数理解析研究所講究録
第 363 巻 1979 年 218-246

Fast List-Copying Algorithms Using Constant Workspace

Hiroshi Hasegawa

Computer Science Division
Electrotechnical Laboratory
2-6-1 Nagata-cho, Chiyoda-ku
Tokyo 100 JAPAN

Abstract

Two linear-time constant-workspace algorithms are presented for copying an arbitrarily linked list structure. Apart from a fixed number of program variables, copying can be done without auxiliary storage, such as a stack. The first algorithm, assuming one mark bit in each cell, demonstrates its efficient list traversal technique which embeds the processing order into the structure being built. The second algorithm, delivered from the first, needs no mark bits and makes a copy into a block of contiguous areas of memory. It is shown to be faster than Clark's algorithm, the fastest previous linear-time algorithm for the same problem.

Key Words and Phrases : list processing, copying, Lisp, linear time, space complexity, constant workspace

CR Categories : 4.34, 4.49, 5.25

1. Introduction

The problem considered here is the creation of a copy of an arbitrary Lisp-style list structure under the constraint of constant workspace. This constraint requires that copying can be done in constant working storage without the use of a stack or any other working storage whose size depends on the size or complexity of the list to be copied. The only storage available is that occupied by both the original list structure and the copy being created, separate from a fixed number of program variables. Copying requires that the original structure may not be permanently destroyed, unlike the task of moving a list[1, 2].

Algorithms for copying list structure using constant workspace have been given by Lindstrom[8], Fisher[6], Robson[10], and Clark[3, 5]. Lindstrom showed that arbitrary n-cell structure can be copied in time $O(n^2)$ if there are no mark bits and in time O(nlogn) if a mark bit is available in each cell. Robson's algorithm uses only linear time and no mark bits. Both Lindstrom's and Robson's algorithms can copy into an arbitrary free-list. Although Fisher's algorithm runs faster than Robson's, it depends on the restriction that copy cells must be allocated into a block of contiguous storage of locations as his free-list to gain speed instead of generality. Clark's algorithm is faster than Fisher's and has the same free-list restriction.

This paper reports two new algorithms, called ListCopy1 and ListCopy2, for copying an arbitrary Lisp-style list structure in linear time using constant workspace. Both algorithms run under the same strategy, but their assumptions are different. The first algorithm, ListCopy1, assumes a mark bit in both original and copy cells and is much faster than Robson's algorithm. The second algorithm, ListCopy2, is faster than Clark's and has the same free-list restriction. While Clark's algorithm requires slightly more than two full passes, depending on the degree of sharing in the structure to be copied, ListCopy2 requires just two passes for any list structures. In addition, the second pass here is more efficient than in Clark's algorithm, especially in case of a full binary tree where all leaf cells consist of cyclic pointers.

2. Copying List with a Mark Bit : ListCopy1

2.1 Outline of List Copying

Consider a list or list structure composed of list cells containing two pointers called car and cdr, which may point to any list cell or to nonlist objects called atom[9]. Atom themselves are not copied, so atom pointers do not change when copied. The list structure to be copied will be called the old or original list; its copy will be called the new list or the copy.

The process of copying a list structure typically entails 2 passes. Pass1 consists of the followings: the original list is traversed to find unprocessed cells; new cell for each unprocessed original cell is allocated; and that cell is linked to its copy. Figure 1(a) shows a typical list structure about to be copied, and Figure 1(b) shows the stage at the end of Pass1. In Pass2, the pair of the original cell and the corresponding copy cell is decomposed by traversing the list consisting of these pairs. The old car and old cdr are stored to the car and cdr part of the original cell respectively, so the original list will be restored. At the same time, the values of the copy car and copy cdr are the copy cells corresponding to the original cells pointed by

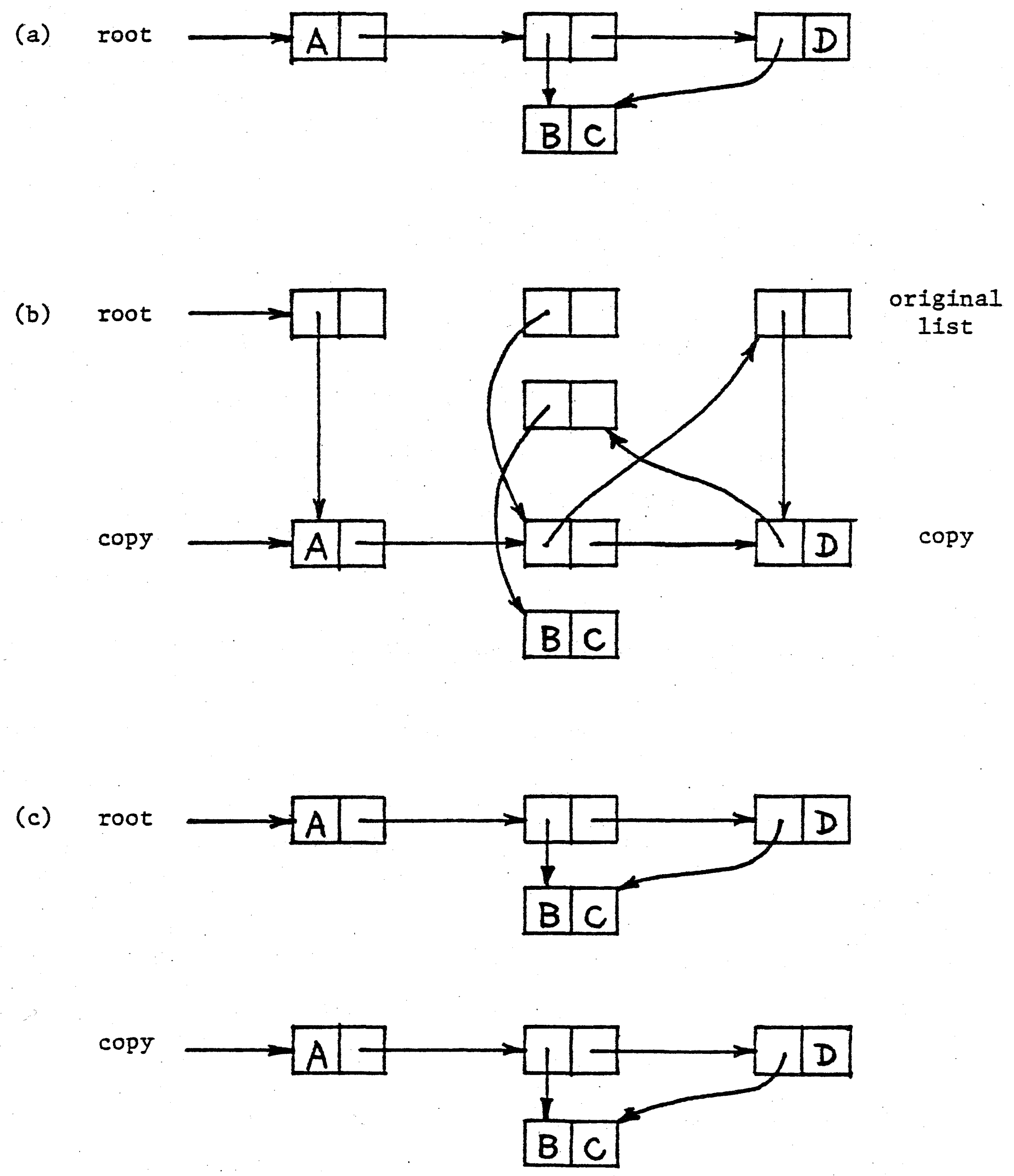


Fig. 1. Steps in copying a list : (a) initial state; (b) intermediate state; (c) final state.

the old car and old cdr, respectively, if these are the list pointers. Thus these values are stored to the car and cdr part of the copy cell as shown in Figure 1(c).

2.2 Pointer Types and List Cell Types

ListCopy1 will perform two traversals of the list structure. Pass1 does a preorder traversal which will visit list cells in the following order: root, then car (if an unvisited list), and then cdr (if an unvisited list). Traversing the original list, a mark bit denoted as mark which is initially zero will be changed to 1. This means that the cell has already been visited:

```
procedure Marked (listcell) : return mark(listcell) = 1 .
```

Because list cells are marked in preorder, a mark bit in the original cells would clearly permit the orderly traversal of possibly cyclic structures and prevent the creation of supurious copies of cells that are pointed to more than once, such as (B . C) in Figure 1.

After finding the unvisited original cell, the new copy cell will be allocated for it from the avail-list. A forwarding link will be installed in car of each original cell in order to link that cell to its copy. The simple function Flink, below, will be used to extract a forwarding link from an old cell.

```
procedure Flink(listcell) : return car(listcell)
```

While some of original cars displaced by forwarding links need be saved in the copy cell, some of them need not. This reason will be described in detail in Section 2.4.

The preorder traversal in Pass1 defines a spanning tree of the list structure to be copied. Pointers that are in the original list but are omitted from this spanning tree are those that point to a cell visited earlier in the traversal. Following Robson[10], we call list pointers included in the spanning tree forward pointers and all other ones omitted back pointers. Figure 2 illustrates the spanning tree: forward pointers are solid; back pointers dotted.

Each list cell belongs to one of nine types, depending on the three possible types of its car and cdr: atom, forward pointer, or back pointer. Cell types will be represented as AA, AF, AB, BA, and so on, following Clark[5]. The computation of these types during Pass1 will be accomplished by the procedure CellType1.

```
procedure CellType1(oldcar, oldcdr) :
  begin
    return
    if not(atom(oldcar)) or not(Marked(oldcar)) then
      if atom(oldcdr) then FA
      else if Marked(oldcdr) then FB else FF
    else
      if not(atom(oldcdr)) or not(Marked(oldcdr)) then
        if atom(oldcar) then AF else BF
      else AA&AB&BA&BB
  end
```

In traversing the original list, two visits are required to a cell of type FF which has two forward pointers and no other types require a second visit. But there appears one complication in classifying a cell as type FF. Consider the case where some chain of forward pointers starting with car eventually reaches the cdr cell, illustrated as the root cell in Figure 2. It is observed that the cdr cell has already been visited when the entire car structure of that cell has been traversed. A cell originally classified to be FF will then reveals to be FB. We will call such cells inscrutable.

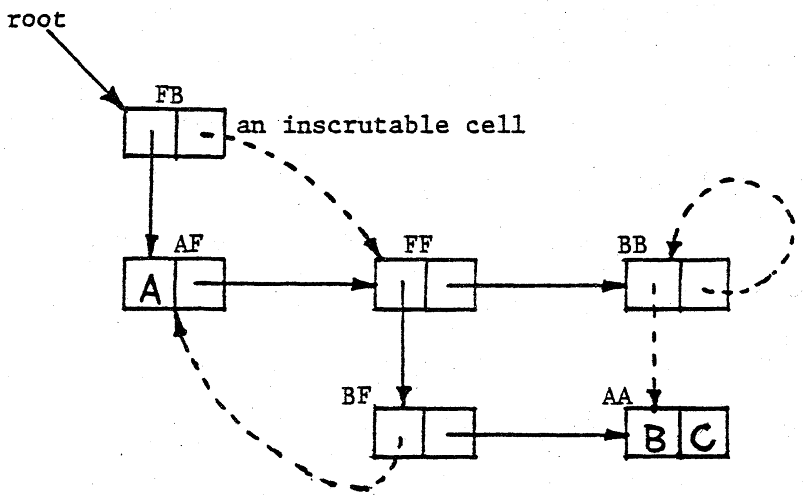


Fig. 2. Spanning tree defined by the preorder traversal. Solid pointers are included in the spanning tree; dotted pointers are omitted.

2.3 List Traversal in Pass1

There are several methods to traverse the list structure. Lindstrom's algorithm[8] uses his link permutation technique; Robson's algorithm[10] uses the Deutsch-Schorr-Waite link reversal technique[7, 11]; Fisher's algorithm[6] uses Cheney's method[1], sequentially scanning the copy for forward-pointing cdrs.

In ListCopy1, however, the traversal of the original list will be accomplished by the usual sort of a stack, just the same as Clark's algorithms[3, 5]. A stack can be implemented without violating the constant workspace constraint by linking stacked cells together through their own cdrs in the copy cells. The familiar procedure Push and Pop can be implemented thus:

```
procedure Push(cell, stack) :
  begin
    cdr(cell) <-- stack ;
    stack <-- cell
  end

procedure Pop(stack) :
  begin pointer t ;
    t <-- stack ;
    stack <-- cdr(stack) ;
    return t
  end
```

The two other stack-referencing procedures Initialize and Empty are obviously implemented as follows:

```
procedure Initialize(stack) : stack <-- NIL

procedure Empty(stack) : return stack = NIL
```

On the first visit to a cell of type FF or an inscrutable cell originally observed to be FF, its copy will be pushed onto a stack. When the traversal of the original list has reached at one of the cells of type AA, AB, BA, or BB (leaves of the spanning tree), the stack will be popped and the cdr of the popped cell, which is the original cdr of the most recently visited type FF cell, will be returned, provided that the cell is not inscrutable. But the possible presence of inscrutable cells on the stack needs the procedure Pop1 to determine if the popped cell is inscrutable or not. Popped cells found to be inscrutable will cause Pop1 to pop up the stack several times until the type FF cell will be found. Inscrutable cells must, of course, get the type FB treatment to be described in Section 2.5. Similiary, a pointer to the next copy cell will be saved to the cdr of the popped copy cell of type FF.

```
procedure Pop1(stack) :
 begin
     until Empty(stack) do
      begin pointer newcell, oldcdr ;
        newcell <-- Pop(stack) ;
        oldcdr <-- car(newcell) ;
        if Marked(oldcdr) then              - newcell is the copy of an
           cdr(newcell) <-- FAddr(oldcdr)   - inscrutable cell.
        else
          begin                             - newcell is the copy of cell
            cdr(newcell) <-- avail ;        - still type FF.
            return oldcdr                   - oldcdr is about to be
          end                               - copied into next free cell
      end ;
    return NIL                              - only if stack is empty.
  end
```

2.4 Reverse Preorder Chain

In Pass2, forwarding links are removed by the orderly traversal of the list structure. At the same time, the original contents of each old cells are restored and final values of new pointers are stored to the copy cells, if they have not been stored in Pass1.

The value of a pointer in the copy depends upon its type: atom, back pointer, or forward pointer. By definition, atom themselves are not copied, so the new value of an atom pointer is always the same as its old value. New values of list pointers are, of course, given by forwarding links. But now that the list is orderly traversed in Pass2, it sometimes happens that a forwarding link has already been removed before it is needed. Then whether or not the necessary forwarding link is available in Pass2 depends crucially upon its traversal order of the list. Thus the new value of a list pointer which points to a cell without a forwarding link in Pass2 must be saved during the traversal in Pass1.

Now, our approach is based on the observation that new values of back pointers need not be saved in Pass1, provided that forwarding links are removed in reverse preorder during the traversal in Pass2, like Robson's algorithm[10]. A chain of original cells called a reverse preorder chain is prepared in Pass1 in order to guide the reverse preorder traversal in Pass2. Then in Pass2, forwarding links will be removed along with the reverse preorder chain.

A reverse preorder chain will be constructed as follows: each time the preorder traversal in Pass1 reaches an unvisited original cell, a pointer to its cell will be added to the front of it. Namely, the history that forwarding links are installed in Pass1 is preserved in reverse preorder as a reverse preorder chain. Therefore, every pair of old and new cells are able to be visited and decomposed by traversing the list along with the reverse preorder chain in Pass2. A reverse preorder chain is clearly a stack. So it can be implemented by linking the visited cells together through their own cdrs in the original cells by the procedure Push defined in Section 2.3. Also, procedure Initialize and Pop will be used to initialize and pop-up a reverse preorder chain, respectively, in Pass1 and Pass2.

Thus so far we have discussed that forwarding links are installed in preorder during Pass1 and they are removed in reverse preorder during Pass2 by the guidance of the reverse preorder chain. But note that a forward pointer points to a cell without a forwarding link in Pass2, while it points to a cell with one in Pass1. A forwarding link is not available for a forward pointer in Pass2. Consider, for example, a type FF cell which needs to save both old and new forward pointers for each car and cdr at the end of Pass1. It is not difficult to see that ListCopy1's storage requirements are most stringent at the end of Pass1, because a forwarding link and a reverse preorder chain occupy a original cell, that is, two of a cell's four available locations (old and new car and cdr). Then we can use only two locations in a copy to save necessary items at the end of Pass1.

How do we solve this problem? There is fortunately a way to compute both old and new forward pointers, although a forwarding link is not available for a forward pointer in Pass2. Pass1 performs a preorder traversal, depending upon the spanning tree consisted of forward pointers. On the contrary, Pass2 does a reverse preorder traversal, depending upon the same spanning tree according to the reverse preorder chain. Thus, in Pass2, the old and new forward pointer point to the old and copy cell which have already been processed, respectively. In particular, forward pointers of type AF, BF, FA and FB cells and those of car of a type FF cell point to a cell processed just before during Pass2. ListCopy1, therefore, need not save during Pass1 both old and new values of forward pointers, except those of part of a type FF cell, because cdr is traversed before car in Pass2.

Now that atom themselves are not copied, new values of atoms need

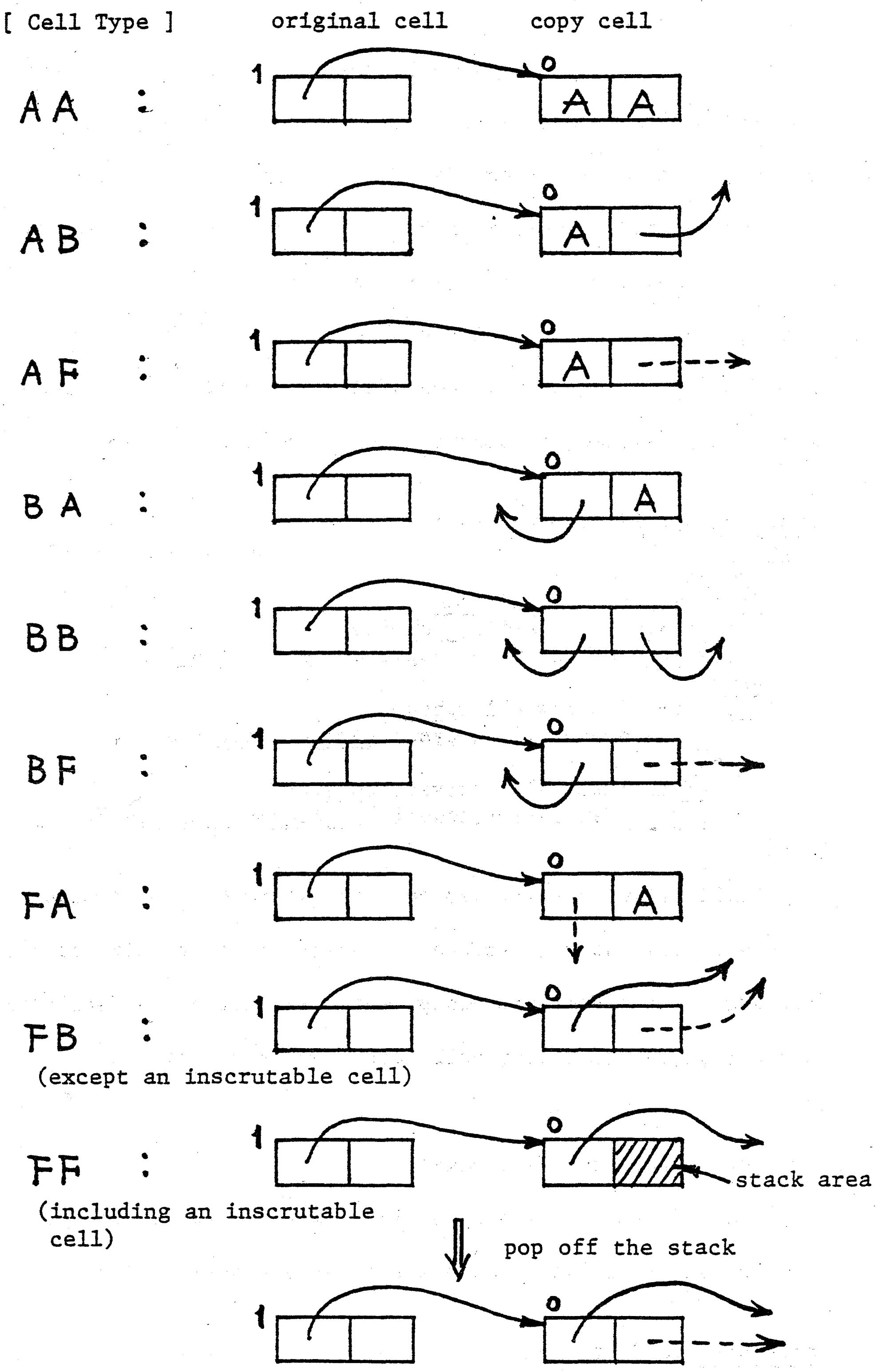


Fig. 3. State of each cell type at the end of Pass1 of ListCopy1. Atoms are denoted as A; original pointers are solid; copy pointers are dotted. A reverse preorder chain omitted is located in cdr of each original cell.

not be saved. And also new values of back pointers need not be saved at the end of Pass1, because new values of all back pointers can be given by forwarding links during Pass2.

Consequently, the number of items that must be stored at the end of Pass1 is managed to become less than two, except forwarding links and a reverse preorder chain. Pointers for each cell type at the end of Pass1 are shown in Figure 3. Depending upon it, the procedure CellType2, below, will be used to compute these types during Pass2.

```
procedure CellType2(newcell) :
  begin                                    - newcell is copy of oldcell.
    return
    if atom(car(newcell)) then
      if atom(cdr(newcell)) then AA
      else if cdr(newcell) = copyson then AF else AB
    else
      if atom(cdr(newcell)) then
        if car(newcell) = copyson then FA else BA
      else
        if cdr(newcell) = copyson then BF
        else if Marked(cdr(newcell)) then BB else FB&FF
  end
```

CellType2 closely relies on the process information during the traversal of Pass2, because it uses a pointer which points to a cell processed just before to recompute a cell's type. In CellType2, copyson means a pointer to a copy cell just processed before.

2.5 The Algorithm : ListCopy1

The procedure ListCopy1 is given in full detail below. Auxiliary procedures are defined in the previous sections so far.

```
procedure ListCopy1(root) :
begin pointer x, y, copy, copyson ;
  - Pass 1
  Initialize(Stack) ; Initialize(Chain) ;
  x <-- root ; copy <-- avail ;           - global variable avail points to
                                          - first free cell
```

```
until x = NIL do
  begin pointer oldcar, oldcdr ;
    mark(x) <-- 1 ; oldcar <-- car(x) ; oldcdr <-- cdr(x) ;
    y <-- avail ; car(x) <-- y ; Push(x, Chain) ;  - avail is the forwarding
    avail <-- cdr(avail) ;                         - address of cell x
    case CellType1(oldcar, oldcdr) of
      AA&AB&BA&BB : car(y) <-- oldcar ; cdr(y) <-- oldcdr ;
                    x <-- Pop1(Stack) ;
      AF : car(y) <-- oldcar ; cdr(y) <-- avail ; x <-- oldcdr ;
      BF : car(y) <--oldcar ; cdr(y) <-- avail ; x <-- oldcdr ;
      FA : car(y) <-- avail ; cdr(y) <--oldcdr ; x <-- oldcar ;
      FB : car(y) <-- oldcdr ; cdr(y) <-- FAddr(oldcdr) ; x <-- oldcar ;
      FF : car(y) <-- oldcdr ; Push(y, Stack) ; x <-- oldcar ;
    endcase
  end ;
- Pass 2
copyson <-- NIL ;
until Empty(Chain) do
  begin pointer oldcar, oldcdr, son ;
    x <-- Pop(Chain) ; y <-- FAddr(x) ;
    case CellType2(y) of
      AA : car(x) <-- car(y) ; cdr(x) <-- cdr(y) ;
      AB : oldcdr <-- cdr(y) ; cdr(y) <-- FAddr(oldcdr) ;
           car(x) <-- car(y) ; cdr(x) <-- oldcdr ;
      AF : car(x) <-- car(y) ; cdr(x) <-- son ;
      BA : oldcar <-- car(y) ; car(y) <-- FAddr(oldcar) ;
           car(x) <-- oldcar ; cdr(x) <-- cdr(y) ;
      BB : oldcar <-- car(y) ; oldcdr <-- cdr(y) ;
           car(y) <-- FAddr(oldcar) ; cdr(y) <-- FAddr(oldcdr) ;
           car(x) <-- oldcar ; cdr(x) <-- oldcdr ;
      BF : oldcar <-- car(y) ; car(y) <-- FAddr(oldcar) ;
           car(x) <-- oldcar ; cdr(x) <-- son ;
      FA : car(x) <-- son ; cdr(x) <-- cdr(y) ;
      FB&FF : car(x) <-- son ; cdr(x) <-- car(y) ; car(y) <-- copyson ;
    endcase ;
    son <-- x ; copyson <-- y
  end ;
return copy                             - x now points to the root of the
                                        - original list structure.
end                                     - y now points to the root of the
                                        - copy.
```

After initializing the stack and the reverse preorder chain denoted as Chain, pointing x at the list to be copied (root), and saving the location of the copy in the variable copy, Pass1 does the following for each cell encountered during the traversal :

1. sets a mark bit of it to 1 ;

2. saves its original car and cdr in oldcar and oldcdr ;

3. points y at avail, which is the head of free list ;

4. plants its forwarding link y in its car ;

5. pushes it to the reverse preorder chain ;

6. performs a sequence of operations that depend on its type, including calculation of its successor in the traversal ; and

7. loops.

Consider a type FB cell x, for example. The required type-specific operations are these : oldcdr is saved in car(y); as oldcdr is a back pointer and its new value FLink(oldcdr) can be given by a forwarding link in Pass2, so its new value is not necessary to be saved, but storage constraints permit it to be saved. And oldcar, being the only forward pointer in x, becomes the successor of x in the traversal.

Operations required for a type FF cell are these : because the list will be traversed in reverse preorder in Pass2, oldcdr must be saved in car(y); copy cell is pushed onto the stack, as discussed in Section 2.3. Then a forward pointer oldcar becomes the successor of x to continue the preorder traversal.

In type AF, new values oldcar and avail are written to the car and cdr in the copy. They are final values for the copy and will not be changed for ever.

When one of the leaves of the spanning tree, a cell of type AA, AB, BA, or BB, becomes the current cell, the procedure Pop1 defined in Section 2.3 is called to find the next cell to visit. Once the popped cell appears to be inscrutable, it must, of course, get the type FB treatment. And the stack is popped up to find a type FF cell. On the other hand, when a popped cell is found to be still type FF, avail, which is the next copy cell to be visited and is a new value of the cdr, is stored to the cdr of the copy cell. It is because both old and new cdr pointer of a type FF cell must be saved, as discussed in Section 2.4. And the cdr of the popped cell will then be returned. Notice that both cells of type FB and FF have the same new values in their copy cells, an oldcdr in the car, and a new value of the original cdr in the cdr at the end of Pass1. The reason is that, because forwarding links will be removed during the reverse preorder traversal of Pass2, a type FB cell which might be an inscrutable cell must get the same treatment as type FF. Figure 4 illustrates the state of the list structure shown in Figure 2 at the end of Pass1.

Now that all original cells are pushed onto the reverse preorder chain during the preorder traversal of Pass1, all cells are processed only once by popping it up in Pass2.

Continuing to the execution of the first pass of ListCopy1, Pass2 does the following for each cell encountered during the reverse preorder traversal :

1. pops an original cell x off the reverse preorder chain;
2. points y at its copy cell;
3. performs a sequence of operations that depend on its type;
4. points son and copyson at x and y, respectively; and
5. loops.

Because in Pass2 the list is traversed in reverse preorder, x becomes son, i.e., left son (in the case that x is type FA, FB, or FF) or right son (in the case x is type AF or BF), at the next loop. Y also does copyson, i.e., left copyson or right one.

Consider again cells of type FB and FF which get the same treatment in Pass2 : car(x) is recovered by son, that is, a pointer to an original

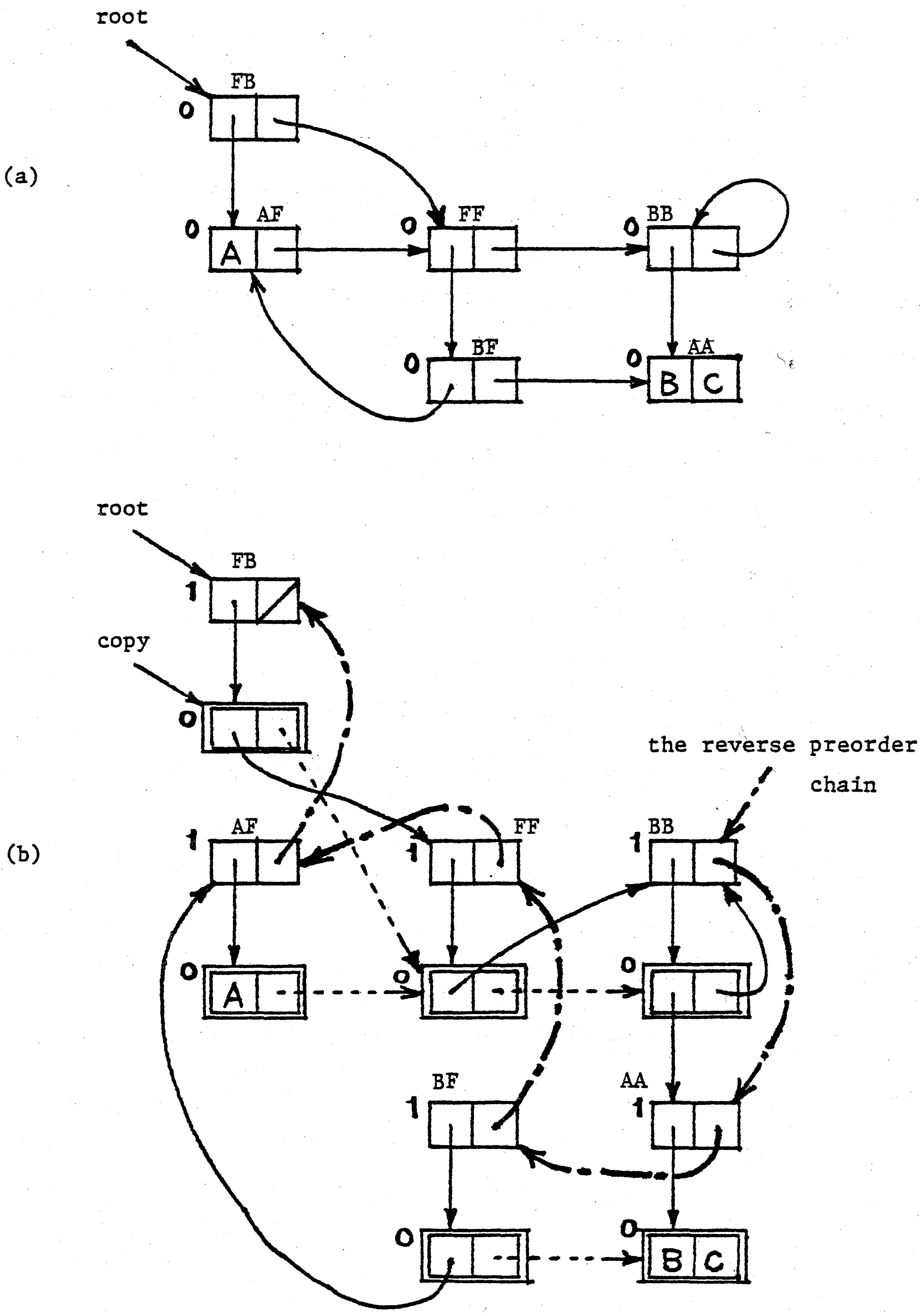


Fig. 4. Copying a list by ListCopy1 : (a) initial state, the same as Fig. 2; (b) the state at the end of Pass 1. Original pointers and forwarding links are solid; copy pointers are dotted; the reverse preorder chain is indicated by the disconnected line. Copy cells are doubly enclosed.

cell just processed before; cdr(x) is restored from car(y), where the original cdr was saved; and car(y), a forward pointer, gets the new value by copyson.

3. Copying List to Contiguous Storage of Locations : ListCopy2

While the algorithm ListCopy1 needs a mark bit to copy an arbitrary list structure, the algorithm ListCopy2, described in this section, needs no mark bits but depends on the restriction that copy must be made to contiguous storage of locations. Since ListCopy2 works under the same strategy as ListCopy1, only the differences between them are described here.

In the first pass of ListCopy2, the original list is traversed in preorder, and a forwarding link is written in car of each original cell in order to link that cell to its copy, and then each original cell is pushed onto the reverse preorder chain with saving the necessary information in its copy. Now that new list region is a block of contiguous storage locations, whether a forwarding link has been planted in car of an original cell can simply be detected by comparing the pointer found in car of an original cell with the address boundaries of the region. Thus, instead of a mark bit used in ListCopy1, we introduce the predicate new(x). The predicate new(x) will be true if and only if x points to a cell in the new list area.

Pointers for each cell type at the end of Pass1 are shown in Figure 5. Original cells are occupied by both forwarding links and the reverse preorder chain. As discussed in Section 2.4, the necessary information that must be saved at the end of Pass1 is as follows : because the new value of an atom is always the same as its old value and that of a back pointer can be given by a forwarding link during the traversal of Pass2, the new values of atoms and back pointers need not be saved.

Forward pointers of type AF, BF, FA and FB cells and that of car of

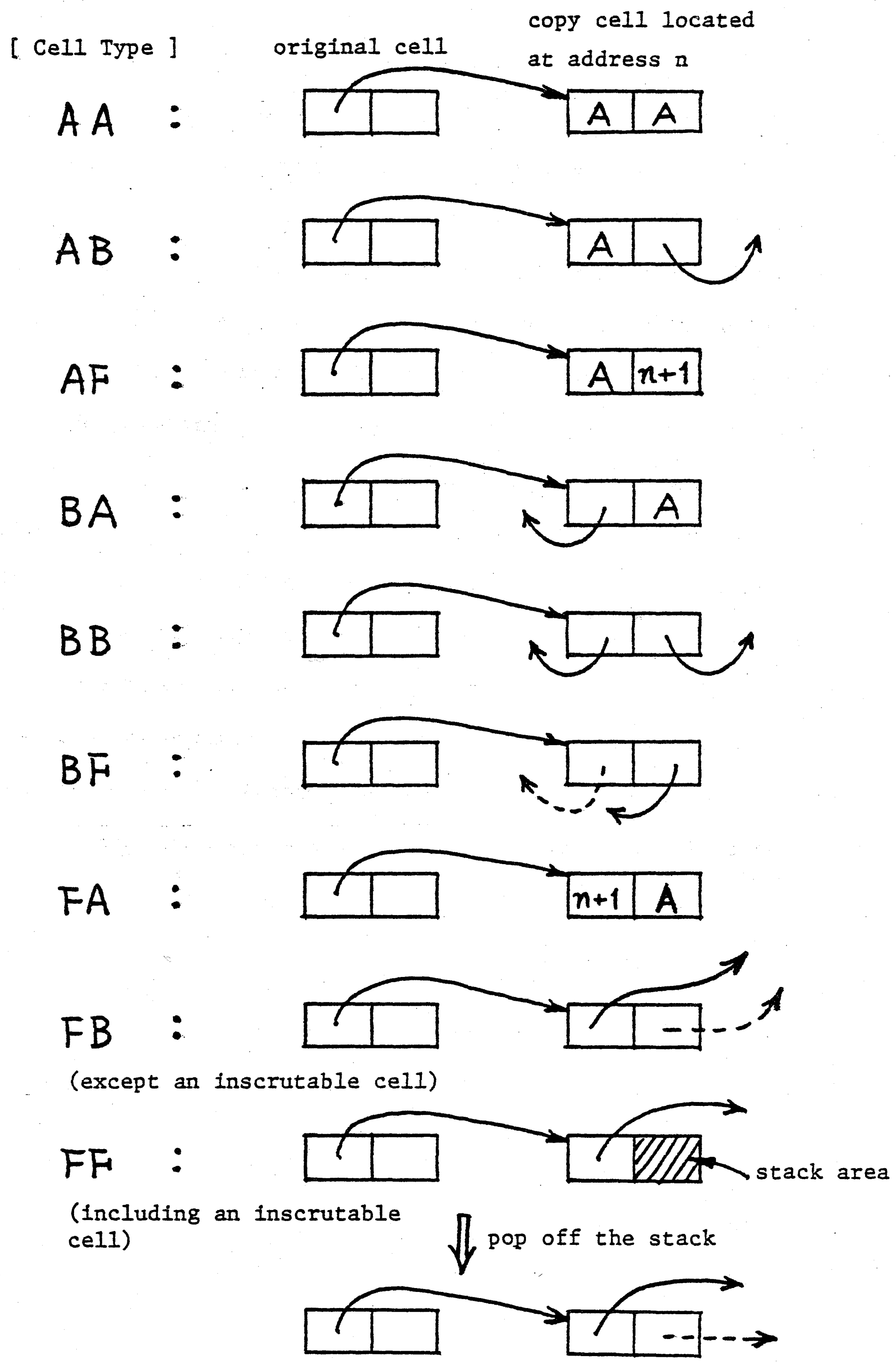


Fig. 5. State of each cell type at the end of Pass1 of ListCopy2. Atoms are denoted as A; original pointers are solid; copy pointers are dotted. A reverse preorder chain omitted is located in cdr of each original cell.

a type FF cell are not necessary to be saved, since old and new values of their forward pointers can be recovered by the reverse preorder traversal of Pass2.

Note that because new cells are contiguous, the new value of a forward pointer except that of cdr of a type FF cell is always the address of the following cell in the copy area. Then the new value of them located in cell n is n+1. Thus the variable copyson used in ListCopy1 does not appear in ListCopy2.

On the other hand, both old and new cdr pointers of a type FF cell point to a cell already processed during Pass2, because cdr is traversed before car. And also, those of a type FB cell that might be inscrutable do so. Thus these values must be saved at the end of Pass1.

Consequently, forwarding links are removed in reverse preorder in Pass2, popping up the reverse preorder chain. The procedure ListCopy2 will be shown in Appendix with all auxiliary procedures.

4. Analysis and Comparison of the Algorithms

Clark[3,5] compared his linear algorithm with Fisher's linear one by looking at how much computational work is done per cell rather than at the number of cell visits. We also compare our two linear algorithms, ListCopy1 and ListCopy2, with Clark's algorithm, following to his analysis method. His analysis uses as its measure the number of times a list cell must be read or written by each algorithm. This leads to a considerable simplification in the analysis and the comparison task by ignoring such things as arithmetic operations, instruction fetches, and the time required by Lisp primitives such as atom(x). For systems in which reading or writing a list cell is expensive relative to, say, fetching an instruction --- for example, a system where the copying algorithm is microprogrammed or resides in a cache --- the analysis done here realistically measures the computational effort involved. In other systems the work measured here is just part of the total.

It is assumed that car and cdr are contained in a single word of memory. It is also assumed that a certain amount of straightforward optimization with respect to this measure, e.g., if car(x) and cdr(x) are both read (written) in a single iteration of an algorithm, we will say that one memory read (write) has taken place. For a given n-cell list structure, let a be the fraction of cars that point to lists; let d be the fraction of cds that do so. Let f be the fraction of type AA cells; let b be the fraction of type BB cells. Let k_1 be the fraction of cells that go on the stack during Pass1; let k_2 be the fraction of type FF cells.

The analysis of the first pass of ListCopy1 is as follows : each cell of the original structure is read once for every pointer to it. The first of these reads occurs when the trace first encounters that cell; the rest occur in order to obtain the forwarding link stored there. The number of list pointers, and therefore the number of reads of original cells, is an+dn+1 (the 1 comes from the pointer root). When new list cells are got from the free-list, more n reads will occur. Then because cells pushed onto the stack will be popped up, another k_1n reads will occur. Write operations occur when each of forwarding link is written (another n), and when popped cells have their contents altered (k_1n writes). Thus the totals are $an+dn+k_1n+1$ reads and $2n+k_1n$ writes.

In Pass2, n reads of original cells will occur by following the reverse preorder chain. And also copy cells will be read at that time. Another reads of original cells will occur when each type of cells except AA, AF and FA is read in order to classify their cell types and to retrieve the forwarding links. Then $n+an+dn+k_2n+1$ reads will occur during Pass2. Since original cells will have their original cars and cdrs restored, n writes will occur for this purpose. Because copy cells of type AA, AF and FA have their final values at the end of Pass1, writes to the cells except them will occur. Thus $an+dn-bn+k_2n+1$ write operations will occur during Pass2.

The grand totals for the algorithm ListCopy1 are

$$T_{L1} = (4+3a+3d-b+2k_1+2k_2)n+3$$

memory operations on list cells to copy an n-cell structure.

Similiarly, Pass1 of ListCopy2 will execute $(2+a+d+2k_1)n+1$ reads and writes of list cells. And Pass2 will execute $(2+a+d-f+2k_2)n+2$ memory accesses. Thus the sum of ListCopy2 will execute

$$T_{L2} = (4+2a+2d-f+2k_1+2k_2)n+3$$

reads and writes of list cells to copy n-cell structure.

Clark's algorithm [5] will execute

$$T_C = (5+2a+d+3b+2k_1+2k_2)n+1$$

memory operations on list cells to copy an n-cell structure.

Since ListCop2 and Clark's algorithm run under the same restriction, we will compare them. Then it can be easily shown that T_{L2} is always less than T_C . Because the non-negative parameters d, b and f are always less than 1, then

$$T_{L2} < T_C ,$$

neglecting constants for large n.

Both Pass1s of ListCopy2 and Clark's algorithm have the same speed. But the difference of the traversal order in Pass2 of each algorithm greatly affect their speed. Clark's algorithm needs the special phase between Pass1 and Pass2 to treat the type BB cell. And a type FF cell is accessed twice, because it is pushed onto the stack and popped according to the same traversal order as in Pss1. In contrast with Clark's algorithm, the second pass of ListCopy2 will execute only one access for every cell, because list is traversed in reverse preorder by the reverse preorder chain.

Table I shows the number of list cell references that each algorithm make for a variety of n-cell list structure. (The constant in each algorithm have been discarded.)

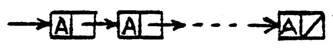
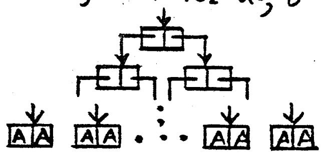
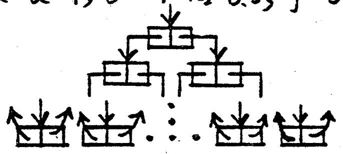
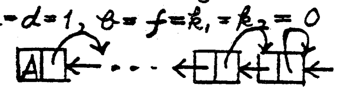
List Structure	list-cell references			$\frac{T_C}{T_{L2}}$
	ListCopy1 T_{L1}	ListCopy2 T_{L2}	Clark's Algorithm T_C	
(a) List of atoms $a=b=f=k_1=k_2=0,\ d=1$ 	7n	6n	6n	1
(b) Balanced binary tree $a=d=f=k_1=k_2=0.5,\ b=0$ 	9n	7.5n	8.5n	1.13
(c) Worst case for Clark's algorithm $a=d=1,\ b=k_1=k_2=0.5,\ f=0$ 	11.5n	10n	11.5n	1.15
(d) Worst case for Fisher's algorithm $a=d=1,\ b=f=k_1=k_2=0$ 	10n	8n	8n	1
(e) Typical Lisp case[4] $a=0.333,\ d=0.75,\ b=0.0015$ $f=0.167,\ k_1=0.23,\ k_2=0.213$	8.13n	6.89n	7.3n	1.06

Table I. Comparison of ListCopy1, ListCopy2 and Clark's algorithm.

Acknowledgement

We would like to thank Dr. Osamu Ishii and Dr. Akio Tojo for providing a chance of the present study. We are also grateful to the members of Information Systems Section for their helpful discussions.

References

1. Cheney,C.J. A nonrecursive list compacting algorithm. Comm.ACM 13, 11 (Nov. 1970), 677-678.

2. Clark,D.W. An efficient list-moving algorithm using constant workspace. Comm. ACM 19, 6 (June 1976), 352-354.

3. Clark,D.W. List structure: Measurements, algorithms, and encodings. Ph.D. Th., Dept. of Comptr. Sci., Carnegie-Mellon U., Aug. 1976.

4. Clark,D.W., and Green, C.C. An empirical study of list structure in Lisp. Comm.ACM 20, 2 (Feb. 1977), 78-87.

5. Clark,D.W. A fast algorithm for copying list structures. Comm.ACM 21, 5 (May 1978), 351-357.

6. Fisher,D.A. Copying cyclic list structures in linear time using bounded workspace. Comm.ACM 18, 5 (May 1975), 251-252.

7. Knuth,D.E. The Art of Computer Programming, Vol.1: Fundamental Algorithms. Addison-Wesley, Reading, Mass., 1969, p.417.

8. Lindstrom,G. Copying list structures using bounded workspace. Comm.ACM 17, 4 (April 1974), 198-202.

9. McCarthy,J. Recursive functions of symbolic expressions and their computation by machine - I. Comm.ACM 3, 4 (April 1960), 184-195.

10. Robson,J.M. A bounded storage algorithm for copying cyclic structures. Comm.ACM 20, 6 (June 1977), 431-433.

11. Schorr,H., and Waite,W. An efficient machine-independent procedure for garbage collection in various list structures. Comm.ACM 10, 8 (Aug. 1967), 501-506.

Appendix

An algorithm, called ListCopy2, which copys list structure into a block of contiguous storage of locations appears below.

```
procedure ListCopy(root) :
begin pointer x, copy ;
  - Pass 1
  Initialize(Stack) ; Initialize(Chain) ;
  x <-- root ; copy <-- n ;            - global variable n points to
                                       - first free cell
 until x = NIL do
   begin pointer oldcar, oldcdr ;
     oldcar <-- car(x) ; oldcdr <-- cdr(x) ;
     car(x) <-- n ; Push(x, Chain) ;   - n is the forwarding link
                                       - of cell x
    case CellType1(oldcar, oldcdr) of
       AA&AB&BA&BB : car(n) <-- oldcar ; cdr(n) <-- oldcdr ;
                     x <-- Pop1(Stack) ;
       AF : car(n) <-- oldcar ; cdr(n) <-- n + 1 ; x <-- oldcdr ;
       BF : car(n) <-- FAddr(oldcar) ; cdr(n) <-- oldcar ; x <-- oldcdr ;
       FA : car(n) <-- n + 1 ; cdr(n) <--oldcdr ; x <-- oldcar ;
       FB : car(n) <-- oldcdr ; cdr(n) <-- FAddr(oldcdr) ; x <-- oldcar ;
       FF : car(n) <-- oldcdr ; Push(n, Stack) ; x <-- oldcar ;
    endcase ;
     n <-- n + 1
   end ;
  - Pass 2
  until Empty(Chain) do
   begin pointer oldcar, oldcdr, son ;
     x <-- Pop(Chain) ; n <-- n - 1 ;
    case CellType2(n) of
       AA : car(x) <-- car(n) ; cdr(x) <-- cdr(n) ;
       AB : oldcdr <-- cdr(n) ; cdr(n) <-- FAddr(oldcdr) ;
            car(x) <-- car(n) ; cdr(x) <-- oldcdr ;
       AF : car(x) <-- car(n) ; cdr(x) <-- son ;
       BA : oldcar <-- car(n) ; car(n) <-- FAddr(oldcar) ;
            car(x) <-- oldcar ; cdr(x) <-- cdr(n) ;
       BB : oldcar <-- car(n) ; oldcdr <-- cdr(n) ;
            car(n) <-- FAddr(oldcar) ; cdr(n) <-- FAddr(oldcdr) ;
            car(x) <-- oldcar ; cdr(x) <-- oldcdr ;
       BF : car(x) <-- cdr(n) ; cdr(x) <-- son ; cdr(n) <-- n + 1 ;
       FA : car(x) <-- son ; cdr(x) <-- cdr(n) ;
       FB&FF : car(x) <-- son ; cdr(x) <-- car(n) ; car(n) <-- n + 1 ;
    endcase ;
     son <-- x
   end ;
  return copy                        - x now points to the root of the
                                     - original list structure.
end                                  - n now points to the root of the
                                     - copy. ( n = copy )

procedure FAddr(listcell) : return car(listcell)
procedure Already_Visited(listcell) : return new(FAddr(listcell))

procedure Initialize(stack) : stack <-- NIL

procedure Empty(stack) : return stack = NIL

procedure Push(cell, stack) :
  begin
    cdr(cell) <-- stack ;
    stack <-- cell
  end

procedure Pop(stack) :
  begin pointer t ;
    t <-- stack ;
    stack <-- cdr(stack) ;
    return t
  end

procedure Pop1(stack) :
  begin
    until Empty(stack) do
      begin pointer newcell, oldcdr ;
        newcell <-- Pop(stack) ;
        oldcdr <-- car(newcell) ;
        if Already_Visited(oldcdr) then    - newcell is the copy of an
           cdr(newcell) <-- FAddr(oldcdr)  - inscrutable cell.
        else
          begin                            - newcell is the copy of cell
            cdr(newcell) <-- n + 1 ;       - still type FF.
            return oldcdr                  - oldcdr is about to be
          end                              - copied into cell n + 1
      end ;
    return NIL                             - only if stack is empty.
  end

procedure CellType1(oldcar, oldcdr) :
  begin
    return
    if not(atom(oldcar)) or not(Already_Visited(oldcar)) then
      if atom(oldcdr) then FA
      else if Already_Visited(oldcdr) then FB else FF
    else
      if not(atom(oldcdr)) or not(Already_Visited(oldcdr)) then
        if atom(oldcar) then AF else BF
      else AA&AB&BA&BB
  end

procedure CellType2(newcell) :
  begin                            - newcell is copy of oldcell.
    return
    if atom(car(newcell)) then
      if atom(cdr(newcell)) then AA
    else if cdr(newcell) = newcell + 1 then AF else AB
  else
    if atom(cdr(newcell)) then
      if car(newcell) = newcell + 1 then FA else BA
    else
      if new(car(newcell)) then BF
      else if new(cdr(newcell)) then FB&FF else BB
end
```