数理解析研究所講究録
第 396 巻 1980 年 310-318

Notes on List Marking Algorithms Using Constant Workspace

Hiroshi Hasegawa

Computer Science Division
Electrotechnical Laboratory
1-1-4 Umezono, Sakura-mura
Niihari-gun, Ibaraki Pref., 305
JAPAN

Key Words and Phrases : list processing, marking, Lisp, linear time, space complexity, constant workspace, input-restricted deque, link permutation, link reversal.

Abstract

The problem considered here is to mark an arbitrary Lisp-style list structure under the constraint of constant workspace. This constraint requires that marking 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 marked. The only storage available is that occupied by the original list structure (and the constant-size auxiliary storage independent of the structure), separate from a fixed number of program variables. Marking requires that the original structure may not be permanently destroyed and each cell of them must be marked.

Algorithms for marking list structures using constant workspace have been given by Schorr and Waite[2], Wegbreit[3] and Lindstrom[5]. Schorr and Waite's algorithm uses the link reversal technique to traverse the list and runs in time O(n) to mark arbitrary n-cell structure, assuming that each cell of the list has both a mark bit and an auxiliary tag bit. Wegbreit showed that Schorr and Waite's algorithm can be improved by using a bit table instead of an auxiliary bit of each cell. Although Schorr and Waite's algorithm and Wegbreit's run in time O(n), their speed is comparatively slower than the algorithm which uses the auxiliary stack and only a mark bit. On the other hand, Lindstrom showed that arbitrary n-cell structure can be

marked without auxiliary tag bits, depending on the link permutation method[4], but his algorithm runs in time O(nlogn).

This paper reports two new algorithms for marking an arbitrary Lisp-style list structure using constant workspace. The first algorithm M1 does not require auxiliary tag bits, i.e., requires only a mark bit in each cell and runs almost in linear time at practical cases. It is based on the link permutation method (the pointer rotation technique) and utilizes the count of mark bits of the cells being processed in order to guide the traversal of the list structure correctly. While the algorithm M1 traverses the list by the link permutation method, the algorithm M2 does it by the link reversal technique based on the same principle as the one of the algorithm M1. The second algorithm M3 assumes an auxiliary tag bit in each cell in addition to a mark bit and runs almost in the same speed as the algorithm which uses an auxiliary stack and only a mark bit. Its speed is due to the combination of the link reversal method and the usage of the small constant size of input-restricted deque, the so-called bottomless stack, independent of the size or complexity of the list to be marked.

Moreover, it is also possible that the combined approach of the first algorithm M1 or M2 and the second M3 can mark the list structure. That is, marking can be done in pracical speed by assuming that there is a mark bit in each cell and a constant-size input-restricted deque, depending on the link permutation or reversal method.

References

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

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

(3) Wegbreit, B. A space-efficient list structure tracing algorithm. IEEE Trans. Comp. C-21, 9 (Sept 1972), 1009-1010.

(4) Lindstrom, G. Scanning list structures without stacks or tag bits. Inf.Proc. Letters 2, 2 (June 1973), 47-51.

(5) Lindstrom, G. Copying list structures using bounded workspace. Comm.ACM 17, 4 (Apr. 1974), 198-222.

Algorithm M1 : marking list structures without stacks or tag bits by the link permutation method.

```
Step 1 (Initialize)
    If ROOT is atomic, halt. Otherwise, set P=-1, C=ROOT
    and ALL=0.

Step 2 (Start of marking in preorder)
    Set S=C and D=car(S).

Step 3 (At S, unprocessed)
    Set MARK(S)=1 and V=car(S). If V is atomic, go to
    step 4. Otherwise, set car(S) =cdr(S), cdr(S)=P, P=S,
    S=V and ALL=ALL+1. If S=-1, go to step 6. Otherwise,
    if MARK(S)=1, set Q=P and go to step 5. Otherwise,
    repeat this step.

Step 4 (Descend to right)
    Set C=cdr(S). If C is atomic or MARK(C)=1, go to step
    14. Otherwise, set cdr(S)=car(S), car(S)=P, P=S and
    go to step 2.

Step 5 (Find S between P and C)
    If Q=S, go to step 12. Otherwise, if Q=C, go to step
    6. Otherwise, set Q=cdr(Q) and repeat this step.

Step 6 (Prepare to search From C)
    Set Q=cdr(C) and CYCLE=1.

Step 7 (Search up)
    If Q=-1, go to step 12. Otherwise, if MARK(Q)=0, go
    to step 8. Otherwise, if Q=S, go to step 12.
    Otherwise, if cdr(Q) is atomic, go to step 8.
    Otherwise, if Q=D, set X=0, Q=S and go to step 9.
    Otherwise, if Q=C, go to step 11. Otherwise, set
    Q=cdr(Q), CYCLE=CYCLE+1 and repeat this step.

Step 8 (Search up to left)
    Set Q=car(Q) and go to step 7.

Step 9 (Count up to ROOT)
    If Q=-1, set X=ALL-X-CYCLE, ALL=ALL-X and go to step
    10. Otherwise, if MARK(Q)=0 or cdr(Q) is atomic, set
    Q=car(Q) and repeat this step. Otherwise, set
    Q=cdr(Q), X=X+1 and repeat this step.

Step 10 (Back nodes)
    Set MARK(P)=0, V=cdr(P), cdr(P)=car(P), car(P)=S,
    S=P, P=V and X=X-1. If X=0, go to step 12. Otherwise,
    repeat this step.

Step 11 (Back nodes to C)
    Set MARK(P)=0, V=cdr(P), cdr(P)=car(P), car(P)=S,
        S=P, P=V and  ALL=ALL-1.  If  S=C,  go  to  step  15.
        Otherwise, repeat this step.

Step 12 (Process right descendant)
        Set  ALL=ALL-1  and  C=car(P).  If C is not atomic or
        MARK(C)=0, set MARK(P)=0, car(P)=cdr(P), cdr(P)=S and
        go  to  step  2. Otherwise,  set  V=cdr(P),   cdr(P)
        =car(P), car(P)=S.

Step 13 (Go up)
        Set S=P and P=V.

Step 14 (Done?)
        If P=-1, halt.

Step 15 (Up from right or left)
        If  MARK(P)=0,  set  MARK(P)=1  and  go  to  step 16.
        Otherwise, if cdr(P) is not atomic, go to step 12.

Step 16 (Go up from right)
        Set V=car(P), car(P)=cdr(P), cdr(P)=S and go to  step
        13.
```

Algorithm M2 : marking list structures without stacks or tag bits by the link reversal method.

```
Step 1 (Initialize)
       If ROOT is atomic, halt. Otherwise, set P=-1, C=ROOT
       and ALL=0.

Step 2 (Start of marking in preorder)
       Set S=C and D=car(S)

Step 3 (At S, unprocessed)
       Set MARK(S)=1 and V=car(S). If V is atomic, go to
       step 4. Otherwise, set car(S)=P, P=S, S=V and
       ALL=ALL+1. If MARK(S)=1, set Q=P and go to step 5.
       Otherwise, repeat this step.

Step 4 (Descend to right)
       Set C=cdr(S). If C is atomic or MARK(C)=1, go to step
       14. Otherwise, set cdr(S)=P, P=S and go to step 2.

Step 5 (Find S between P and C)
       If Q=S, go to step 12. Otherwise, if Q=C, set
       Q=car(C), CYCLE=1 and go to step 6. Otherwise, set
       Q=car(Q) and repeat this step.

Step 6 (Search up)
       If Q=-1, go to step 12. Otherwise, if MARK(Q)=0, go
       to step 7. Otherwise, if Q=S, go to step 12.
       Otherwise, if car(Q) is atomic, go to step 7. Other-
       wise, if Q=D, set X=ALL-CYCLE, V=-1 and go to step 8.
       Otherwise, if Q=C or Q=P, go to step 11. Otherwise,
       set Q=car(Q), CYCLE=CYCLE+1 and repeat this step.

Step 7 (Search up to right)
       Set Q=cdr(Q) and go to step 6.

Step 8 (Prepare to back)
       Set Q=cdr(P)

Step 9 (Back a node)
       If Q=V, set MARK(P)=0, V=car(P), car(P)=S, S=P, P=V,
       X=X-1 and ALL=ALL-1. If X=0, go to step 12.
       Otherwise, set V=S and go to step 8.

Step 10 (Count up nodes)
       If MARK(Q)=0 or car(Q) is atomic, set Q=cdr(Q) and go
       to step 9. Otherwise, set Q=car(Q), X=X-1 and go to
       step 9.

Step 11 (Back nodes to C)
       Set MARK(P)=0, V=car(P), car(P)=S, S=P, P=V and
       ALL=ALL-1. If S=Q, go to step 15. Otherwise, repeat
       this step.
```

```
Step 12 (Process right descendant)
      Set ALL=ALL-1 and C=cdr(P). If C is not atomic or
      MARK(C)=0, set MARK(P)=0, cdr(P)=car(P), car(P)=S and
      go to step 2. Otherwise, set V=car(P) and car(P)=S.

Step 13 (Go up)
      Set S=P and P=V.

Step 14 (Done?)
      If P=-1, halt.

Step 15 (Up from right or left)
      If MARK(P)=0, set MARK(P)=1 and go to step 16.
      Otherwise, if car(P) is not atomic, go to step 12.

Step 16 (Go up from right)
      Set V=cdr(P), cdr(P)=S and go to step 13.
```

Algorithm M3 : the combined algorithm of stack and link reversal for list marking.

Use STACK[A], STACK[A+1] , ..., STACK[B](A≤B)

```
Step 1 (Initialize)
      If ROOT is atomic, halt. Otherwise, set P=-1, S=ROOT,
      BOTTOM=TOP=A and COUNT =0.

Step 2 (Start of marking in preorder)
      Set X=S.

Step 3 (At X, unprocessed)
      Set MARK(X)=1 and V=car(X). If V is atomic or
      MARK(V)=1, go to step 6. Otherwise, if stack-full(),
      set Y=STACK[BOTTOM]. Otherwise, go to step 5.

Step 4 (Release the bottom of the stack)
      If S=Y, set AUXBIT(S)=1, V=car(S), car(S)=P, P=S,
      S=V, release() and go to step 5. Otherwise, set
      AUXBIT(S)=0, V=cdr(S), cdr(S)=P, P=S, S=V and repeat
      this step.

Step 5 (Descend to left)
      Push(X), set X=car(X) and go to step 3.
```

```
Step 6 (Descend to right)
          Set X=cdr(X). If X is atomic or MARK(X)=1, go to step
          7. Otherwise, go to step 3.

Step 7 (Pop up the stack)
          If stack-empty(), go to step 9. Otherwise, set
          X=pop() and go to step 6.

Step 8 (Go up)
          Set S=P and P=V.

Step 9 (Done?)
          If P=-1, halt.

Step 10 (Go up from right or left descendant)
          Set V=cdr(P). If AUXBIT(P)=0, set cdr(P)=S and go to
          step 8. Otherwise, if V is atomic or MARK(V)=1, set
          V=car(P), car(P)=S and go to step 8. Otherwise, set
          AUXBIT(P)=0, cdr(P)=car(P), car(P)=S, S=V and go to
          step 2.
```

Auxiliary Functions and Predicates :

push(X) : Set STACK[TOP]=X and COUNT=COUNT+1. If TOP=B, set TOP=A. Otherwise, set TOP=TOP+1.

pop() : Step 1. If TOP=A, set TOP=B. Otherwise, set TOP=TOP-1. Step 2. Set COUNT=COUNT-1 and return STACK [TOP].

release() : Set COUNT=COUNT-1. If BOTTOM=B, set BOTTOM=A. Otherwise, set BOTTOM=BOTTOM+1.

stack-full() : If TOP=BOTTOM and COUNT=0, return TRUE. Otherwise, return FALSE.

empty() : If COUNT=0, return TRUE. Otherwise, return FALSE.