Volume 7, number 1 INFORMATION PROCESSING LETTERS January 1978

GENETIC ORDER AND COMPACTIFYING GARBAGE COLLECTORS

Motoaki TERASHIMA and Eiichi GOTO
Department of Information Science, Faculty of Science, University of Tokyo, Tokyo, 113, Japan

Received 31 August 1977

Linear order among lists, garbage collection, compactifying garbage collection, hardware garbage collector, binary tree search

1. Introduction

A linear order to be called the *genetic order*, or G-order for short, can be established among list structures by using their storage addresses, endowed upon creation (the genesis of the term "genetic"), as ordering indices, provided that the list processing system employed consistently preserves the G-order. Especially, the system garbage collector (GC) has to be of the type of "*genetic order preserving* (GOP)". It should be noted that the classic GC in LISP 1.5 [10] is GOP. Wegbreit's compactifying GC [13] is also GOP, but other compacting and compactifying GC's by Bobrow [3], Cheney [4] and Hansen [7] are non-GOP. (While "compacting" means the relocation of active cells into a consecutive area with the elimination of cdr cells of lists [4,7], "compactifying" means the same without cdr cell elimination [3,13].)

A large number of searching, sorting and merging techniques which all utilize some kind of linear order among the data, has been developed and applied to the speeding up of operations on various data structures [1,9]. An objective of this note is to point out the fact that the techniques just mentioned are all readily applicable to list structures by utilizing the G-order in GOP-systems. Although the utilization of the G-order is not practiced in most existing list processing schemes, it would be worth while to fully exploit the effectiveness of such utilization of the G-order, and as the first step, we classify list processing schemes especially garbage collectors into GOP and non-GOP.

Another objective of this note is to propose two compactifying GOP-GC's. Compactifying GC's generally consist of three phases: marking of active cells, pointer adjustments and relocation of cells. Marking and relocation algorithms are relatively simple, and fast algorithms (with the time complexity being proportional to the amount of storage, e.g., see [8]) can be designed easily. On the other hand, the pointer adjustment phase of most known schemes is rather costly in time or space or in both [3,4,7,13].

Two pointer adjustment schemes are proposed in this note. One is slow, but requires no extra storage. It uses a G-ordered binary tree, i.e., a binary tree with the address of each tree node itself being used as a nodal datum. This scheme also gives an example of an algorithm which fully utilizes the G-order (G-order self applied to an implementation of a GOP system).

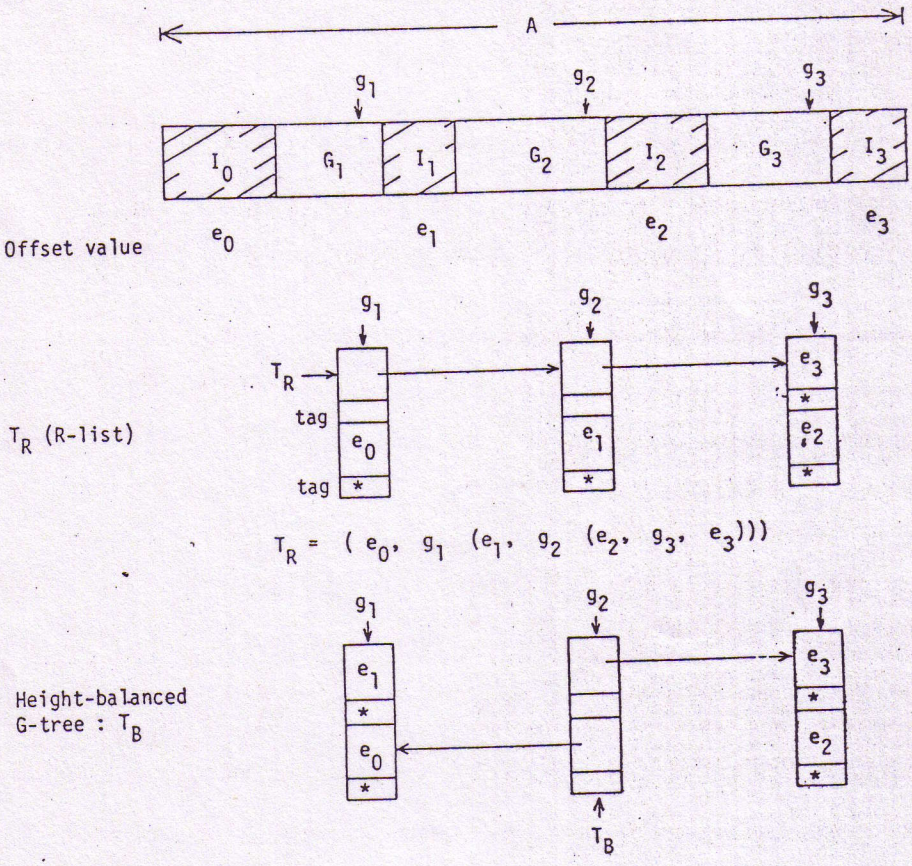
The other is fast, O(1) per pointer adjustment and is believed to be suited for hardware implementation, but requires some extra storage.

2. Pointer adjustment scheme using a G-ordered tree

Let A be the address space. At the beginning of the pointer adjustment phase, let A be partitioned into *islands* of successive active cells and *gaps* between islands as

$$A = I_0 \cup G_1 \cup I_1 \ldots G_n \cup I_n$$

(cf. Fig. 1) where I_i is the i-th island and G_i is the gap between islands I_{i-1} and I_i. The trivial case consisting



A : the address space
I_i: i-th island
G_i: gap between I_{i-1} and I_i
e_i: the offset value of I_i
g_i: an address chosen from G_i

Fig. 1. Islands, gaps and the G-tree representation.

of a single island is omitted. Without loss in generality we assume that all islands and gaps are non-void. When the active pointer belongs to I_i, the pointer has to be adjusted by an *offset value* e_i which is the amount that I_i has to be moved upon relocation. Let g_i be an address chosen from each gap G_i as a representative. The offset value for adjusting a pointer p is defined by the following conditional formula:

$$\text{offset value} = \begin{cases} e_0 & \text{if } p < g_1, \\ e_i & \text{if } g_i < p < g_{i+1}, \quad (1 \leqslant i < n) \\ e_n & \text{if } g_n < p. \end{cases} \tag{2.1}$$

Obviously this conditional formula can be fully

specified by the $2n + 1$-tuple $D = (e_0, g_1, e_1, \ldots, g_n, e_n)$ of offset and gap data e_i and g_i.

Definition 1. The *G-tree* representation of D is defined as nested ordered triples made by inserting pairs of parentheses into D so that each second element of triples is g_i ($1 \leqslant i \leqslant n$, cf. Fig. 1). Each triple $(T' g_i T'')$ is said to be a node of the G-tree, where each of T' and T'' is either one of the offset values $e_0, e_1, \ldots, e_n$ (leaf of the tree) or a left- or right-subtree node.

Proposition 1. *For a given data D with n gaps, there are* ${}_{2n}C_n/(n+1)$ *different* G-*trees.*

Proof. This follows directly from the well known result in combinatorics (e.g., see [8, p.389]) on the number of binary trees with n nodes.

Proposition 2. *Any of the* G-*trees can fully specify the conditional formula* (2.1).

Proof. D can be reconstructed from the G-tree by eliminating all pairs of parentheses except for the outmost one.

Definition 2. A G-tree is called an R-*list* if all closing parentheses of the nested triples are placed at the right end (cf. Fig. 1). A G-tree is called an L-*list* if all opening parentheses are placed at the left end.

Proposition 3. *Data* $2n + 1$-*tuple D has a unique* R-*list and a unique* L-*list, and for* $n \geqslant 1$ *the* R-*list and the* L-*list are different.*

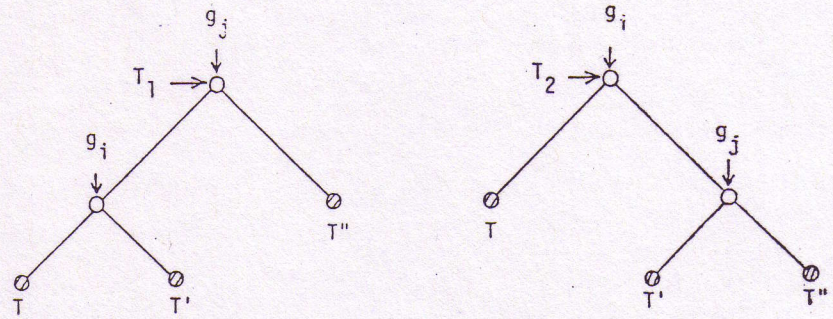
Proof. By induction on the number of G-tree nodes n.

Definition 3. Let $T_1 = ((Tg_iT')\, g_jT'')$ and $T_2 = (Tg_i(T'g_jT''))$ be G-trees or sub G-trees. A G-tree transformation $T_1 = \mathrm{L}(T_2)$ is called an L *transformation* (cf. Fig. 2). The inverse transformation $T_2 = \mathrm{R}(T_1)$ is called an R *transformation*.

Proposition 4. *For a given data* $2n + 1$-*tuple D, all* ${}_{2n}C_n/(n+1)$ *different* G-*trees can be obtained by applying* R *and* L *transformations successively to a* G-*tree.*

Proof. By induction on the number of the tree nodes n.

Since each gap G_i is assumed to be non-void containing at least one cell at address g_i, each node of the G-tree can be represented by using one cell at each gap, provided that each list cell can hold two tag bits and two data. A datum with tag bit on represents an offset value (leaf of tree), and a datum with tag bit off represents a pointer to another subtree node. Given a pointer p to be adjusted, and a root r of a G-tree, the offset value in accordance with (2.1) can be found by the recursive procedure (2.2) written in Pidgin ALGOL [1] in which "call-by-value" param-



R-transformation

$$T_1 = ((T\ g_i\ T')\ g_j\ T'') \quad \rightleftarrows \quad T_2 = (T\ g_i\ (T'\ g_j\ T''))$$

L-transformation

Fig. 2. Transformation of G-trees.

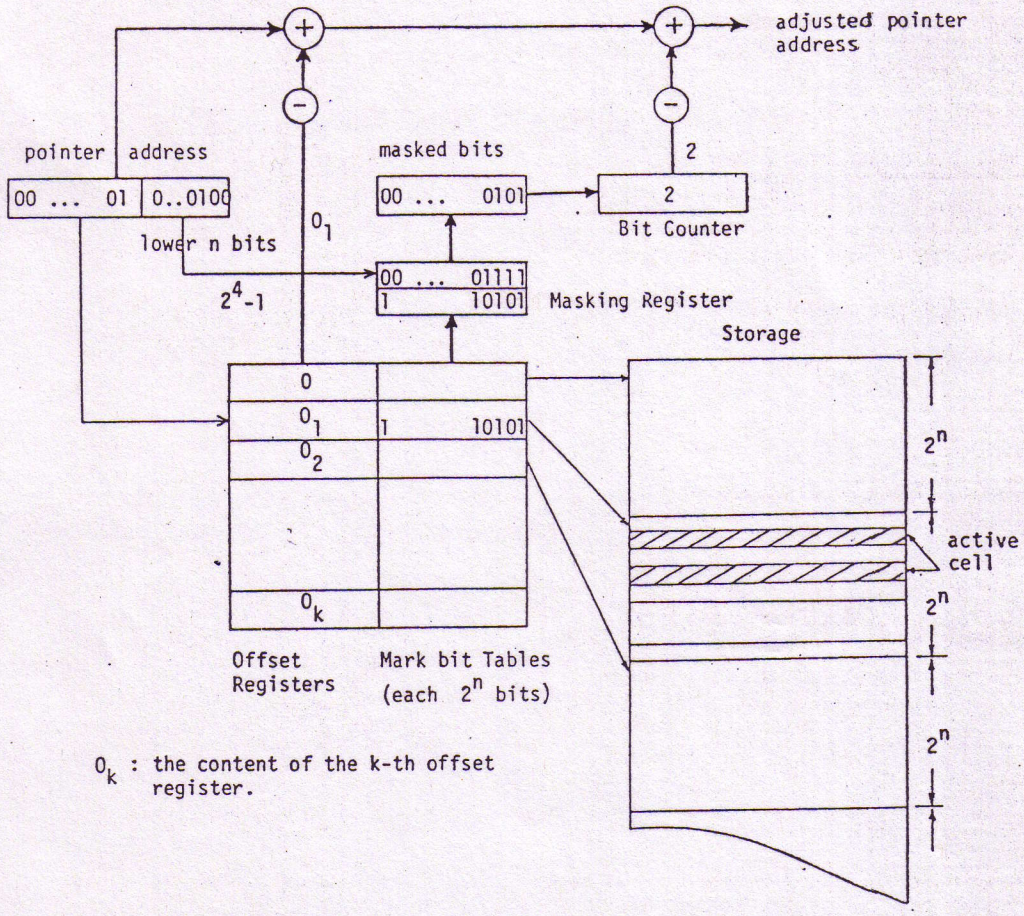


Fig. 3. The organization of pointer adjustment hard ware.

eter binding and mono data type representation of pointers and leaf (offset) values are assumed.

```
procedure GETOFFSET (p, r):
if r is a leaf then return r
else
    if r >_g p  then return GETOFFSET(p, leftsubtree(r))
                else return GETOFFSET(p, rightsubtree(r))
                                                    (2.2)
```

Here $>_g$ is the predicate for checking the G-order.

This procedure can be readily transformed into the following iterative procedure.

```
procedure GETOFFSET(p, r):
begin
L: if r is a leaf then return r;
   if r >_g p  then r ← leftsubtree(r)
               else r ← rightsubtree(r);
   goto L
end                                                 (2.3)
```

The correctness of this procedure applied to G-trees can be easily proved by induction on the number of nodes n of the G-tree.

Each node of the G-tree contains only two data. Procedure (2.2) ((2.3) as well) makes a binary tree search on the G-tree with making use of the address of each tree node itself as the third (implicit) datum. Note that three (explicit) data are needed per tree node in conventional binary tree search methods [1,9].

Wegbreit [13] proposed to make a binary search on a tree constructed in the inactive (gap) space to find the offset value, but the tree requires the storage for a nodal datum ('*covering address*' in his terminology) in addition to the two pointers. The gap consisting of one cell is not enough for storing these three data in his case, so that the binary search tree

cannot be constructed without using extra storage in case each gap consists of only one cell.

The speed of binary tree search can be maximized by balancing the height of the tree like in the case of AVL trees [2]. An R-list with height n can be easily constructed by a linear sweep through the storage. By using a well established method of balancing the height of trees (e.g., see [1,9]), we readily obtain the following procedure (2.4) which transforms the R-list into a height-balanced tree with height $\lceil \log_2 n \rceil$.

```
procedure BALANCETREE(t):
begin
1.    p ← e_n;
2. A: s ← t; r ← rightsubtree(t);
3. B: u ← rightsubtree(t);
4.    if u = p then if t = s then return t
          else
5.            begin p ← t; t ← r; goto A
              end;
6.    v ← rightsubtree(u);
7.    if v = p then if u = r then return t
          else
8.            begin p ← u; t ← r; goto A
              end;
9.    replace rightsubtree of s by u;
10.   replace rightsubtree of t by leftsubtree(u);
11.   replace leftsubtree of u by t;
12.   t ← v; s ← u; goto B
end                                          (2.4)
```

Here t is initially the root of the R-list, and the procedure returns the root of the resultant balanced tree.

Proposition 5. *Given a root r of an* R*-list, the result of procedure* (2.4) *is a (height-balanced)* G*-tree.*

Proof. In procedure (2.4) tree transformation is performed at lines 9–11, which turns out to be an L transformation. Hence, the resultant tree is a G-tree by Proposition 4.

By using the height-balanced G-tree with n nodes in procedure (2.2), time for each pointer adjustment is $O(\log_2 n)$.

3. A fast pointer adjustment scheme

Each pointer P can be adjusted in O(1) time by using extra storage for *offset registers* and marking bit tables. This scheme is believed to be suited for hardware implementation (see Fig. 3) to calculate the offset value efficiently. The calculation of the offset value for adjusting pointer P is to be made as follows:

Mask the 2^n bit string from the $\lfloor P/2^n \rfloor$-th marking bit table with the value $2^{\mathrm{mod}(P,2)}-1$ (bit string of '1's, $\mathrm{mod}(P, 2^n)$ in length, where $\mathrm{mod}(P, 2^n) = P - \lfloor P/2^n \rfloor * 2^n$). Count inactive state bits (marked '1') of the masked bit string. (The count is the number of inactive cells from $\lfloor P/2^n \rfloor * 2^n$ to P.) Add it to the content of $\lfloor P/2^n \rfloor$-th offset register which holds the total number of inactive cells counted from 0 to $\lfloor P/2^n \rfloor * 2^n - 1$. The contents of offset registers are set by counting inactive state bits in the bit table at the beginning of the pointer adjustment phase.

If buffer (cache) memory is used for bit table and offset registers and if the i-th offset register and the i-th marking bit table can be read simultaneously, this calculation can be performed within a single buffer cycle time. Moreover, if a machine with a pipe line control for read and write operations on memory be built, the pointer adjustment time will be absorbed into the data relocation time.

In case $n = 4$ and the storage capacity is 65 K list cells each 64 bit long, 16 bits are enough for each offset register, so that the extra storage needed is 3.125% of the total storage.

4. Concluding remarks

The concepts of G- ("genetic") order and GOP ("genetic order preserving") system were introduced, and the effectiveness of the utilization of the G-order was exemplified by a binary search on G-ordered binary tress.

The concept of the G-order is believed to have many applications in searching, sorting and merging techniques which all utilize some kind of linear order among the data. Concurrent garbage collector [11,12] of the GOP type would be an interesting theme for further research.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms (Addison-Wesley, Reading, MA, 1974) 113–169.

[2] G.M. Adel'son-Vel'skii and E.M. Landis, An Algorithm for the Organization of Information, Doklady Akademiia Nauk, SSSR 146 (1962) 263–266; English translation in Sov. Math. 3, 1259–1263.

[3] D.J. Bobrow, ed., Symbol Manipulation Languages and Techniques (North-Holland, Amsterdam, 1971) 296.

[4] C.J. Cheney, A Nonrecursive List Compacting Algorithm. Comm. ACM 13 (11) (Nov. 1970) 677–678.

[5] J.P. Fitch and A.C. Norman, A Note on Compacting Garbage Collection, University of Cambridge, Computer Laboratory (1976).

[6] E. Goto, Monocopy and Associative Algorithms in an Extended LISP, Information Science Lab. Technical Report 74-03, University of Tokyo (Apr. 1974).

[7] W.J. Hansen, Compact List Representation: Definition Garbage Collection and System Implementation. Comm. ACM 12 (9) (Sep. 1969) 499–506.

[8] D.E. Knuth, The Art of Computer Programming, Vol. 1, Fundamental Algorithms, 2-nd ed. (Addison-Wesley, Reading, MA, 1973) 413–420.

[9] D.E. Knuth, The Art of Computer programming, Vol. 3, Sorting and Searching (Addison-Wesley, Reading, MA, 1973) 422–471.

[10] J. McCarthy et al., LISP 1.5 Programmer's Manual (MIT Press, Cambridge, MA, 1965).

[11] K.G. Müller, On the Feasibility of Concurrent Garbage Collection, Doctor thesis, Technical University, Delft, The Netherlands (1975).

[12] G.L. Steel, Jr., Multiprocessing Compactifying Garbage Collection, Comm. ACM, 18 (9) (Sept. 1975) 495–508.

[13] B. Wegbreit, A Generalized Compactifying Garbage Collector, The Computer J. 15 (3) (Aug. 1972) 204–208.