KUIS-95-0008

Depth-First Copying Garbage Collection without Extra Stack Space

Hiroshi Nakashima[†] and Takashi Chikayama[‡]

[†] Dept. of Information Science, Kyoto University
[‡] ICOT Research Center

March, 1995

Abstract

In this paper, we propose two copying garbage collection methods. These two methods copy data structures in depth-first order, while conventional method copies in breadth-first order. This modification greatly improves memory access locality during both garbage collection and computation after. Restriction on data structure representation is also relaxed because the proposed methods don't require that a data object itself indicates whether it is a pointer.

For the depth-first copying, it is necessary to memorize information about non-leaf data structures so that their unprocessed elements are visited afterward. The first proposed method, the *link* method, links data structures that have unprocessed elements. Another method, the *reservation-stack* method, pushes each unprocessed element being a pointer to another structure to a stack allocated at the end of the area to which data objects are copied. Therefore, neither of methods requires extra stack space.

We evaluated the performance of the proposed methods together with the conventional method. The result shows that the proposed methods have a great advantage over the conventional method when data objects are distributed in a large memory space, because the number of page faults are drastically reduced.

---

* Original Japanese version is in Trans. Information Processing Society of Japan, Vol. 36, No. 3, March 1995.

† nakasima@kuis.kyoto-u.ac.jp

1 Introduction

Automatic memory management is an indispensable function of high level language systems, especially those for symbolic processing. It undertakes complicated tasks of memory allocation and reclamation in order that programmers concentrate on their essential part of problem solving.

A key technology of memory management is garbage collection that reclaims memory area for data no longer accessed. One of well known and well used garbage collection method is *copying* collection with two memory area. This method copies active data from an area to another without accessing garbages. Therefore, copying collection is more efficient than those that scan the whole of memory area, especially when the amount of active data is small. Another good feature is to gather active data packing them into a continuous memory space, making data allocation easy and working set size small after collection. Implementation is also easy due to its simple algorithm. These advantages overcome the drawback that two memory area consume address space twice.

Conventional copying collection methods, such as that originally proposed in [1] and its modified versions [2, 3], however, has another drawback. These methods copies and lines nested data structure up in breadth-first order. Since programs usually generate and process these data in depth-first order, breadth-first copying causes large working set size not only at the copying but also after collection.

In order to solve this problem, this paper proposes two depth-first copying collection methods, named *link* method and *reservation-stack* method. For the depth-first copying, it is necessary to memorize information about non-leaf data structures so that their unprocessed elements are visited afterward. Link method links data structures that have unprocessed elements, while reservation-stack method push each element pointing another structure into a stack allocated on the end of the area to which data objects are copied. Therefore neither of methods requires extra stack spaces, data tags nor marking bits. Both methods has same computation complexity as conventional ones. Moreover, since they access a data structure through the pointer to it, the structure may not indicate its attribute by itself.

This paper is organized as follows: Section 2 summarizes the conventional copying method and its merits and demerits. Section 3 describes the proposed methods in detail. Section 4 discusses the performance of the proposed and conventional methods. After a brief discussion on related works in Section 5, the conclusion is given in Section 6.

2 Conventional Copying Collection Method

This section summarizes the conventional copying method to clarify the background of our research.

2.1 Collection Procedure

The basic procedure of copying garbage collection is outlined as follows.

(1) Divide usable memory space into two areas of same size.

(2) Allocate data generated in program execution on an area, while keep another area empty.

(3) When *old-area* used for data allocation becomes full, copy active data in the area to another *new-area* that will have free continuous space for further data allocation.

(4) Exchange the roles of two areas.

This outline of the basic procedure is common to the conventional and proposed methods. Differences is in the copying procedure of the step (3).

In the conventional method, this step is based on Cheney's algorithm summarized as follows. (See Appendix A.2 for detail.) In the following description, it is assumed that a memory area or a data structure consists of continuous memory words. *Head* of a data structure is the word having the least address in the data, while *tail* is that having the greatest. The terms head and tail are also used to specify the words of a memory (sub-)area in the same sense. The word pointed by a pointer p is represent by $*p$.

[B1] Copy roots

Set the pointer *new_alloc*, which points the head of free space of new-area, to the base (i.e. head) of new-area. Then for all roots, which are primarily known as active pointers, set pointer *new* to each address and execute the procedure [B3]. After that, set *new* to the base of new-area.

[B2] Scan new-area
Execute the procedure [B3] incrementing *new* while *new* is less than *new_alloc*.

[B3] Copying procedure

[B3-1] Examine pointer
Exit if $*new$ is not a pointer.

[B3-2] Examine copied
Let S be the data structure pointed by $*new$. If the head of S is a pointer to a word in new-area, modify $*new$ so that it points the destination of S and exit, because S has been already copied.

[B3-3] Copy data structure
Copy S to the region from *new_alloc* and modify $*new$ so that it points the destination of S (i.e. *new_alloc*).

[B3-4] Indicate copied
Store a pointer to the destination (*new_alloc*) into the head of S to indicate S has been copied. Then add the size of S to *new_alloc*.

It should be noted that data structures area copied in breadth-first order because this procedure uses the region between *new* and *new_alloc* as a FIFO queue containing data structures having unprocessed elements. Also note that this FIFO queue is sequentially scanned in [B2] incrementing *new*, and each word in the queue is examined whether it is pointer.

In addition to the assumption shown before, the following are assumed for a data structure S.

(a1) The size of S is known by examining S and the pointer to it.

(a2) The location of the word that indicates whether S is copied is known by examining S and the pointer to it. This word cannot have inherent values that may be interpreted as a pointer to new-area.

(a3) The word of (a2) is the head of S.

(a4) S does not have sub-structures.

The assumptions (a1) and (a2) are indispensable and reasonable irrespective of variation of copying methods. The assumption (a3) is introduced only to simplify explanations and is easily removed to cope with general cases. On the other hand, removing (a4) is so complicated that limited space of this paper makes the discussion on it hard. However, it can be removed by introducing;

- *transparent* pointers such as variable reference pointers used in logic programming language systems, and
- a special tag to indicate the head of a copied data structure

as described in [4].

2.2 Features of Conventional Method

The conventional copying method has the following advantages.

(+1) Since garbages are not accessed in collection, the copying method is more efficient than others that scan the whole of memory area, especially when the amount of active data is small.

(+2) Active data are packed into a continuous memory space. This feature is advantageous over other methods that leave active data unmoved, because;

- data allocation after collection is easy because of continuous free memory space, and;
- working set size for access to data after collection is smaller.

(+3) Its implementation is easy because of its simple algorithm.

The disadvantage common to copying methods is that two memory area consume address space twice. Besides it, the conventional methods inherently has the following other disadvantages.

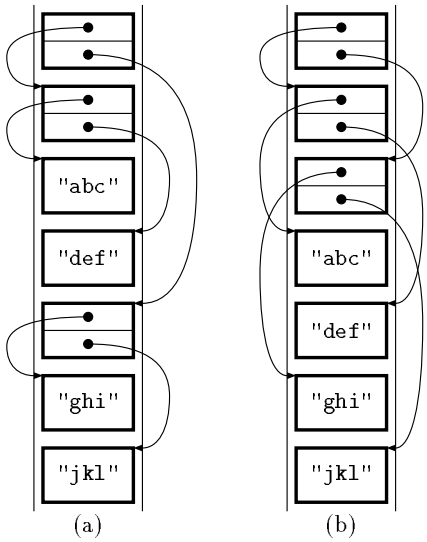

Figure 1: Copying a binary tree in the conventional method

(−1) Nested structures are copied in breadth-first order, while usual programs access them in depth-first order. For example, let us consider the following Prolog program.

```
tree([L|R]):- left(L), right(R).
left(["abc" | "def"]).
right(["def" | "ghi"]).
```

This will make a binary tree;

```
[["abc" | "def"] | ["ghi" | "jkl"]]
```

that has four character strings as its leaves. The cons cells of the tree will be lined up as shown in Figure 1(a)[1]. On the other hand, breadth-first copying of the tree will lines up it as shown in Figure 1(b). This feature is inferior to that of other methods, e.g. sliding-compaction, which keep the original order of active data with respect to the locality of access when;

- a data structure generated in depth-first order is copied in breadth-first order, and when;
- a data structure copied in breadth-first order is accessed in depth-first order after collection.

(−2) Since a data structure that has to be copied is found by scanning new-area linearly and blindly, data representation must make it possible to distinguish pointers from others. This means that a word should be identified as a pointer or not by examine only the data structure containg the word. This condition restricts the freedom of data representation, excluding an efficient design in which only the pointer expresses what the data structure pointed is, because the types of its contents cannot be known in collection.

For example, if we want to represent a character string `"abc"` by its length and a raw sequence of character codes, it is necessary to prevent from confound the raw sequence with a pointer. One way to do this is to attach a tag to the head of a data structure to express the data type as shown in Figure 2(a). This *object tag* helps garbage collector to find that two pointers in a cons cell should be processed in collection, or the contents of a character string should be skipped. The object tag, however, must be looked up when one want to examine the type of data pointed by a pointer. On the other hand, *pointer tag* shown in Figure 2(b) is more efficient than the object tag, because it does not require to access the data structure on the data type examination. However, this efficient way is not applicable when the conventional copying method is employed.

[1] To simplify the explanation, we assume that a character string is represented by a special data structure, while usual Prolog system represent it by a list of character codes.

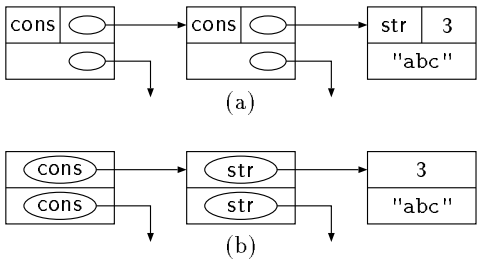

Figure 2: Object tag and pointer tag

3 Depth-first Copying Collection Methods

The drawbacks of the conventional method dicussed in the previous section are strongly due to that new-area is used as a FIFO queue to keep pointers to uncopied nested data structures. Therefore, if we have a LIFO stack for such data structures, they will be copied in depth-first order without blind scan of new-area, getting rid of the drawbacks.

Let S be a *non-terminal structure* that has multiple elements that may be pointers to other data structures. During S is processed, an element e of S, which is a pointer to other non-terminal structure T, will be found. At this time, since depth-first copying strategy requires to process T, the following context, for example, should be saved into the stack.

(1) the pointer word that points S

(2) the information to know the element next to e (e.g. the offset from the head of S to the next element)

When T is completely processed, the stack top will be popped to process S again from the element next to e. At this resumption, the size of S and its destination will be obtained by examing the saved pointer to S. The completion of the copying process originated by a root is detected when the stack becomes empty.

The kernel part of this method using an extra stack space, named *naive-stack method*, is as follows. (See Appendix A.3 for datail.)

[D1] Variables keeping context
When a non-last element e of a non-terminal structure S is processed, the context is represented by the pointer variable *pword* to S and the variable *offset* from the head of S to e.

[D2] Other variables
The pointer *new_alloc* points the head of free space of new-area. The variable *word* has the value of a word in old-area, which is beeing copied to the address *new*.

[D3] Examine data type
The following are performed depeding on the data type of *word*.

[D3-1] Copy non-pointer
If the *word* does not have a pointer, store *word* into $*new$ and go to [D4].

[D3-2] Examine copied
Let T be the data structure pointed by *word*. If the head of T is a pointer to a word in new-area, modify $*new$ so that it points the destination of S and go to [D4], because S has been already copied.

[D3-3] Copy header
Store a pointer to $*new$ so that it has the same data type as *word* and points the same address as *new_alloc*. If T has a header, which consists of consecutive words known as non-pointers, copy it to the region from *new_alloc*. Then set *word* to the word next to the header, store a pointer to the destination of T into the head of T. After that, reserve the destination region for T in new-area by;

$$new \leftarrow new\_tail + \langle \text{size of } T\text{'s header} \rangle \ ;$$
$$new\_alloc \leftarrow new\_alloc + \langle \text{size of } T \rangle \ ;$$

Then let n be the size of T minus the size of its header.

[D3-4] Header only structure
If n is 0, go to [D4], because T does not have any pointers.

[D3-5] One element structure
If n is 1, go back to [D3], because the context of T will not need to be saved.

[D3-6] Non-terminal structure
If n is greater than 2, push *word* and *offset* + *1* into the stack. Then change the context to that for T by;

$$pword \leftarrow word \ ; \quad offset \leftarrow \langle \text{size of } T\text{'s header} \rangle \ ;$$

and go back to [D3].

[D4] Get next element

[D4-1] Examine stack empty
If stack is empty, the copying process originated by a root is completed. Otherwise, proceed to the following.

[D4-2] Set next element
Increment *offset* by 1, obtain the value of next element and its destination of copy from *pword* and *offset*, and set *word* and *new* to them.

[D4-3] Examine tail
If the element is the last one, pop the stack top to restore the context to *pword* and *offset*.

[D4-4] Process next element
Go to [D3] to process the element.

For simplicity of the explanation, the algorithm shown above assumes the following for a data structure S.

(a5) S may have a header containing non-pointer words only. The size of the header is known by examining S and the pointer to it.

(a6) Each word excluded from the header may be a pointer and it is known by examining only the word whether or not the word is pointer.

These assumptions, however, is not essentially required for naive-stack method. Instead, if it is known whether a word w of S is pointer by examining the pointer to S and the offset to w, the method will work.

In naive-stack method, when a pointer to a non-terminal structure T is found as an element of a non-terminal structure S, [D3-6] pushes the context of S into the stack and starts the process for T changing the context to that for T. On the other hand, if the element is a non-pointer, a pointer to a data structure which has been or is beeing copied, or a pointer to a data structure having non-pointers only, [D4] proceed to the next element of S. If the next element is the last one, however, the stack is popped to restore the context of S's parent because the context of S is no longer necessary. This is a kind of tail recursion, and is also applied for the structure having only one element, whose context is not saved.

The naive stack method has an obvious drawback that the size of stack may be same as that of old-area in worst case. This makes it more serious that copying methods need larger address space than others. Therefore, we propose two depth-first copying methods that does not require extra memory space.

3.1 Link Method

If the assumption (a5) and (a6) hold, the following are sufficient to represent the context at the time when an element e of a non-terminal structure S is processed.

(1) old-area address of e (*old_link*)

(2) new-area address of e (*new_link*)

(3) number of unprocessed elements of S

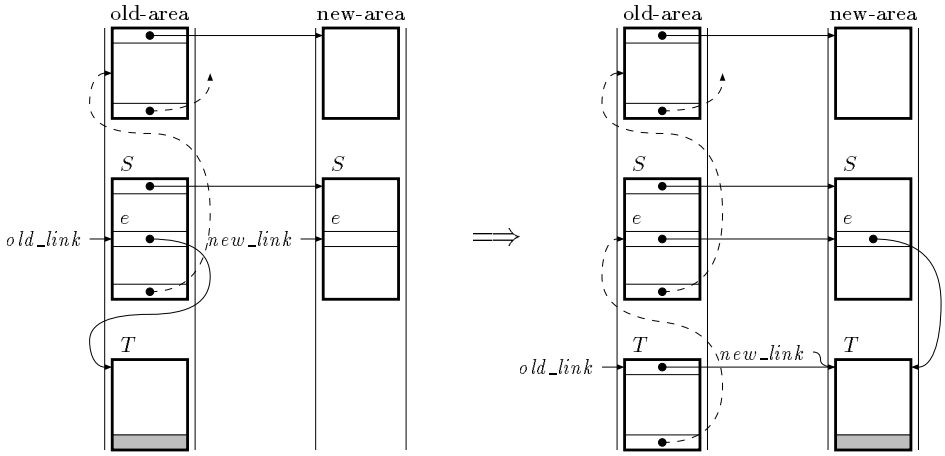

Figure 3: Links for non-terminal data structures.

In link method, the contexts are saved in non-terminal structures that are partially processed by linking them as shown in Figure3. This technique makes it possible to copy data structures in depth-first order without stack. The context variables *old_link* and *new_link* are saved by the following procedure that replaces the step [D3-6] of naive-stack method. (See Appendix A.4 for detail.)

[D3-6′] Non-terminal structure

Let T' be the region of new-area, to which T is copied. Save the value of the tail word of T into the tail of T', then save *old_link* into the tail of T. Save *new_link* into $*$*old_link* where e is located. This saving operation is safe because the value of e has been already moved into *word*. Then set *old_link* and *new_link* to the address next to the header of T and T', respectively.

The number of unprocessed elements, on the other hand, is not saved in raw form because it is enough that we can detect the last element of S, which is *old_link* to the parent of S. In order that the last element acts as the sentinel without introducing any special tags nor additional bits, the old-area address contained in *old_link* is converted to a new-area address *old_link′* by and saved in the tail attaching an appropriate tag that represents a pointer. This old-to-new conversion is performed by

$$old\_link' = old\_link - old\_base + new\_base$$

where *old_base* and *new_base* are the bases of old- and new-area, respectively. This conversion makes the last element pretend to have a pointer to new-area. Such a pointer, however, must not be found in a non-last element because of assumption (a4) shown in Section 2.1. Thus we have the procedure to replace the step [D4-2] and 3 as follows. (See Appendix A.4 for detail.)

[D4-2′] Set next element

Increment *old_link* and *new_link* by 1, and set *word* to the value of the next element, $*$*old_link*. Set *new* to the destination of the element, *new_link*.

[D4-3′] Examine tail

If *word* has a pointer to new-area, restore the context because the element is the last one. It is performed by;

$$old\_link \leftarrow \langle \text{old-area address converted from } word \rangle \ ;$$
$$word \leftarrow *new\_link \ ;$$

Stack emptiness will be detected when it is found that *old_link* has its initial value, an null pointer, which can be any address other than that in old-area. This null pointer representation, however, requires some special treatments when we save and restore *new_link* and examine whether an element is the last one. On the other hand, if we reserve an unused word in old-area and its counterpart in new-area, their addresses are utilized as null pointers eliminating those special treatments.

3.2 Reservation-Stack Method

When a pointer to a non-terminal structure T is found in a non-terminal structure S, naive-stack method saves the context of S into the stack in order to switch the copying procedure from S to T. Reservation-stack method that we propose here, however, does not postpone processing S, but saves the context of T into the stack to start the procedure for T after S is completely processed. This stack, named *reservation stack* because copying T is *reserved*, is placed an end of new-area opposite to the base and will grow in opposite direction that *new_alloc* goes. Therefore, if we can assure that the stack top should not collide against *new_alloc*, no extra space for the stack is needed.

The collision should not occur if the context to be saved in the stack consists of two or less words for a non-terminal structure and it is saved only once. This claim is proved as follows. Let N be the set of all data structures to be copied. Also let N' and N'' be the set of data structures that has already been copied and reserved at a point of the collection, respectively. Since N', $N'' \subseteq N$ and $N' \cup N'' = \emptyset$, the total word sizes of structures in each set, $W(N)$, $W(N')$ and $W(N'')$, satisfy the following equation

$$A \geq W(N) \geq W(N') + W(N'')$$

where A is the size of new-area. Since a non-terminal structure has two or more words, the size σ of the stack that contains the contexts structures in N'' is not greater than $W(N'')$. Therefore the following is satisfied proving the claim.

$$A \geq W(N') + \sigma$$

The context of T consists of the following two words.

(1) destination address of the pointer t to T

(2) value of the head word of T

The destination address of t is necessary because the destination of T is unknown when the context is saved. That is, t is simply copied to new-area when T is found, and will be modified to point the destination of T when T is copied afterward. Note that it is not necessary to save the old-area address of T because t has the address.

Also note that if other pointers to T are found before T is actually copied, they are chained so that the context points the head of the chain and the tail of the chain points T. For example, assume that T is reserved when a pointer t to T is found, and then another pointer t' to T is found afterward. Since t and t' should be modified when T is copied, the following chain

$$\mathit{stack} \rightarrow \langle \text{destination of } t' \rangle \rightarrow \langle \text{destination of } t \rangle \rightarrow T$$

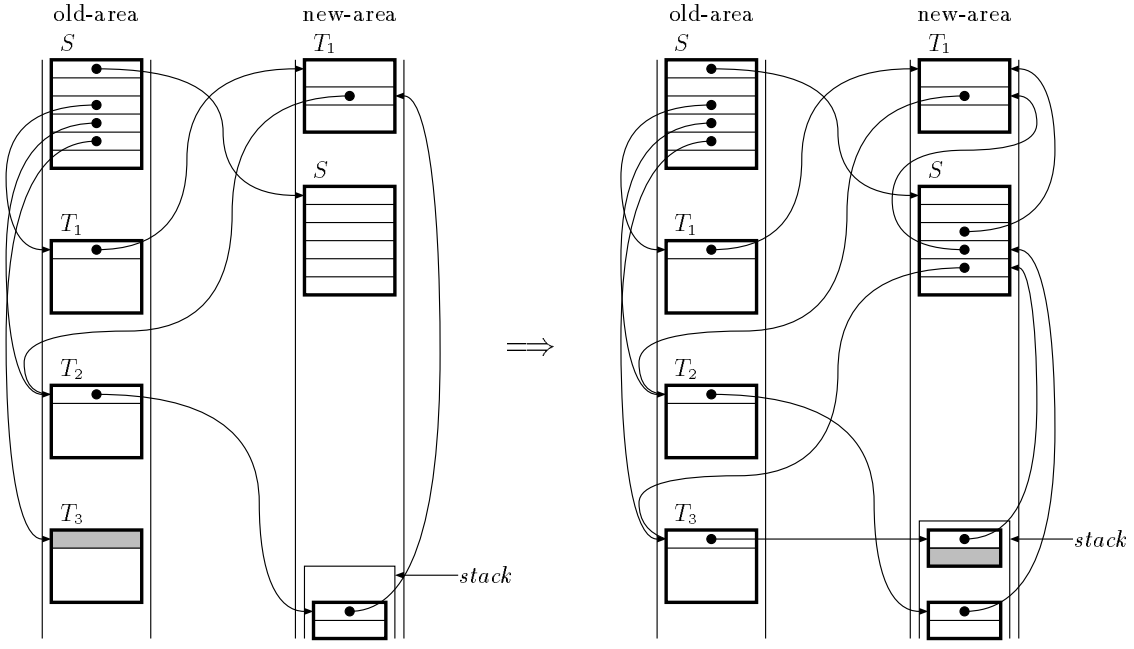

Figure 4: Reservation stack

is constructed by inserting destination of t' before that of t. The end of the chain, the pointer to T in old-area, is easily detected because other elements points new-area or the area for roots.

The head word is saved in order to avoid multiple reservation. That is, when T is reserved, its head word replaced with the pointer to the context on the stack. Since the stack is in new-area, the head word h of a non-terminal structure indicates copied, reserved or unprocessed as follows (Figure 4).

- copied if h is a pointer to new-area but not to the stack (T_1)
- reserved if h is a pointer to the stack (T_2)
- unprocessed if h is not a pointer to new-area (T_3)

The discussion above leads us to the following procedure performed when a pointer *word* to a data structure T having two or more words is copied to the address *new* in new-area.

[R1] Copied
If T has been copied, store a pointer, which points the destination of T and has the same type as *word*, into $*new$.

[R2] Reserved
If T has been reserved, let *sentry* be the value of T's head word, which is the stack address where the context of T is saved. Insert the destination of *word*, which is *new*, by the following operations.

$$*new \leftarrow *sentry \ ; \quad *sentry \leftarrow new \ ;$$

[R3] Unprocessed
If T has not been copied nor reserved yet, push *new* and the head word of T into the stack. Then store *word* into $*new$ and store the stack top address into the head word.

If *word* is not a pointer or a pointer to a one word structure, the procedure is basically same as that of naive-stack method.

When all elements in a data structure are processed, the following is performed.

[R4] Examine stack empty
If stack is empty, the copying process originated by a root is completed. Otherwise, proceed to the following.

[R5] Pop stack
Pop the top of the stack to obtain the head of the chain of pointers to reserved data structure T and its head word.

[R6] Modify pointer chain
Modify all the pointers in the chain so that they points the destination of T (*new_alloc*), traversing the chain until a pointer to old-area, which is the pointer to T, is found.

[R7] Start copy
Store the destination address of T (*new_tail*) into the head word of T to indicate T has been copied. Reserve the destination region for T in new-area by;

$$new \leftarrow new\_alloc \ ;$$
$$new\_alloc \leftarrow new\_alloc + \langle \text{size of } T \rangle \ ;$$

Then start copying T from its head word obtained from the stack top.

Appendix A.5 shows the detail of the algorithm shown above. Note that although the algorithm assumes that the assumption (a5) and (a6) shown in the previous section are hold, they are not essentially required. Also note that the order of processing elements of a data structure may be either ascending as the algorithm in appendix follows, or descending in order to copy nested structures left-to-right.

4 Performance Evaluation

This section shows the performance of proposed two methods, link method (LINK) and reservation-stack method (RSVS) together with that of conventional (CONV), with respect to the number of memory accesses, the execution time with small to medium scale memory space, and the number of page faults with large scale memory space.

4.1 Number of Memory Accesses

CONV copies a data structure from old-area to new-area, and then processes its elements scanning the new-area. Therefore, an element of the data structure, which may be a pointer, is accessed three times. That is, it is accessed twice at copying, one read in old-area and one write in new-area, and once when new-area is scanned. On the other hand, since proposed two methods dose not scan new-area, number of memory accesses for each element is two, one for read in old-area and another for write in new-area. Therefore, if a data structure has enough many elements, proposed methods are faster than CONV.

The actual number of accesses depends on methods because certain number is required for a data structure independent of the number of its elements. For example, the constant number of accesses for a data structure that is pointed by only one pointer and has two or more pointer elements varies according to methods as follows.

CONV $= 2$

- write the head word to indicate already copied (1)
- write the pointer to modify so that it points the destination (1)

LINK $= 7$

- write the head word to indicate already copied (1)
- read and write to save and restore the tail word (2)
- read and write to save and restore the *old_link* (2)
- read and write to save and restore the *new_link* (2)

RSVS $= 8$

- write the head word to indicate already copied (1)
- write the head word to indicate already reserved (1)
- write and read of two-word context on the stack (4)
- read and write the pointer to modify so that it points the destination (2)

Therefore, number of memory accesses for a n-word data structure whose elements may pointers is as follows.

$$\begin{aligned} CONV &= 3n + 2 \\ LINK &= 2n + 7 \\ RSVS &= 2n + 8 \end{aligned}$$

Thus LINK will require less memory accesses than CONV when $n > 5$, while RSVS will be so when $n > 6$.

4.2 Execution Time

The performance of the garbage collection is not only depends on the number of memory accesses, but also on the complexity of the algorithm and access locality. It is expected that proposed methods are more complicated than CONV, while they will show better locality. We developed the garbage collection program for each method and measured the execution time of each in order to evaluate overall performance.

The data structures to be copied on the measurement are n-ary balanced tree where $n = 2^k$. Each node of the tree is an data structure that has n elements and consists of $n + 1$ words including one-word header indicating the number of elements. Copying this header requires three additional memory accesses in CONV method, while two in proposed ones. The header is exceptionally omitted for a two-element node, which is *cons* cell usually represented as a two-word structure in Lisp and Prolog systems. Thus the pointer to a cons cell has a type different from that to a general n-element structure. A element of a structure is a pointer to another structure in case of non-leaf node, while an atomic non-pointer word if leaf. One word consists of 32 bits, whose least significant 2-bit is data type tag and another 30-bit is for an address or an atomic value.

The algorithms are basically those that described Section 2 and 3. As for LINK and RSVS, however, tail recursion optimization is introduced to remove redundant memory accesses. This optimization

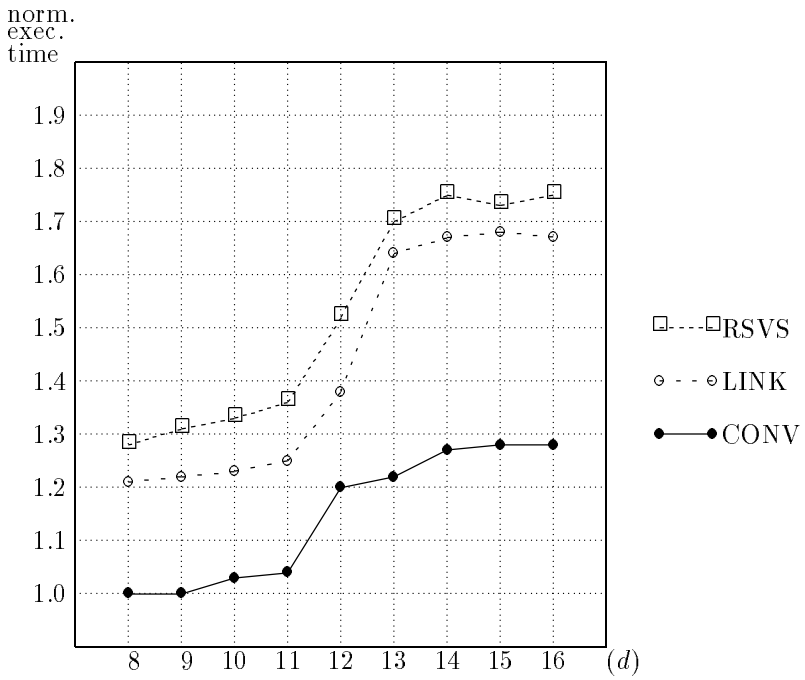

Figure 5: Execution time for binary trees

reduces average number of accesses for n-ary tree. It approximates the following when $n > 2$.

$$\begin{aligned} LINK &= 2n + 7 + 2(n-1)/n \\ RSVS &= 2n + 3 + 7(n-1)/n \end{aligned}$$

In case of cons cells, the number is approximately 9.5 for LINK and 8.5 for RSVS. In addition, RSVS processes elements in descending order for left-to-right traverse.

The program does not only consists of garbage collection but also has a procedure that traverse the tree in depth-first left-to-right order. This procedure is equivalent to the Prolog program;

```
traverse([L1|R1],[L2|R2]):- !,
    traverse(L1,L2), traverse(R1,R2).
traverse(T,T).
```

in case of $n = 2$. Measured execution time includes that of this procedure.

Programs are written in C, are compiled by gcc version 2.4.5 with the compile option "`-O2 -fomit-frame-pointer`", and are executed on SparcStation1 that has 16 MB main memory and 64 KB cache.

Figure 5 shows normalized execution times for binary trees whose depth d is 8 to 16. The normalized time $t_2(d)$ is obtained by;

$$t_2(d) = \frac{\tau_2(d)}{\tau_2(8) \cdot W_2(d)}$$

where $\tau_2(d)$ and $W_2(d)$ is the raw execution time and total number of words of binary tree of depth d. Note that d cannot be greater than 16 because of many page faults of CONV method. The figure shows that the performance of proposed methods is a little bit degraded, about 20 % in LINK and 30 % in RSVS when d is small. The main reason of the degradation in LINK is the increase of memory accesses, 1.5 times or 18.8 % for a node. On the other hand, the degradation in RSVS is mainly due to its complicatedness because the increase of accesses is only 0.5 times or 6.3 %. In all methods, performance is abruptly degrade at $d = 12$ because of the cache overflow. The reason why CONV method shows better tolerance of the overflow than proposed is less cache miss ratio due to the pre-fetch effect on scanning.

Figure 6(a) shows the performance measured varying n with the greatest d that CONV method can cope with. The execution time is normalized using $\tau_2(16)$ of CONV method as the reference point. The figure makes it clear that the performance of proposed methods becomes better than CONV as n increases. This is mainly due to the number of memory accesses, as shown in Figure 6(b) in which the number is normalized using that of $n = 2$ and $d = 16$ of CONV method as the reference point.

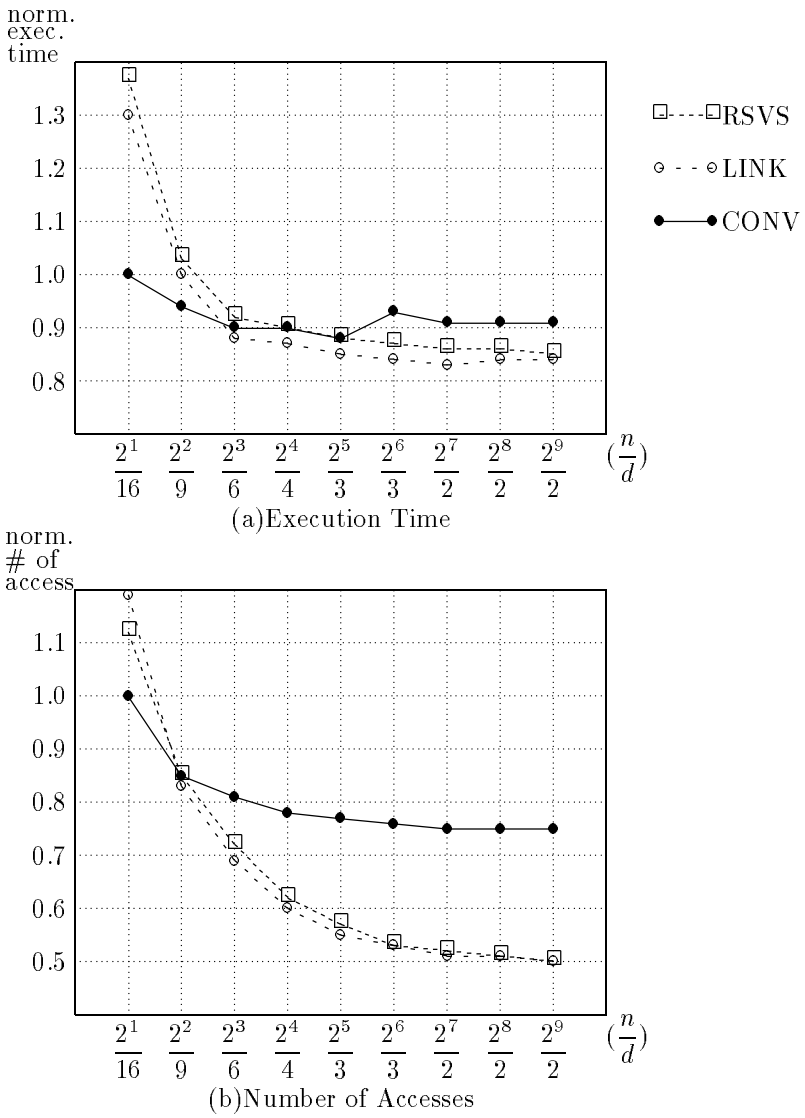

Figure 6: Execution time and number of memory accesses for n-ary trees

Since it is expected that the whole of a tree is in real memory space, bad access locality of CONV method will not seriously affect the measurement results. To confirm it, we also measured the performance for linear list having 2^n $(8 \leq n \leq 16)$ elements whose car parts are atomic non-pointers. The execution time per word is approximately constant independent to the length of lists. The time in LINK is 1.1 times as long as that of CONV, while 1.5 times in RSVS. The reason why the performance of RSVS is far worse than LINK and CONV is strongly related to tail recursion. That is, LINK processes car first so that tail recursion greatly reduce the number of accesses, while RSVS cdr first. In fact, accesses per node in LINK is only one more than that in CONV, while that in RSVS is four more. This observation is supported by the result in which RSVS processes elements in ascending order. This result shows that accesses per node is three less than that in CONV, and the execution time is 0.87 as long as CONV.

4.3 Number of Page Faults

Bad locality, which is the major drawback of the conventional method, will cause frequent page faults as active data size becomes larger. Since data structures that is consecutively accessed by breadth-first traverse are located distantly from each other, it is unlikely that they are in a memory page.

In order to evaluate whether this problem is serious and how well our methods solve it, we measured the number of page faults that occur when a binary tree of depth 20 is copied by each method. This evaluation, however, cannot be actually performed using the program discussed in the previous section because of too many page faults in CONV. Therefore, we gathered address trace in garbage collection

Table 1: Required physical pages to avoid additional page faults

CONV	LINK	RSVS
3585	1026	2050

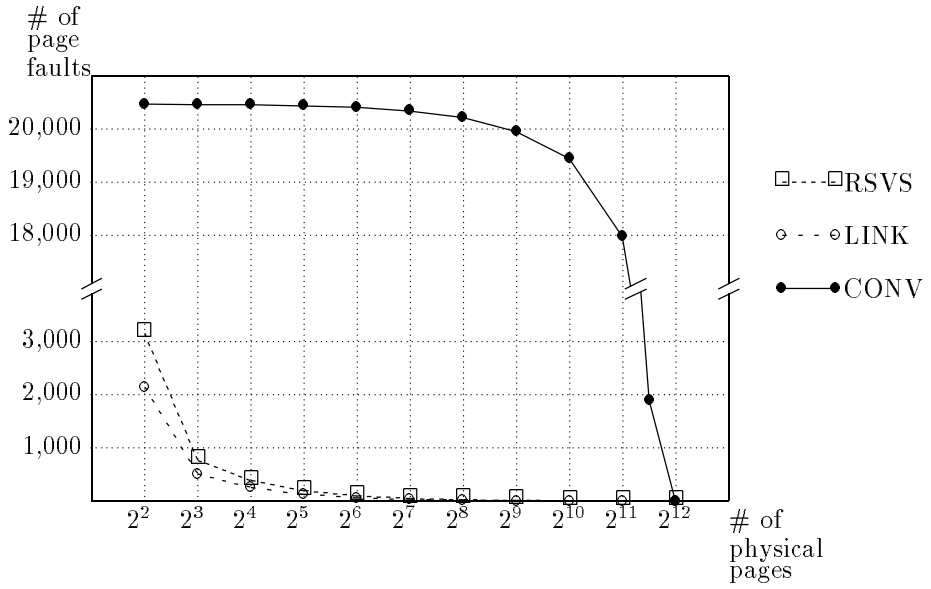

Figure 7: Number of page faults with physical pages less than required

and simulated page fault occurrence with it. On the simulation, it is assumed that the size of a memory page is 4 KB and all the pages are swapped out at the beginning of the collection. Since the binary tree occupies 8 MB or 2048 page memory space, 4096 page faults is unavoidable. Therefore, we measured the number of faults additional to 4096 varying number of physical pages.

Table 1 shows the minimum number of physical pages so that no additional faults occur. The result is that LINK requires only about a quarter of accessed pages and RSVS does about a half. On the other hand, about 7/8 of accessed pages are necessary in order that CONV completes the collection without page faults. This result is an evidence for bad locality of CONV. Another evidence is obtained when we measured the minimum number on depth-first left-to-right traverse of the collection result. In this case, only one page is enough for LINK and RSVS, while CONV needs 2047 pages that is almost the whole of accessed.

We also measured the number of page faults decreasing physical pages from that shown in Table 1. The result in Figure 7 shows that faults in LINK and RSVS gently increase as physical pages decrease. In case of CONV, contrarily, the slope is far steeper and about 18,000 times faults occur even with 2048 physical pages, which is a half of accessed. The reason of this terribly bad result is that nodes deeper than certain level are located in completely different pages so that visiting one to another always cause a page fault.

These evaluation results reveal that CONV cannot work practically when active data structures spread in large memory space, because of too many page faults. This problem will not be solved even by limiting the size of old- and new-area to that of physical memory space or less, because bad locality should cause frequent TLB miss.

5 Related Works

Depth-first copying will be implemented by using a stack naively as described in Section 3, or using reversed pointers proposed by Deutsch, Schorr and Waite[5, 6]. Using such a method that is less complicated than ours, various evaluation results, for example that in [7], shows good locality. However, those methods are seriously inferior to ours because the former requires extra stack space and the latter needs a marking bit for each word.

Another method to improve access locality, proposed by Wilson et al., is two level breadth-first

copying[8]. In this method, when a data structure is found during the scan of a page p in new-area and the structure is copied to another page p', scanning is switched to p' suspending the scan of p. Since scanning pages are switched recursively, a stack to keep the context of a page is necessary. Another problem is that this method only improves access locality in collection, but not in mutation in which data structures are traversed depth-first order.

Copying garbage collection has another problem that copied data will not line up as they are generated. Although our methods do not contribute to solving this problem, they can be combined with other methods, such as that shown in [9] which keeps the order of segments of Prolog heap, in order to remove stack area and/or marking bits. A method that keep the order completely is proposed by Koide[10]. This method, however, uses a marking stack, and assumes that a data structure has two or more words in order to remove marking bits.

6 Conclusion

In this paper, we proposed two depth-first copying garbage collection methods. Both methods inherit merits from the conventional copying method that access active data only and gather and pack them into a consecutive memory area. Features to distinguish our methods from other related ones are that neither of ours require extra memory spaces, data tags nor marking bits and their computation complexity is the same as that of the conventional.

The most remarkable advantage of our methods is good memory access locality due to depth-first copying. This advantage is proved by the evaluation result that shows drastically reduction of page faults. Another merit is to improve the flexibility of data representation design. That is, our methods allow pointer tags because they copy data structures without blind scan.

References

[1] Fenichel, R. and Yochelson, Y.: A Lisp Garbage Collector for Virtual Memory Computer Systems, *Comm. ACM*, Vol. 12, No. 11, pp. 419–429 (1969).

[2] Baker, Jr., H. G.: List Processing in Real Time on a Serial Computer, *Comm. ACM*, Vol. 21, No. 4, pp. 280–294 (1978).

[3] Unger, D.: *The Design and Evaluation of a High Performance Smalltalk System*, ACM Distinguished Dissertations, The MIT Press (1987).

[4] Nakashima, H. and Chikayama, T.: Copying Garbage Collection Methods Allowing Substructure Pointers, Technical Report TR-870, ICOT (1994). (in Japanese).

[5] Cohen, J.: Garbage Collection of Linked Data Structure, *Computing Surveys*, Vol. 13, No. 3, pp. 341–367 (1981).

[6] Schorr, H. and Waite, W. M.: An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures, *Comm. ACM*, Vol. 10, No. 8, pp. 501–506 (1967).

[7] Stamos, J. W.: Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory, *ACM Trans. on Programming Languages and Systems*, Vol. 2, No. 2, pp. 155–180 (1984).

[8] Wilson, P. R., Lam, M. S. and Moher, T. G.: Effective "Static-graph" Reorganization to Improve Locality in Garbage-Collected Systems, *Proc. ACM SIGPLAN'91*, pp. 177–191 (1991).

[9] Bekkers, Y., Ridoux, O. and Ungaro, L.: Dynamic Memory Management for Sequential Logic Programming Languages, *Proc. Intl. Workshop on Memory Management 92*, LNCS 637, Springer-Verlag, pp. 82–102 (1992).

[10] Koide, H. and Noshita, K.: On the Copying Garbage Collector which Preserves the Generated Order, *Trans. Information Processing Society of Japan*, Vol. 34, No. 11, pp. 2395–2400 (1993). (in Japanese).

Appendix

A.1 Definition of Data Type, Variables, Procedures and Functions

```
define mem_word = ⟨data type for memory words⟩ ;
define mem_addr = ⟨data type for memory word addresses⟩ ;
define mem_tag = ⟨data type for tags of memory words⟩ ;

old_base :      mem_addr ;          {base of old-area}
new_base :      mem_addr ;          {base of new-area}
new_size :      integer ;           {number of words in new-area}
set_of_roots :  set of mem_addr ;   {set of addresses of roots}
ptag :          mem_tag ;           {some pointer type tag}

function load(a : mem_addr) : mem_word ; ⟨return the value of the word at a⟩
procedure store(a : mem_addr, w : mem_word) ; ⟨store w into a⟩
function tag(w : mem_word) : mem_tag ; ⟨return the tag of w⟩
function address(w : mem_word) : mem_addr ; ⟨return the address of the word pointed by w⟩
function meke_word(t : mem_tag, a : mem_addr) : mem_word ;
        ⟨return a pointer to address a with tag t⟩
function is_pointer(w : mem_word) : boolean ; ⟨true if w is a pointer⟩
function copied(w : mem_word) : boolean ; ⟨true if the data structure pointed by w has been copied⟩
function object_size(w : mem_word) : integer ; ⟨return the size of data structure pointed by w⟩
procedure copy_words(ao : mem_addr, n : integer, an : mem_addr) ; ⟨copy n words from ao to an⟩
```

A.2 Conventional Method

```
procedure breadth_gc ;
   new_alloc :  mem_addr ;
   new :        mem_addr ;
   word :       mem_word ;
   old :        mem_addr ;
   size :       integer ;
   procedure copy_data ;
   begin
      word ← load(new) ;
      if is_pointer(word) then begin
         old ← address(word) ;
         if copied(word) then
            store(new, make_word(tag(word), address(load(old)))) ;
         else begin
            store(new, make_word(tag(word), new_alloc)) ;
            size ← object_size(word) ;
            copy_words(old, size, new_alloc) ;
            store(old, make_word(ptag, new_alloc)) ;
            new_alloc ← new_alloc + size ;
         end ;
      end ;
   end ;
begin
   new_alloc ← new_base ;
   for ∀new ∈ set_of_roots do
      copy_data ;
   new ← new_base ;
   while new < new_alloc do begin
```

```
        copy_data ;
        new ← new + 1 ;
    end ;
end ;
```

A.3 Naive-Stack Method

```
procedure depth_gc ;
    ⟨variable declarations same as breadth_gc⟩
    :
    pword  :        mem_word ;
    offset  :       integer ;
    old_head  :     mem_addr ;
    new_head  :     mem_addr ;
    n  :            integer ;
    hsize  :        integer ;
    stack_empty  :  boolean ;
    function copy_header(w : mem_word, a : mem_addr) ;
            ⟨copy the header of the data structure pointed by w and return its size⟩
    procedure initialize_stack ;
    begin
        stack ← stack_base ;
        pword ← NULL ;
        offset ← 0 ;
    end ;
    procedure push_stack ;
    begin
        store(stack, pword) ;
        store(stack + 1, offset + 1) ;
        stack ← stack + 2 ;
        pword ← word ;
        offset ← hsize ;
        old_head ← old ;
        new_head ← new − offset ;
        n ← size − offset ;
    end ;
    procedure next_element ;
    begin
        if stack = stack_base then
            stack_empty ← true ;
        else begin
            offset ← offset + 1 ;
            word ← load(old_head + offset) ;
            new ← new_head + offset ;
            n ← n − 1 ;
            if n = 1 then begin
                stack ← stack − 2 ;
                pword ← load(stack) ;
                offset ← load(stack + 1) ;
                if pword ≠ NULL then begin
                    old_head ← address(pword) ;
                    new_head ← address(load(old_head)) ;
                    n ← object_size(pword) − offset ;
                end ;
            end ;
        end ;
```

```
    end ;
begin
    new_alloc ← new_base ;
    initialize_stack ;
    for ∀new ∈ set_of_roots do begin
        word ← load(new) ;
        repeat begin
            stack_empty ← false ;
            if is_pointer(word) then begin
                old ← address(word) ;
                if copied(word) then begin
                    store(new, make_word(tag(word), address(load(old)))) ;
                    next_element ;
                end ;
                else begin
                    size ← object_size(word) ;
                    store(new, make_word(tag(word), new_alloc)) ;
                    hsize ← copy_header(word, new_alloc) ;
                    new ← new_alloc + hsize ;
                    word ← load(old + hsize) ;
                    store(old, make_word(ptag, new_alloc)) ;
                    new_alloc ← new_alloc + size ;
                    if size = hsize then
                        next_element ;
                    else if size − hsize > 1 then
                        push_stack ;
                end ;
            end ;
            else begin
                store(new, word) ;
                next_element ;
            end ;
        end ;
        until stack_empty ;
    end ;
end ;
```

A.4 Link Method

```
procedure link_gc ;
    ⟨variable declarations same as breadth_gc⟩
    :
    old_base  :    mem_addr ;
    old_link  :    mem_addr ;
    new_link  :    mem_addr ;
    hsize  :       integer ;
    stack_empty  :  boolean ;
    function copy_header(w : mem_word, a : mem_addr) ; ⟨same as that in depth_gc⟩
    function old_to_new(a : mem_addr) ; old_to_new ← a − old_base + new_base ;
    function last_word(w : mem_word) ; ⟨true if w is pointer to new-area⟩
    procedure initialize_stack ;
    begin
        old_link ← NULL ;
    end ;
    procedure push_stack ;
    begin
```

```
        store(new_alloc − 1, load(old + size − 1)) ;
        store(old_link, new_link) ;
        store(old + size − 1, make_word(ptag, old_to_new(old_link))) ;
        old_link ← old + hsize ;
        new_link ← new ;
    end ;
    procedure next_element ;
    begin
        if old_link = NULL then
            stack_empty ← true ;
        else begin
            old_link ← old_link + 1 ;
            new_link ← new_link + 1 ;
            new ← new_link ;
            word ← load(old_link) ;
            if last_word(word) then begin
                old_link ← new_to_old(address(word)) ;
                word ← load(new_link) ;
                new_link ← load(old_link) ;
            end ;
        end ;
    end ;
begin
    ⟨same procedure as depth_gc⟩
end ;
```

A.5 Reservation-Stack Method

```
procedure stack_gc ;
    ⟨variable declarations same as breadth_gc⟩
    :
    old_e :          mem_addr ;
    new_e :          mem_addr ;
    hsize :          integer ;
    n :              integer ;
    stack_empty :    boolean ;
    function copy_header(w : mem_word, a : mem_addr) ; ⟨same as that in depth_gc⟩
    function old_area(a : mem_addr) ; ⟨true if a is new-area address⟩
    function reserved(w : mem_word) ;
⟨true if the data structure pointed by w is reserved⟩
    procedure pop_stack ;
    begin
        new ← address(load(stack)) ;
        repeat begin
            word ← load(new) ;
            store(new, make_word(tag(word), new_alloc)) ;
            new ← address(word) ;
        end ;
        until old_area(new) ;
        old ← new ;
        store(old, load(stack + 1)) ;
        size ← object_size(word) ;
        hsize ← copy_header(word, new_alloc) ;
        new ← new_alloc + hsize ; new_e ← new ;
        old_e ← old + hsize ;
        word ← load(old_e) ;
```

```
        store(old, make_word(ptag, new_alloc)) ;
        new_alloc ← new_alloc + size ;
        n ← size − hsize ;
        stack ← stack + 2 ;
    end ;
    procedure next_element ;
    begin
        n ← n − 1 ;
        if n > 0 then begin
            old_e ← old_e + 1 ;
            new_e ← new_e + 1 ; new ← new_e ;
            word ← load(old_e) ;
        end ;
        else if stack = new_base + new_size
            stack_empty ← true ;
        else
            pop_stack ;
     end ;
begin
    new_alloc ← new_base ;
    stack ← new_base + new_size ;
    for ∀new ∈ set_of_roots do begin
        word ← load(new) ;
        n ← 1 ;
        repeat begin
            stack_empty ← false ;
            if is_pointer(word) then begin
                old ← address(word) ;
                if copied(word) then begin
                    store(new, make_word(tag(word), address(load(old)))) ;
                    next_element ;
                end ;
                else if reserved(word) then begin
                    old ← address(load(old)) ;
                    store(new, make_word(tag(word), address(load(old)))) ;
                    store(old, make_word(ptag, new)) ;
                    next_element ;
                end ;
                else if object_size(word) = 1 then begin
                    store(new, make_word(tag(word), new_alloc)) ;
                    word ← load(old) ;
                    store(old, make_word(ptag, new_alloc)) ;
                    new ← new_alloc ;
                    new_alloc ← new_alloc + 1 ;
                end ;
                else begin
                    store(new, word) ;
                    stack ← stack − 2 ;
                    store(stack, make_word(ptag, new)) ;
                    store(stack + 1, load(old)) ;
                    store(old, make_word(ptag, stack)) ;
                    next_element ;
                end ;
            end ;
            else begin
                store(new, word) ;
                next_element ;
            end ;
```

```
        end ;
        until stack_empty ;
    end ;
end ;
```