Regular Paper

Time- and Space- Efficient Garbage Collection Based on Sliding Compaction

Mitsugu Suzuki [†] and Motoaki Terashima [††]

A new type of garbage collection (GC) based on sliding compaction is presented. It is called LLGC, and performs sliding compaction in a time proportional to A plus $Nf(N)$, where A is the total size of data objects in use, N is the number of clusters of such objects ($N \leq A$), and $\log N \leq f(N) \leq N$. It requires an additional space whose size is less than A. The time cost of $Nf(N)$ and the additional space cost result from a sorting scheme adopted in LLGC. When the load factor is small, the time cost is far less than that of conventional sliding compaction GC, which is proportional to the total storage space size, and the space cost is of no importance. Therefore, LLGC has a great advantage of time economy in such a case. When the load factor is large, it carries out conventional sliding compaction using no additional space. The advantages of LLGC are shown by experimental data for a successful implementation on PLisp, a dialect of Lisp.

1. Introduction

Automatic storage management, called garbage collection, or GC for short, is essential to the implementations of Lisp, Prolog, and other programming languages that provide dynamic data structures. Therefore, GC has been implemented on a great many software and hardware bases for more than thirty years, since the invention of Lisp and Algol.[1)]

In this paper we present a new type of GC called LLGC which is based on sliding compaction. The initials LL mean "time of Linear Log" and "use of Less Load factor", which sum up the features of our GC. The LLGC has a great advantage of time economy in cases where the load factor (α) is small; that is, when the ratio of the size of all the data objects in use to the total storage space is small. All the data objects in use are called "active data objects" hereafter, and their size is denoted by the symbol A, while the total storage space size is denoted by another symbol S. The load factor is now written as $\alpha = A/S$, using A and S.

GC based on sliding compaction is called sliding compaction GC, and has the merit of managing data structures of variable size efficiently as regards space. It performs a very sophisticated task of object relocation; all active data objects are gathered up into one end of the storage space with their allocated order prereserved so that no "hole" of unused space is made. Consequently their allocated order remains unchanged forever in accordance with "genetic order preserving", in the terminology of Terashima and Goto.[2)] For instance, this is indispensable for the implementation of Prolog based on WAM.[3)] Notwithstanding this merit, sliding compaction GC has been regarded as very expensive, owing to time-consuming nature of tasks such as pointer adjustment for each active data object and multiple scans of the storage space. The former problem was successfully solved by the invention of fast and bounded workspace algorithms that perform each pointer adjustment within a constant time independent of A or S.[4),5)] But the latter is still unsolved. Its time cost is obviously proportional to S, so that conventional sliding compaction GC requires a time proportional to S plus A.

LLGC has the feature of performing the sliding compaction in a time proportional to A plus $Nf(N)$, where N is the number of clusters and $\log N \leq f(N) \leq N$. A cluster is defined as a "brick" of successive active data objects or a single active data object that has no neighboring ones, while an (active) data object is defined as a collection of contiguous machine words (or "fields" in our terminology) in which part of a datum or a whole datum can be stored. Therefore, the number of clusters is not greater than that of all active data objects, or $N \leq A$ symbolically if A is counted by the field. LLGC requires an additional space called a U-space,

† Department of Computer Science, University of Electro-Communications
†† Graduate School of Information Systems, University of Electro-Communications

of a size less than A, in order to make a set of storage addresses for each active data object. The storage address points to the portion of a storage space in which a data object is located. A data object may have various values for its storage address, but only one is stored in the U-space as a storage address datum, such as CAR's and CDR's for a CONS data object. Therefore, the size of the U-space may be less than $A/2$ in the case of a typical implementation where a CONS data object consists of two machine words. The storage address data are unique and without duplication, and this is achieved by a slight modification of the marking process. They are then reduced to a set of storage addresses that point only to each cluster, and their size (i.e., the number of storage address data) becomes N. The form $Nf(N)$ depends on how such data are sorted in the U-space from scratch; their order may be irregular, and they need to be put into a specific order by means of a sorting technique. It is well-known that sorting requires a time proportional to $Nf(N)$. A set of such sorted storage address data is utilized in the following process, which will require a time proportional to only A.

If the load factor is very small, or $A \ll S$ symbolically, the time required by LLGC may be drastically shortened, because A and $Nf(N)$ are far less than S, and the size of the U-space is of no importance. The U-space is maintained so that its size does not exceed a predestinated value (e.g., $S/10$) for space economy. Therefore, when the load factor becomes large, the U-space may overflow; in this case, LLGC can carry out conventional sliding compaction GC (with the U-space given up).

The design and software implementation of LLGC are described in Section 3 with a data representation in PLisp (Portable Lisp), on which LLGC has been successfully implemented. LLGC is superior to other types of sliding compaction GC with respect to the total GC time in the case of a small load factor, as is shown by experimental data and also described in Section 4. Recently, applications based on Lisp have become very large, and require a lot of storage space for their execution. Therefore, they require a more efficient execution time and working space for GC. This is the design goal of LLGC.

2. Background

A GC scheme based on sliding compaction was invented as early as the 1960s, as well as other GC schemes based on copying collection[6] and free listing.[7] Though sliding compaction GC had the great merit of managing data structures of variable size efficiently for a small storage space, it was known to be very costly in time. Many sliding compaction schemes have since been proposed to improve the time cost.[8]

One target of such improvement was the pointer adjustment process, which seemed to be costly in time or space, or both. Consequently, fast and bounded workspace algorithms were invented, which perform each pointer adjustment within a constant time, using no additional space. One of them, known as Morris's algorithm,[4] scans through a storage space only twice, including for data relocation, but an extra one bit is required for each cell other than a marking bit. Therefore, strictly speaking, it cannot not be a bounded workspace algorithm. For this reason, LLGC uses another algorithm,[9] described in the next section.

Sliding compaction GC has a time cost proportional to the sum of A and S. Of course, a large part of the former may be eliminated by using the fast and bounded workspace algorithms described above. However, the latter has been left untouched, being regarded as indispensable to the sliding compaction GC. It has gradually come to account for a large part of the total cost.

The GC based on copying collection is also called copying collection GC, and it is suitable for data structures of variable size, as well as those of fixed size. Moreover it requires a time proportional only to A, and has therefore been widely used in many Lisp implementations, especially those using a large amount of storage space. The demerit of the copying collection GC is its storage utilization; only half of the storage space is available to a copying collection scheme at a time. The copying collection GC may destroy the genetic order of active data objects during the copying process. This is another demerit.

Some reports have compared the actual time costs of the two schemes based on sliding compaction and copying collection. Koide and Terashima used experimental data to show that the balance point at which two schemes are nearly equal in their processing time is 0.24

(by the load factor).[10] This indicates that (conventional) sliding compaction GC is inferior to copying collection GC as regards time economy unless the load factor is greater than about 0.24.

LLGC overcomes such inferiority to some extent. It performs sliding compaction in a time proportional to A plus $Nf(N)$. When the load factor is very small ($N \leq A \ll S$), the latter is closer to A than to S. Consequently, LLGC will perform sliding compaction in a time proportional to A under such a condition. LLGC uses a sorting technique in order to achieve its purpose (i.e., the time cost described above). Introduction of a sorting technique into a GC scheme is not new in itself. Koide and Noshita have already described the use of a sorting technique to design a copying collection GC that effectively preserves the genetic order, which differs from our framework.[11]

3. Implementation

This section presents the design and implementation of LLGC on Plisp, which is a dialect of Lisp.

3.1 Data Representation of PLisp

PLisp is a compiler-based portable Lisp system being designed as compatible with Common LISP,[12] and is characterized by the use of a large storage space such as 256 (2^{28}) MB, in which many data type objects are located. A data object of PLisp is made up of a storage unit called a "field", which corresponds to one machine word, typically 32 bits. The field is divided into two parts, namely, a tag part and address part. They correspond to the upper 4 bits and the lower 28 bits, respectively, in the case of a 32-bit machine word. The tag is what is called a "pointer tag". **Fig. 1** shows the tag part of PLisp data.

The address part consists of four data types, namely CONS data, block, symbol, and long number, and represents a storage address by the byte, in which its object is located. This is used as a pointer, and can provide 256 MB of

0001	CONS data
0011	block (vector, array, etc.)
0101	symbol
0111	long number (long integer etc.)
1x00	short real
1x01	short integer
1x10	character code
1x11	indirect pointer (unused)

Note: 'x' indicates a marking bit.

Fig. 1 Tag part of PLisp data representation.

the storage space. Each data object is located at a distance from a word boundary, so that the lower two bits are usually set to zero and one of them is used for marking at GC. The address part of three data types—namely, short floating-point number, short integer, and character code—represents their value as an immediate datum, using the full 28 bits. The second bit of the tag part is reserved for marking. Therefore no bits are available for other use.

3.2 A Fast and Bounded Workspace Algorithm

This subsection presents a fast and bounded workspace algorithm called Algorithm F, which is used in the process for adjusting the pointers of LLGC. The reason we use this newly invented algorithm is that PLisp provides no additional storage space nor any extra bit, as other effective algorithms[2),4)] require. However, it is worth pointing out that these algorithms can be also utilized for LLGC unless such a restriction exists. Since the purpose of this paper is not to give precise and full details of Algorithm F, the following is merely a summary of it.

The basic idea of Algorithm F is a mixed strategy consisting of two different schemes used for pointer adjustment: a fast scheme based on table search and a space efficient scheme based on list search (or R-list search in the terminology of Terashima and Goto.[2]

The former is very easy to implement, if a continuous area is provided for a table. The table is made after the marking process. As an entry, it contains each pair of an integer called the "offset register" and a bit string called the "mark bit map". An entry of the table corresponds to a portion of a storage space called a "subspace", so that its mark bit map represents all the mark bits of the subspace in one-to-one correspondence. All subspaces are disjoined, and form the total storage space. If the mark bit map is m bits in length, typically $m=32$, each subspace is made up of m fields. Each offset register is an accumulated value obtained by counting up the non-marked fields from one end of the storage space to the first field of the subspace (exclusive of the field itself) to which its entry corresponds. A non-marked field is a field that has not been marked at the marking process. For a given pointer p that points to any field, the number of non-marked fields counting from the one end of the storage space to p is given by adding an offset register of the entry to which p corresponds and the total number of 0 bits in its mark bit map that are bitwise inclusive ORed with $-2^{p \underline{mod} m}$. The number indicates the distance that an active data object part of which is located in p will move toward the one end of the storage space by the field, and is called an "offset value" with which p needs adjusting at the time of such a move. Obviously each pointer can be adjusted in a constant time independent of N, A, or S. This is not slower than Morris's algorithm , as shown by the experimental data in Terashima and Sato.[5]

Algorithm F is modified to use each table entry that has been constructed in its subspace itself rather than a continuous area of an additional storage space. A table entry requires two successive non-marked fields. Such entries may be able to be constructed in the subspace, because each PLisp data object is located in a storage space using at least two fields.

However, there may be one or more subspaces in which no entries can be constructed for lack of non-marked fields, while there is at least one cluster called an "entry-free" cluster in this subspace or subspaces. In such a case, another scheme is used. A list (or an R-list[2]) is made for entry-free cluster(s). Each node is located in two successive fields that follow each entry-free cluster, and contains the offset value for the cluster and a pointer to the next node. Therefore, the offset value of any entry-free cluster can be obtained by an R-list search comparing the two storage addresses of the cluster and its node. Obviously the time cost of the R-list search is proportional to its length, so that pointer adjustment based on this scheme may be costly. However, many experimental data obtained from the execution of the LLGC show that the length remains very small (not greater than 10) even if many data amounting to more than m fields per object are constructed intermittently. Therefore the time required for the pointer adjustment can be regarded as a constant per pointer.

A cluster called an "anchor" may exist on the border of the end of the storage space toward which active data objects move. Of course the anchor is not subject to relocation, and its offset value is zero. Therefore no table entry or R-list node is constructed for the anchor.

It is clear that the offset value of any pointer can be obtained by using a table entry or R-list. The table entry is checked first. If it does not exist, the R-list is searched, except for the anchor. Hereafter, we call these entries and the R-list simply the "offset table".

3.3 LLGC

LLGC performs the following process sequentially: marking, offset table making, pointer adjustment, and relocation. It is invoked when a storage space is exhausted except for the U-space, or when a function GC is called explicitly. Since active data objects move toward the bottom (lower address) of the storage space, the U-space is located at the top (higher address) of the storage space.

The size of the U-space is decided at run time, taking account of the load factor measured most recently, and is kept below $S/10$ for efficiency. This means that the U-space is always available when the load factor is less than 0.1, and LLGC shows its time economy in such a case. When the load factor is higher, LLGC provides no U-space, and proceeds to conventional sliding compaction GC.

3.3.1 Marking

The marking process marks all active data objects and stores the storage address of its first field in the U-space if this exists. The former is done by traversing list structures and scanning vectors and/or arrays from root(s), which is the same as the marking process of other GC methods, while the latter is specific to LLGC. List structures are traversed in "post order"; CAR, CDR, and then the nodal addresses are stored as a sequence of data. A sequence generated for most LIST data may be in order of increasing address, as if it has been sorted, because on the construction of data objects in Plisp, a specific order of data objects is established such that any son is allocated before its parent. Therefore, the post order seems to have a good effect on sorting.

The task of storing a storage address datum is performed within a constant time by the following procedure, written in C.

```
    void g_store(Object * obj)  {
        if (htp<ADmax_storage)
            * (htp++)=(Object)obj;
    }
```

where htp is a pointer to the U-space, and its initial value is the beginning address of U-space. The final address of the U-space is ADmax_storage. Notice that this procedure does nothing when the U-space overflows. It is clear that the time cost of the marking process is proportional to A.

3.3.2 Offset Table Making

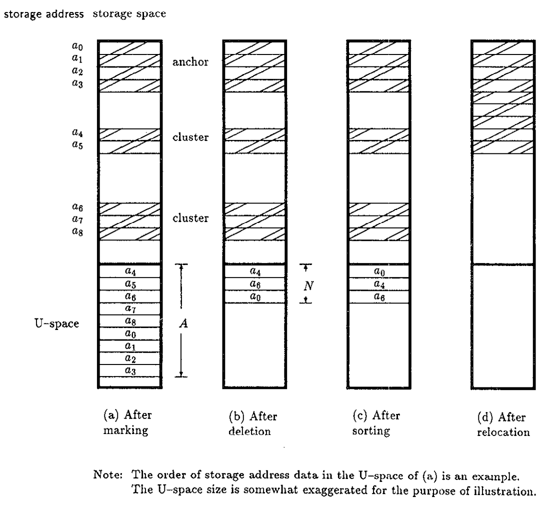
The offset table-making process sorts a set of storage address data stored in the U-space and makes an offset table by using the sorted data. Therefore, if the U-space overflows or is not provided, the former is omitted and an offset table will be made by scanning through the total storage space.

After the marking process, the U-space contains a set of storage address data that point to the first field of every active data object (see **Fig. 2** (a)). Then the quantity of these storage address data is reduced and a subset of them is so made that consists of data pointing to the first field of every cluster ; each datum is deleted if it points to a field following a marked field (see Fig. 2 (b)). This requires a time proportional to the size of the (original) set, or A symbolically. Consequently, the subset contains N storage address data of clusters, and these data are sorted. Of course it is sufficient to know all the active data objects.

Both quick sorting and insertion sorting are implemented as sorting methods. They generate sequences of sorted storage address data in times proportional to $N \log N$ and $N \times N$, respectively, though quick sorting requires an additional space of N implicitly in the worst case. The sorted data of the sequence correspond to clusters in order of increasing address (see Fig. 2 (c)).

The sequence in the U-space is used for skipping non-marked fields between clusters. The number of non-marked fields is needed for offset table making, but they do not need to be looked up. When a non-marked field is encountered after scanning of a cluster, the scanning is simply transferred to the following cluster pointed to by the next datum in the sequence. This allows the offset table to be made in a time proportional to A, not S, provided that such a sequence is available. Consequently, the time cost of this process is proportional to the sum of A and $Nf(N)$.

When the U-space is not available, the offset table is made by scanning the total storage space. This requires a time proportional to S. A chain called an "a-link" is also made simultaneously by using a pointer stored in a non-marked field between clusters, in order to connect them. It has the same effect on time economy as the sorted storage address data.



Note: The order of storage address data in the U-space of (a) is an example. The U-space size is somewhat exaggerated for the purpose of illustration.

Fig. 2 Action of LLGC.

3.3.3 Pointer Adjustment

The pointer adjustment process is carried out mainly by Algorithm F. Each pointer is checked to determine whether it points to an anchor before it is applied to Algorithm F. The anchor is free from relocation, and every pointer that points to the anchor needs not to be adjusted.

Every pointer of every active data object must be adjusted, and it turns out that only a constant time is required to find the offset value of each pointer by using Algorithm F. Therefore, the time cost of the pointer adjustment process is proportional to A by virtue of the sorted data or a-link.

3.3.4 Relocation

The relocation process is separated from the pointer adjustment process, because it may destroy an offset table needed by the latter. The relocation process relocates active data objects by unmarking them. It is performed within a constant time for each field, so that the time cost of the relocation process is proportional to A by virtue of the sorted data or a-link.

4. Analysis of LLGC

LLGC has been successfully implemented on PLisp, which runs on a workstation called Sony NWS-3460 (its CPU is a MIPS R3000). The test programs presented here are the Tarai and the Bit functions designed by I. Takeuchi and M. Nakanishi, respectively.[13] The former has been modified by the authors. The programs are named Modified Tarai-4 and BitA-8, and are shown in **Fig. 3** (a) and (b), respectively.

The Tarai function is often used as a simple

```
(DEFUN TARAI (X Y Z w) (prog2
       (setq w (list (cons 'X X) (cons 'Y Y) (cons 'Z Z) X Y Z))
       (COND ((GREATERP X Y)
         (TARAI
            (TARAI (SUB1 X) Y Z ())
            (TARAI (SUB1 Y) Z X ())
            (TARAI (SUB1 Z) X Y ())
            ()))
         (T Y))))

(TARAI 8 4 0 ())
```

Fig. 3 (a) Test program (Modified Tarai-4).

```
(DEFUN BITA (A) (COND
       ((NULL (CDR A)) A)
       ((NULL (CDDR A)) (LIST (CONS (CAR A) (CONS '$ (CDR A)))))
       (T (BIT1 (CDR A) (LIST (CAR A))))) )
(DEFUN BIT1 (X J) (COND
       ((NULL X) NIL)
       (T (NCONC (MAPAPPEND (BITA X)
             (FUNCTION (LAMBDA (K) (MAPCAR (BITA J)
                (FUNCTION (LAMBDA (L) (LIST L '$ K))) ))))
             (BIT1 (CDR X) (APPEND J (LIST (CAR X))))))) ) )
(DEFUN MAPAPPEND (X F) (COND
       ((NULL X) NIL)
       (T (NCONC (F (CAR X)) (MAPAPPEND (CDR X) F)))))

(BITA '(a b c d e f g h))
```

Fig. 3 (b) Test program (BitA-8).

Table 1 GC processing time.

Program	Modified Tarai-4			
Storage space	Load factor (average)		Total GC time (sec.)	
K fields	Conventional	Sort version	Conventional	Sort version
1.2	0.475	0.569	3.99	5.53
1.6	0.354	0.403	2.40	2.85
2.0	0.278	0.318	1.78	2.07
2.4	0.232	0.261	1.53	1.51
2.8	0.203	0.217	1.25	1.08
3.2	0.177	0.187	1.24	0.97
3.6	0.155	0.167	0.96	0.88
4.0	0.140	0.148	0.95	0.76

Program	BitA-8			
Storage space	Load factor (average)		Total GC time (sec.)	
K fields	Conventional	Sort version	Conventional	Sort version
18	0.709	—	0.36	—
20	0.627	—	0.26	—
25	0.503	0.510	0.17	0.48
30	0.500	0.505	0.23	0.32
31	0.499	0.450	0.22	0.21

benchmark for estimating the efficiency of function calls and recursion. It does little but wide circulation on function call with a small number of nests. Modified Tarai-4 acts as if the Tarai function is being executed on a LISP 1.5 interpreter.[1] It consumes much storage space in making both an association list and EVLIS's lists. BitA-8 also consumes much storage space in making binary trees and *funarg* closures, even if it is compiled.

Table 1 shows the total GC time of these two test programs, which is measured by LLGC in two modes that use varying storage space sizes: "conventional" LLGC performs conventional sliding compaction GC using no U-space, and "sort version" LLGC is guaranteed to use

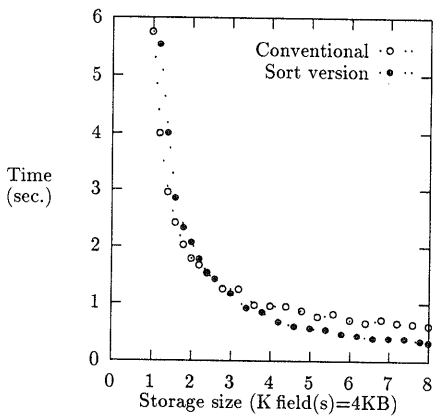


Fig. 4 (a) Total GC time.

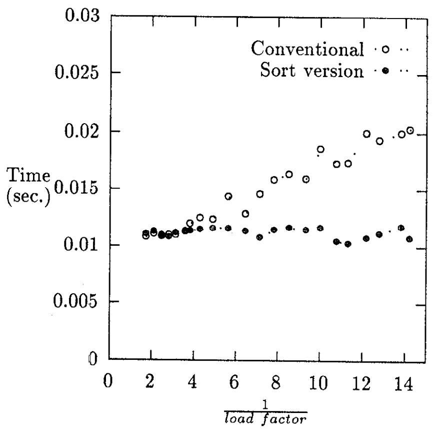


Fig. 4 (b) Average GC time.

the U-space. The latter uses insertion sorting, and the time cost of the sorting will be the worst. Notice that the storage space totally includes the U-space. This is why the load factors of the two differ for the same storage size. **Fig. 4**(a) shows the total GC time of Modified Tarai-4 graphically.

It is clear that the sort version is superior to the other in time if the total storage space size is relatively large. Fig. 4(b) shows this more visually. Variations of the average GC time per invocation are plotted for the execution of Modified Tarai-4. The average time of the sort version LLGC remains nearly level, because A varies by only small amounts such as 550 ± 100 fields at each invocation. On the other hand, the average time of the conventional LLGC seems to be proportional to $1/a$. Notice that this is a unit of the horizontal axis. Since $A/a=S$ is always satisfied, the average time is proportional to S, if A is fixed. This proves the well-known fact that conventional sliding compaction GC requires a time proportional to S.

However, the sort version is inferior when the total storage space size becomes small. The balance point of the load factor at which the two versions are nearly equal in their GC time is about 0.26. Our strategy of keeping the size of the U-space less than $S/10$ seems to be good in both time and space. These two programs are typical of many test programs being executed that all show similar effects.

5. Concluding Remarks

We introduced a new type of GC called LLGC. This has the great advantage of time economy, which conventional sliding compaction GC has never achieved. It requires a GC time far less than the conventional sliding compaction GC when the load factor is small. Therefore, it will be suitable for large storage use such as the 256 MB provided by PLisp.

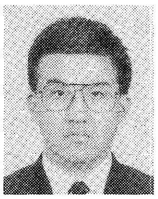
The time efficiency of LLGC largely depends on the cost of sorting, though a sequence of storage address data seems to have a great many sub-sequences that have been naturally sorted.

LLGC makes no use of effective algorithms such as Morris's, but they can be applied to it. Sliding compaction GC generates an anchor that is free from relocation. It would be interesting to design another type of LLGC that processes the anchor exclusively.

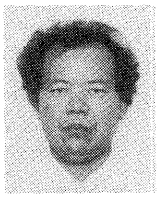
Acknowledgement The authors thank the anonymous referees for their valuable suggestions.

References

1) McCarthy, J. et al.: *LISP 1.5 Programmers Manual,* MIT Press, Mass. (1962).
2) Terashima, M. and Goto, E.: Genetic Order and Compactifying Garbage Collectors, *Inf. Process. Lett.,* Vol. 7, No. 1, pp. 27-32 (1978).
3) Appleby, K., Carlsson, M., Haridi, S. and Sahlin, D.: Garbage Collection for Prolog Based on Wam, *Comm. ACM,* Vol. 31, No. 6, pp. 719-741 (1988).
4) Morris. F. L.: Time- and Space- Efficient Garbage Collection Algorithm, *Comm. ACM,* Vol. 21, No. 8, pp. 662-665 (1978).
5) Terashima, M. and Sato, K.: A Garbage Collector Efficient for Varisized Cells (in Japanese), *Trans. IPS Japan,* Vol. 30, No. 9, pp. 1189-1199 (Sep. 1989).
6) Fenichel, R. R. and Yochelson, J. C.: A Lisp Garbage Collector for Virtual Memory Computer Systems, *Comm. ACM,* Vol. 12, No. 11, pp. 611-612 (Nov. 1969).
7) ed. Bobrow, D. J.: *Symbol Manipulation Languages and Techniques,* North Holland, Amsterdam (1971).
8) Cohen, J.: Garbage Collection of Linked Data Structures, *ACM Comput. Surv.,* Vol. 13, No. 7, pp. 341-367 (1981).
9) Terashima, M.: A Note on Time- and Space-Efficient Garbage Collection (in Japanese), *Preprint of WGSYM (IPSJ)*, 68-5 (Mar. 1993).
10) Koide, H.: A Hybrid Garbage Collection, Master's thesis, Department of Computer Science, University of Electro-Communications, Tokyo, Japan (1993).
11) Koide, H. and Noshita, K.: On the Copying Garbage Collector Which Preserves the Generated Order (in Japanese), *Trans. IPS Japan,* Vol. 34, No. 11, pp. 2395-2400 (1993).
12) Steele, G. L. Jr.: *Common LISP, 2nd ed.,* Digital Press, Mass. (1990).
13) Takeuchi, I.: The 2nd Lisp Contest, *Preprint of WGSYM (IPSJ)*, 5-3 (Aug. 1978).



Mitsugu Suzuki was born in Tokyo, Japan on August 1, 1964. He received the B.E. degree and the M.E. degree in computer science from the University of Electro-Communications, Tokyo, Japan, in 1989 and 1991 respectively. He is now a doctoral student in computer science at the University of Electro-Communications. His current research interests include storage management and parallel algorithms.



Motoaki Terashima was born in Shizuoka, Japan on June 8, 1948. He received the degrees of B.Sc. (1973), M.Sc. (1975) and D.Sc. (1978) in Physics from the University of Tokyo for his work on computer science. Since 1978, he has been a research associate of the Department of Computer Science, University of Electro-Communications, and is currently an associate professor of Graduate School of Information Systems. He was a visiting scholar at the Computer Laboratory of the Cambridge University in 1992. His current research interests include programming language design and implementations, memory management, and symbolic and algebraic manipulation systems. He is a member of IPSJ, ACM and AAAI.