markable or get marked. Thus active cells remain active after Lrplaca. Conversely, inactive cells (i.e., cells not active) remain inactive for the following reasons. As shown in the proof of Invariant 3.6, no inaccessible cell becomes reachable from gcs by Lrplaca. Since inactive cells are inaccessible, no inactive cell becomes markable by Lrplaca. Since Lrplaca does not mark inaccessible cells as shown in the proof of Invariant 3.5, inactive cells remain non-marked. Similarly, Lrplacd is shown to preserve the activeness property of cells. Now, let us prove that Invariant 3.15 holds after each call of Lcons. We divide this proof into three cases according to the phase before the call.

i) Case phase = idling: If phase remains idling after the call, then no cell is above sweeper because sweeper = Htop+1 then. If the call turns phase from idling to mark_phase, then all and only accessible cells become reachable from gcs, one cell is removed from the freelist and is marked, and other freelist cells remain in the freelist. Since sweeper is set to Hbtm, Invariant 3.15 holds after the call.

ii) Case phase = mark_phase: The procedure mark() preserves the activeness property of cells, because the above discussion on Lrplaca applies also to mark(). Obviously, the rest of the call of Lcons also preserves the activeness property. Thus after the call, a cell is active iff it is active before the call. Since all cells are above sweeper before the call, by the induction hypothesis of Invariant 3.15 itself, we conclude that, after the call, a cell is active iff it is red.

iii) Case phase sweep_phase: Let m be an arbitrary cell that is above sweeper after the call of Lcons. The operation to remove a cell from the freelist does not affect the activeness property of m, because, if m happens to be the cell removed from the freelist, then m is certainly marked after the operation. Since m remains above sweeper during the execution of sweep(), m is marked after sweep() iff m is marked before, and m is in the freelist after sweep() iff m is in the freelist before. Note that m cannot be reachable from gcs since gcs is empty during sweep phase. Thus sweep() also preserves the activeness property of m. Therefore, this call of Lcons does preserve the activeness property of m. By the induction hypothesis of Invariant 3.15 itself, after the call of Lcons, m is active iff m is red. Q.E.D.

Now we are ready to prove Theorem 3.1 given at the beginning of this section.

Proof of Theorem 3.1 : It suffices to prove that sweep() eventually puts all black cells into the freelist but it never puts red cells into the freelist. By the definition of sweep(), a call of sweep() puts a cell m into the freelist iff m satisfies the following conditions before the call of sweep() (and, therefore, before the call of Lcons).

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

In addition, since gcs is empty.

4. m is not markable

That is, m is inactive and above sweeper By Invariant 3.15, m is a black cell but not a red cell. On the other hand, each black cell satisfies these conditions as long as it is above sweeper Thus when sweeper encounters a black cell, that cell is certainly put into the freelist. Q.E.D.

4. The dynamic behavior

In this section, we analyzes the dynamic behavior of our realtime system. In particular, we are interested in the status of the freelist during the execution of a given user program. From the analysis, we postulate a sufficient condition on the system parameters N, M, k1, and k2 to avoid the "no storage" error during execution of Lcons. This condition suggests safe values of N and M for the given program and, in addition, provides an estimation of memory overhead of our realtime system. Also in this section, we estimate the number of times the garbage collector is invoked during execution of the given program. In the following discussions, we naturally make use of the system invariants proved in the previous section, as the theoretical basis of the discussions.

In order to measure the course of computation, we use the number of times that Lcons is invoked: "at time t" means "at the t-th call of Lcons" Given a user program, let T be the total number of times that Lcons is invoked. (If the program is to run endlessly, then let T be ∞.) Let $F(t)$ and $A(t)$ be the number of freelist cells and the number of accessible cells, respectively, at time t (t = 1, 2, ..., T). Note that $A(t)$ depends only on the given program, but not on the system parameters N, M, k1, and k2.

Let us trace $F(t)$. Clearly, $F(1)$ = N since all the cells are freelist cells when Lcons is called for the first time. 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, phase is turned to mark_phase and the first garbage collection begins. Suppose that the first garbage collection begins during the a-th call of Lcons and ends during the c-th call. Also suppose that, during the first garbage collection, phase is switched from mark_phase to sweep_phase by the b-th call of Lcons(i,x,y). Clearly,

$$F(\mathrm{a}) = \mathrm{M}$$

During the mark phase, gcs_pop( ) is called exactly k1 times for each call of Lcons, except for the last call. As discussed in the previous section, pointers to cells 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 exactly $A(\mathrm{a})$ times. Thus

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

($\lceil x \rceil$ gives the integer n such that $n \geqq x > n-1$. Incidentally, $\lfloor x \rfloor$ gives the integer n such that $n+1 > x \geqq n$.) During the sweep phase, the value of sweeper is incremented by k2 for each call of Lcons except for the last call. Since sweeper is totally incremented by N,

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

To simplify the calculation, hereafter we assume that N/k2 is an integer. With this assumption,

$$c - b = N/k2$$

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 not in the freelist at t - a) over the heap. For i = 0, 1, ..., N/k2, let $B(i)$ be the number of black cells among the first i*k2 cells H[0], ..., 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 of sweep() occurs during the (b+1)-th call of Lcons and thus, at time t ($b+1 \leqq t \leqq c+1$), $B(t - b \quad 1)$ black cells have been put into the freelist. Since one cell is removed from the freelist by each call of Lcons, we obtain

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

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, ..., N/k2
3. $B(N/k2) = N - A(a) - F(a)$ (= { number of black cells })

Since

$$\begin{aligned} B(i) = B(N/k2) \\ &- (B(N/k2) - B(N/k2 - 1)) \\ &\quad \ldots \\ &- (B(i + 1) - B(i)) \end{aligned}$$

$$\begin{aligned} &\geq N - A(a) - F(a) - (N/k2 - i)*k2 \\ &= i*k2 - A(a) - F(a) \end{aligned}$$

and since $B(i)$ is non-negative,

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

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

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

Note that in the extreme case when all black cells are located at the higher part of the heap, $F(t)$ is identical to $F_{lb}(t)$. $F_{lb}(t)$ decreases monotonically when $a \leq t \leq b+d+1$, but increases monotonically when $b+d+2 \leq t \leq c+1$. A simple calculation tells 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 $F(t) \geq 0$ $(a \geq t \geq c+1)$ is

$$F(a) - (b + \lfloor (A(a) - F(a))/k2 \rfloor + 1 - a) \geq 0 \tag{4.1}$$

which is equivalent to

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

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

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

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

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

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

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

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

Theorem 4.1. Given a list processing program on our realtime system, if both (4.2) and (4.3) hold, then all garbage collection proceeds successfully.

Theorem 4.1 is useful to find safe 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 = $0.105 A_{max}$ and N $\quad 1.216 A_{max}$. In comparison, for the conventional system with stop garbage collector in Fig.1, the smallest safe value for N is $:A_{max}$. In this case, therefore, the realtime system needs 21.6% more memory for the heap.

Now we will 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 i-th garbage collection begins during the a_i-th call of Lcons and ends during the c_i-th call. As already shown,

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

Immediately after the i-th garbage collection, the number of freelist cells are

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

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

$$F(t) = F(c_i+1) - (t - c_i - 1) \qquad \text{for } c_i+1 \leq t \leq a_{i+1}$$

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

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

From this, we obtain

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

This formula, together with the initial value of $a_1 \quad N - M + 1$, completely specifies the sequence $\{ a_i \}$. 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

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

(Remember T is the number of times Lcons is called during the execution of the given program.) For the conventional system with stop garbage collector given in Fig.1, the sequence { a_i } is defined by

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

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

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

This expression supports the widely believed rule that the larger N is, the less 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 estimation for the realtime system above suggests that the smaller M is and the larger N is, the less 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 realtime system. In contrast, with the safe value of $\mathrm{N} = A_{max}$ for the conventional system, the number of garbage collection is about T/A_{mean}. Thus, with these safe values of N and M, the realtime system causes *less* garbage collection than the conventional system. On the other hand, it is clear that, with the same size of heap, the realtime system causes more garbage collection than the conventional system. For instance, with the heap size $\mathrm{N} = 1.216A_{max}$, the conventional system calls the garbage collector about $0.7T/A_{mean}$ times. Thus the realtime system causes 17% more garbage collection than the conventional system.

5. System stack

In this section, we extend our realtime system so that the user program can handle the *system stack*. The system stack contains pointers and is typically used for argument passing and variable allocation. The system stack is like the root array R in that pointers on the system stack are regarded as root pointers. Unlike R, however, the size of the system stack differs from time to time and the maximum size of the system stack NSS is assumed much larger than NR, the size of R. 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.

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 Fig.5. Now, the pointers R[1], ..., 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 algorithms presented in Fig.3 correctly work, 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 realtime-ness of the system. Instead of processing the root pointers at a time, our revised system processes at most a fixed number of root pointers each time Lcons is called (see Fig.6). Since the contents of the system stack will be changed as computation proceeds, we need to save the contents of the system stack when 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 between some kilobytes and some ten kilobytes in most list processing systems, we can assume that the copying operation takes only a very short time.

Primitive operations other than Lcons are the same as those in the realtime system without the system stack. In particular, the revised realtime system puts 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 puts no extra burden on stack operations including direct access to the stack entities by Lsetr and Lgetr This means that the compiler can expand stack operations inline in compiled code without any penalty both in code size and in execution efficiency.

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.

Definition 5.1. A cell *m* is *markable* iff there is a sequence of distinct cell pointers q_0, q_1, ..., q_n ($n>0$) such that

1. q_0 is either on gcs or on the save stack,
2. either q_i^.car = q_{i+1} or q_i^.cdr = q_{i+1} ($0 \leq i < n$),
3. q_i^.mark = false ($0 < i \leq n$), 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 not reachable from gcs nor from the save stack.

And, accordingly, we must replace Invariant 3.12 with

Invariant 5.3. No freelist cell is reachable from gcs nor from the save stack.

These changes are necessary to make sure that the statements

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

in the body of Lcons do not affect the markability of cells (in the sense of Definition 5.1). Finally, the notion of "active" should be redefined as follows. A cell is *active* if it is marked, reachable either from gcs or from the save stack, or in the freelist. The proof is similar to the one in Section 3 and is left to the reader.

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 $t = d_1, \ldots, d_{n+1}$ ($a<d_1< \ldots <d_n<d_{n+1}=b$). Clearly, $n = \lceil SStop0/k3 \rceil$. Also assume that, after $t = d_i$ ($i = 1, 2, \ldots, n$), gcs_pop( ) is called e_i times until the save stack is processed next time. Then we

have

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

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.

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

($\Sigma\{i \mid P(i)\}[f(i)]$ 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

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

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 a user program for our realtime 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 realtime 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 realtime system needs $1.216A_{max}$ + 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 realtime system so that it supports other kinds of cells as well, such as symbol cells in Lisp systems. 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 αfree_list. The system keeps track of the maximum number of allocatable cells N_α, separately for each type α. A pointer can points to a cell of any type and, given a pointer, the system can determine which type of cells the pointer points to. 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, ..., f_{n_α} be the names of the pointer fields. Then we have the following primitive operations on each type α.

1. The allocation procedure Lαcons(i, x_1, ..., x_{n_α}), which allocates an α cell from αfree_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 Lαf_1(x),...., Lαf_{n_α}(x), similar to Lcar(x). Each Lαf_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 Lαrplacf_1(x, y), ..., Lαrplacf_{n_α}(x, y), similar to Lrplaca(x, y). Each Lαrplacf_j(x, y) receives two pointers, the first one pointing to an α cell, and replaces the f_j field of x with y.

As in Section 2, the system initialization procedure init( ) prepares freelists so that each freelist αfree_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 realtime system in Fig.3 is extended as follows. Each allocation procedure Lαcons(i, x_1, ..., x_{n_α}) begins with the same garbage collection dispatcher as that in Lcons in Fig.3, except that the dispatcher in Lαcons(i, x_1, ..., x_{n_α}) uses an α-specific number M_α instead of M (see Fig.7). That is, garbage collection begins when the size of αfree_list becomes less than or equal to M_α for some type α. The rest of each allocation procedure is similar to that of Lcons. Like Lcar, each retrieval function Lαf_j(x) simply returns the value of x.^f_j. Like Lrplaca, each update procedure Lαrplacf_j(x, y) checks the current phase before replacing x^.f_j with y. If the system is currently in mark phase, then it executes gcs_push(x^.f_j).

```
procedure Lαrplac$f_j$(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 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 αfree_list. Both mark( ) and sweep( ) check the type of a cell by the type field of the cell. The call of 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

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

In order to make sure that the allocation procedures 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 see 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 allocation 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 Lαcons 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 collection 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. $F_\alpha(\mathrm{a})$ may be larger than M_α, since the garbage collection may be triggered by the allocation procedure of the cell type other than α.

$$F_\alpha(\mathrm{a}) \geqq M_\alpha$$

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

$$\mathrm{b} \quad \mathrm{a} = \lceil A(\mathrm{a})/\mathrm{k1} \rceil$$

$$\mathrm{c} - \mathrm{b} = \mathrm{N}/\mathrm{k2}$$

Here, N is the some 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_α/k2 is also an integer. By the same calculation as in Section 4, we can see that during the garbage collection, $F_\alpha(t)$ takes the smallest value

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

at t = b+d+1, where d = $(\mathrm{N} - \mathrm{N}_\alpha)/\mathrm{k}2 + \lfloor(A_\alpha(\mathrm{a}) + F_\alpha(\mathrm{a}))/\mathrm{k}2\rfloor$. $D_\alpha(\mathrm{b+d+1}) - D_\alpha(\mathrm{a})$, 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

$$F_\alpha(\mathrm{a}) - (\mathrm{b} + (\mathrm{N} - \mathrm{N}_\alpha)/\mathrm{k}2 + \lfloor(A_\alpha(\mathrm{a}) - F_\alpha(\mathrm{a}))/\mathrm{k}2\rfloor + 1 - \mathrm{a}) \geqq 0$$

which is equivalent to

$$F_\alpha(\mathrm{a}) \geqq ((\mathrm{N} - \mathrm{N}_\alpha)/\mathrm{k}2 + A_\alpha(\mathrm{a})/\mathrm{k}1 + A(\mathrm{a})/\mathrm{k}2 + 1) / (1 - 1/\mathrm{k}2)$$

By using $A_{\alpha\,\mathrm{max}}$ (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).

$$\mathrm{M}_\alpha \geqq ((\mathrm{N} - \mathrm{N}_\alpha)/\mathrm{k}2 + A_{\alpha\,\mathrm{max}}/\mathrm{k}1 + A_{\mathrm{max}}/\mathrm{k}2 + 1) / (1 - 1/\mathrm{k}2) \qquad (6.1)$$

Since

$$\begin{aligned} F_\alpha(\mathrm{c+1}) &= F_\alpha(\mathrm{a}) + (\mathrm{N}_\alpha - A_\alpha(\mathrm{a}) - F_\alpha(\mathrm{a})) - (D_\alpha(\mathrm{c+1}) - D_\alpha(\mathrm{a})) \\ &\geqq \mathrm{N}_\alpha - A_\alpha(\mathrm{a}) - (\mathrm{c} - \mathrm{a} - 1) \\ &> \mathrm{N}_\alpha - \mathrm{N}/\mathrm{k}2 - A_{\alpha\,\mathrm{max}} - A_{\mathrm{max}}/\mathrm{k}1 - 2 \end{aligned}$$

the sufficient condition for $F_\alpha(\mathrm{c+1})\geqq\mathrm{M}_\alpha$ is

$$\mathrm{N}_\alpha - \mathrm{N}/\mathrm{k}2 - A_{\alpha\,\mathrm{max}} \quad A_{\mathrm{max}}/\mathrm{k}1 - 2 \geqq \mathrm{M}_\alpha \qquad (6.2)$$

Theorem 6.1. Given a list processing program on our realtime system with multiple kinds of cells, if both (6.1) and (6.2) hold for each type α, then all garbage collection proceeds successfully.

The similar extension as in Section 5 enables the realtime 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 a list processing program on our realtime system with multiple kinds of cells and with the system stack, if both

$$\mathrm{M}_\alpha \geqq ((\mathrm{N} - \mathrm{N}_\alpha)/\mathrm{k}2 + A_{\alpha\,\mathrm{max}}/\mathrm{k}1 + A_{\mathrm{max}}/\mathrm{k}2 + (\mathrm{NR} + \mathrm{NSS})/\mathrm{k}3) / (1 - 1/\mathrm{k}2)$$

and

$$\mathrm{N}_\alpha - \mathrm{N}/\mathrm{k}2 - A_{\alpha\max} - A_{\max}/\mathrm{k}1 - (\mathrm{NR} + \mathrm{NSS})/\mathrm{k}3 - 1 \geq \mathrm{M}_\alpha$$

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

$$\mathrm{M}_\alpha = ((\mathrm{N} - \mathrm{N}_\alpha)/\mathrm{k}2 + A_{\alpha\max}/\mathrm{k}1 + A_{\max}/\mathrm{k}2 + 1) / (1 - 1/\mathrm{k}2)$$

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 non-negative number C_α for each type α such that

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

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

$$\mathrm{M}_\alpha \geq C_\alpha*((\mathrm{N} - \mathrm{N}_\alpha)/\mathrm{k}2 + A_{\max}*(1/\mathrm{k}1 + C_\alpha/\mathrm{k}2) + 1) / (1 - C_\alpha/\mathrm{k}2)$$

and

$$\mathrm{N}_\alpha - C_\alpha*\mathrm{N}/\mathrm{k}2 - A_{\max}*(C_\alpha + C_\alpha/\mathrm{k}1) - 2*C_\alpha \geq \mathrm{M}_\alpha$$

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

$$\mathrm{N}_\alpha - \mathrm{N}*(2*C_\alpha/\mathrm{k}2 - C_\alpha{}^2/\mathrm{k}2^2)$$
$$\geq A_{\max}*(C_\alpha + 2*C_\alpha/\mathrm{k}1 - C_\alpha{}^2/(\mathrm{k}1*\mathrm{k}2)) + (3*C_\alpha - 2*C_\alpha{}^2/\mathrm{k}2)$$

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

$$\mathrm{N}*(1 - 2/\mathrm{k}2 + 1/(m*\mathrm{k}2^2)) \geq A_{\max}*(1 + 2/\mathrm{k}1 - 1/(m*\mathrm{k}1*\mathrm{k}2))$$

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

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

Note that, if m=1, then this safe value is identical to that given in Section 4. As m increases, this safe value of N also increases. For example, $\mathrm{N} = 1.220A_{\max}$ in case $\mathrm{k}1 = \mathrm{k}2 = 20$ and m=3. In this case, the realtime system needs 22.0% more space as the heap than the conventional system.

7. Arrays and relocation

Our realtime 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 allocation procedures in Section 6. Laref(x,j), which returns the j-th element of (the array pointed to by) x, and Laset(x,j,y), which replaces the j-th 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 the elements of the array at once, then the realtime-ness 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 user 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, it is expected that the garbage collector relocates (or compacts) arrays in use so that they may be packed into a consecutive memory area.

In order to discuss how our realtime algorithms can be applied for array relocation, we use the following model, which is based on the 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 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 user 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 realtime algorithms 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 work

We presented algorithms for realtime garbage collection in list processing systems running on general-purpose machines. These algorithms enable the list processing system to execute each list processing primitive within a small constant time and thus not to suspend execution of list processing programs during garbage collection. Although the execution efficiency decreases with the realtime garbage collection, the overhead is kept small because the algorithms put 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 algorithms, 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 realtime algorithms is relatively small. Application of the algorithms to already existing list processing systems is easy since it does not require modification on the data representation. The primary disadvantage of the algorithms is that they do not support compaction of the whole data space. However, we have seen that the algorithms efficiently support array relocation which is highly desired in modern list processing systems.

Since garbage collection proceeds while the user program keeps running, the correctness of the algorithms are not obvious and the proof is much more complicated than the proof of the conventional stop collector. In order to overcome the potential difficulty of the proof, we postulated several system invariants, each of which can be proved relatively easily. We naturally used induction on calls to the primitive operations for the proof of each invariant. The invariants were then used as the theoretical basis for the evaluation of the algorithms. One important thing we learned from this study is that proof of an algorithm is useful not only for the correctness but also for the strict discussion on the algorithm.

The algorithms presented in this paper are planned to be implemented in a portable Common Lisp [17] system, called Kyoto Common Lisp [19], which is already running under several operating systems on several general-purpose machines, including VAX and 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 [16] 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 realtime algorithms in this system is straightforward and we expect that the same source programs can be shared also by the coming versions of the system with realtime garbage collection.

Acknowledgement

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.

References

[1] Baker, Henry G., *Lisp Processing in Real Time on a Serial Computer*, *Comm. ACM*, vol.21, no.4, pp.280-294, 1978.

[2] Ben-ari, Mordechai, *Algorithms for On-the-fly Garbage Collection*, *ACM Transactions on Programming Languages and Systems*, vol.6, no.3, pp. 333-344, 1984.

[3] Brooks, Rodney A., *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] Cheney, C.J., *A Nonrecursive List Compacting Algorithm*, *Comm. ACM*, vol.13, no.11, pp. 677-678, 1970.

[5] Deutsch, Peter L. and Bobrow, Daniel G., *An Efficient, Incremental, Automatic Garbage Collector*, *Comm. ACM*, vol.19, no.9, pp. 522-526, 1976.

[6] Dijkstra, Edsger W., Lamport, Leslie, Martin, A.J., Scholten, C.S., and Steffens, E.F.M., *On-the-Fly Garbage Collection: An Exercise in Cooperation*, *Comm. ACM*, vol.21, no.11, pp. 966-975, 1978.

[7] Fenichel, Robert R. and Yochelson, Jerome C., *A LISP Garbage-Collector for Virtual-Memory Computer Systems*, *Comm. ACM*, vol.12, no.11, pp. 611-612, 1969.

[8] Jensen, Kathleen and Wirth, Niklaus, *PASCAL User Manual and Report*, Lecture Notes in Computer Science 18, Springer-Verlag, 1974.

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

[10] Kung, H.T. and Song, S.W., *An Efficient Parallel Garbage Collection System*, in *18th Annual IEEE Symposium on Foundations of Computer Science*, Providence, Rhode Island, pp. 120-131, 1977.

[11] Lieberman, Henry and Hewitt, Carl, *A Real Time Garbage Collector That Can Recover Temporary Storage Quickly*, MIT A.I.Memo, no.569, 1980.

[12] McCarthy, John, *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*, *Comm. ACM*, vol.3, no.4, pp. 184-195, 1960.

[13] Moon, David A., *Garbage Collection in a Large Lisp System*, in *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pp. 235-246, 1984.

[14] Newman, I.A., Stallard, R.P., Woodward, M.C., *A Parallel Compaction Algorithm for Multiprocessor Garbage Collection*, in *Parallel Computing 83*, North-Holland Publishing Company, pp. 455-462, 1983.

[15] Steele, Guy L., *Multiprocessing Compactifying Garbage Collection*, *Comm. ACM*, vol.18, no.9, pp. 495-508, 1975.

[16] Steele, Guy L., *Data Representations in PDP-10 MacLisp*, in *Proceedings of the 1977 MACSYMA User's Conference*, Washington, D.C., 1977.

[17] Steele, Guy L. *et.al.*, *Common Lisp: The Language*, Digital Press, 1984.

[18] Wholey. Skef and Fhalman Scott E., *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.

[19] Yuasa, Taiichi and Hagiya, Masami, *Kyoto Common Lisp Report*, Teikoku Insatsu Publishing, 1985.

Fig.1. The conventional system with stop garbage collector

```
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;

        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);
```

Fig.2. An alternative definition of gc( )

```
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);
```

Fig.3. The realtime system
(Operations not described here are same as in Fig.1)

```
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");

        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;
```

Fig.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).)

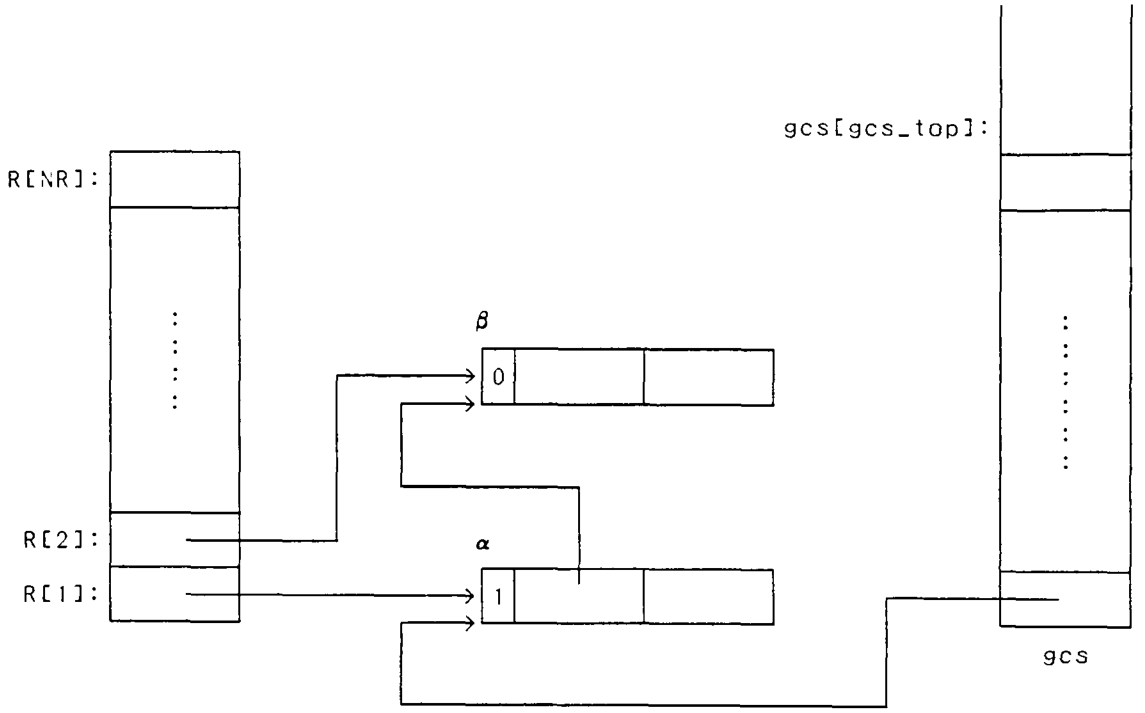

Fig.5. Primitive operations on the system stack

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

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

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

Fig.6. Lcons of the realtime system with the system stack
(Operations not described here are same as in Fig.3)

```
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");

        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);
```

Fig.7. The allocation procedure for type α

```
procedure L$\alpha$cons(i,$x_1$,$x_2$, ..., $x_{n_\alpha}$);
 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 $\alpha$free_count $\leqq M_\alpha$ then
         begin phase := mark_phase;
               sweeper := Hbtm;
               for i := 1 to NR do gcs_push(R[i])
         end;

       if $\alpha$free_count $\leqq$ 0 then error("no storage for type $\alpha$");

       p := $\alpha$free_list;
       $\alpha$free_list := p^.$f_1$;
       $\alpha$free_count := $\alpha$free_count-1;

       p^.$f_1$ := $x_1$;
       p^.$f_2$ := $x_2$;
         ....
       p^.$f_{n_\alpha}$ := $x_{n_\alpha}$;
       p^.mark := (p $\geqq$ sweeper);

       R[i] := p
 end (of Lcons);
```