記号処理 12－5
(1980. 6. 19)

# 木構造をコピーする高速汎用アルゴリズム

長谷川 洋（電子技術総合研究所）

## 1. はじめに

リストや木などのたどり（traversal）は、一般に補助スタックや補助キューなどを補助作業領域として使用することにより基本的には実現できる。しかし、安易に補助作業領域を使用することは、記憶の効率的使用という面からも好ましくなく、例えばくず集め（garbage collection）でのリスト・マーキング（list marking）では弊害すら生じている。

プログラム変数を除いては、処理する対象によって占められる領域以外の補助作業領域を一切使用しないアルゴリズムがあり、リスト処理では特に、固定作業領域アルゴリズム（constant / bounded workspace algorithm）と呼ばれ重視されている。

リストや木のコピー（list copying / tree copying）は、リスト・マーキングやリストの移動（list moving）などと同様に、リスト処理において重要なものとして広く認識され、その固定作業領域アルゴリズムは活発に研究されている。移動とコピーとは原構造（original structure）のコピーが作成されるという点では同一であるが、アルゴリズムの実行終了時に、移動では原構造が破壊されているのに対し、コピーでは原構造が保存される。したがって、移動では、原構造中にアルゴリズムの実行情報を保存し易いのに対し、コピーではそれらを保存するスペースが大きく制約されており、移動よりも困難な問題として認識されている。そのコピーに関しては、種々の固定作業領域アルゴリズムが示されている。

Lindstrom[5]はポインタ循環法（link permutation method / pointer rotation method）により2進木のコピーを行う固定作業領域アルゴリズムを示し、Lindstrom[6], Robson[7]はこれを発展させて、ポインタ循環法を用いて任意のリスト構造をタグ付きあるいはタグなしの自由リスト（avail list）へコピーするアルゴリズムを実現している。しかし、ポインタ循環法では、処理速度の点で補助作業領域を用いる場合よりも劣るため、Clark[1]は、コピーが作成される領域は自由リストより取り出すのではなく、コピーは連続領域に作成されるという制限条件をつけた2進木をコピーする固定作業領域アルゴリズムを示した。Clarkの方法は、作成しつつある2進木のコピー中にスタックを作成し、その結果、補助作業領域を用いる場合よりも高速なコピーの作成を実現している。さらにClark[2]は、彼の内部埋め込みスタック方式により、任意のリストを連続領域へコピーするアルゴリズムを示し、長谷川[3]は、Clark[2]より更に高速であるばかりでなく、コピーをタグ付き自由リストと連続領域の双方に作成できる固定作業領域を示している。

---

〔註〕 Lee[4]は、著者とは独立に、本稿で示すアルゴリズムTreeCopy1と類似したアルゴリズムを示している。しかし、LeeのアルゴリズムよりもTreeCopy1の方が洗練されているばかりでなく、効率の上からもTreeCopy1の方が優れている。LeeのアルゴリズムとTreeCopy1との比較は才4節において示す。

この結果、2進木をコピーする固定作業領域アルゴリズムとしては、効率は悪いが汎用である Lindstrom[5] のアルゴリズムと、コピーが連続領域に作成されるという制限のついた Clark[1] の高速アルゴリズムが、従来の最良アルゴリズムであった。したがって、タグなしあるいはタグ付き自由リストへの2進木のコピーに関しては、補助スタックを使用するコピー・アルゴリズムが最速であった。

本論文で示す2進木のコピー・アルゴリズム TreeCopy1 は、Lindstrom[5] のアルゴリズムの汎用性と、Clark[1] のアルゴリズムの高速性を兼ね備えた1パスの高速汎用アルゴリズムであり、次の様な特徴をもつ。

(1) コピーは、タグなし／タグ付き自由リスト，連続領域のいずれにも作成可能。
(2) Clark[1] の内部スタック方式を一般化した固定作業領域アルゴリズムであるため、連続領域へのコピーでは Clark[1] と同一の効率を有する。また、自由リストへのコピーの場合には、Lindstrom[5] のアルゴリズムよりも高速であるだけでなく、補助スタックを用いるアルゴリズムよりも高速である。
(3) コピー作成中に原木 (original tree) を変更しない。そのため、複数のコピー・アルゴリズムの同時実行による複数のコピーの同時作成が可能である。
(4) アルゴリズム自身を容易に並列アルゴリズムとすることができる。
(5) 2進木のコピーから n進木のコピーへ容易に拡張することが可能である。

以下に続く才2節では、2進木の汎用コピー・アルゴリズム TreeCopy1 について述べ、TreeCopy1 を連続領域へ適用した場合のアルゴリズム TreeCopy2 については才3節で述べる。才4節では、種々の2進木のコピー・アルゴリズムの解析を行う。TreeCopy1 での考え方を発展させ、一般の任意のリストのコピーに適用した場合については、才5節で述べる。

## 2. 2進木の汎用コピー・アルゴリズム TreeCopy1

### 2.1. 2進木とコピー領域の性質

2進木は、LISPタイプの car, cdr の2つのポインタを含むセルから構成され、ポインタは2進木を構成する他のセルまたはアトムを指す。部分木の共有や再入ポインタは存在しないものとし、又、各セルにはタグは付いていないものとする。アトムを指すポインタをAポインタ、セルを指すポインタをFポインタと呼ぶ。car, cdr のポインタがAまたはFポインタかにより、2進木を構成するセルは、図1に示すように、AA, AF, FA, FF の4種類のタイプに分類される。

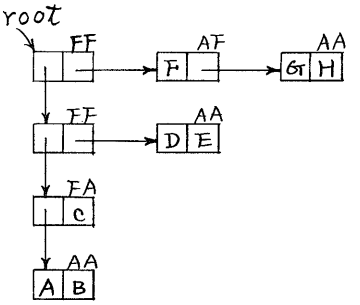


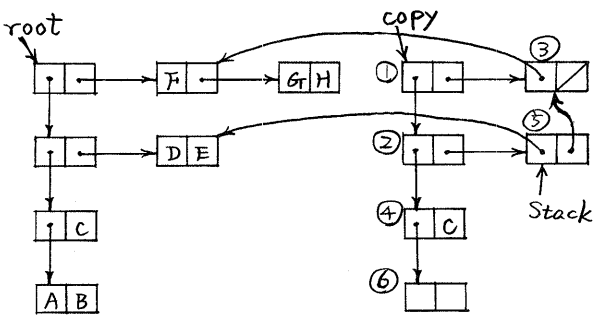

図1. 2進木
(各セルの肩にセル・タイプを示す)

コピー領域としては、一般に、タグなし自由リスト，タグ付き自由リスト，連続領域の3種類が存在する。ここで、例えば、コピー領域としてのタグなし自由リストとは、コピーを構成するセルが、各セルにタグの付いていないセルから構成される自由リストより取り出されることを意味する。また、連続領域へのコピーの場合には、コピーがある番地より始まる連続領域に作成されることを意味する。前節で示した種々のコピー・アルゴリズムは、それぞれ上記3種類のいずれかのコピー領域において動作する。アルゴリズムを開発する上での困難さという観点からコピー領域を見れば、コピー領域が、連続

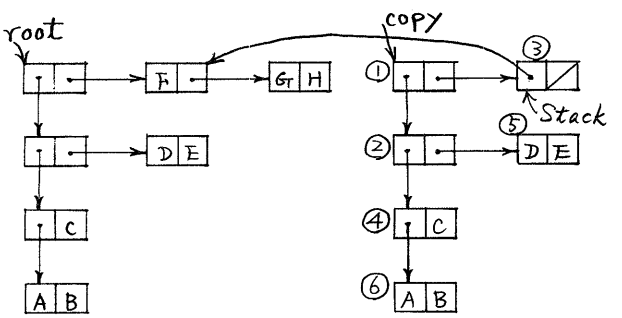
領域，タグ付き自由リスト，タグなし自由リストの順で困難さが増してゆく。その理由は、コピーを連続領域に作成する場合には、ある番地以降よりコピー領域が始まるということがタグ付き自由リストのタグと同等の情報を有し、それに加えて、連続番地という性質を利用することができるからである。従って、これらの情報を一切利用できないタグなし自由リストへのコピーの場合には、原(original)セルおよびコピー・セルのcar, cdr部分に必要な情報を格納する以外に手立てがないため、最も困難な場合と言うことができる。

## 2.2. アルゴリズムの原理

アルゴリズムTreeCopy 1では、root, car, cdrと前順序(preorder)で2進木をたどりつつ原木のコピーを作成する。原木とコピー木によって占められる以外の補助作業領域を使用しないという条件を満たしつつ、効率のよい2進木のたどりを実現するために、Clark[1]と同様に、TreeCopy 1でも作成しつつあるコピー中にスタックを作成する。しかし、ここではコピー・セルは、タグなし自由リストより取り出されるので、Clarkとは異ったスタックの作成法が必要となる。




(a). 原セル(A.B)のコピーに取りかかる状態




(b). 原セル(D.E)のコピーを終了した状態

図2. 内部スタックの実現

(各コピー・セルに付した㋑は、自由リストから取り出される順序を示す。n=1, 2, 3, ……)

外部補助スタックがコピー内部に実現する内部スタックかを問わず、スタックを用いて2進木をたどれば、下ポインタを2つ持つ下下セルのみがスタックにプッシュ(push)され、残る3つのタイプのセルはプッシュされる必要がない。そこで、rootから葉(leaf)へと原木をたどっている時には、次に出会う原セルに対応して、自由リストよりコピー・セルが1つ取り出される。もし、その原セルのタイプが下下ならば、このコピー・セルはコピー木を構成する下下セルとなり、そのcar, cdrの下ポインタは他のコピー・セルを指すことになる。そこで、原下下セルに対応するコピー・セルを自由リストから取り出すだけでなく、それに加えて、コピーの下下セルのcar, cdrの下ポインタが指すべきコピー・セルをも自由リストより先取りしておき、コピーの下下セルのcar, cdrに、その先取りしたコピー・セルの番地を格納する。この様にして、図2-(a)で、コピーのプロセスが原セル(A.B)に達する状態では、コピーの下下セル①に対してコピー・セル②, ③が、また、コピーの下下セル②に対応してコピー・セル④, ⑤が取られることになる。コピーの作成では、原木を前順序でたどりながら、原セルに対応するコピー・

セルを1セルづつ自由リストより取り出しながら先へ進むことが基本となる。したがって厳密に云うならば、例えばコピーの所セル①のcarとなるコピー・セル②は、前順序で原木をたどる上で必然的に必要となるコピー・セルであり、コピーの所セル①のcdrのコピー・セル③が先取りされたと云える。1パスのコピー・アルゴリズムでは、原木をたどる時に原セルに対応して必然的に自由リストより取り出されるコピー・セルには、その場で最終値が入れられなくてはならない。それ故、スタック領域としては、先取りされたコピーの所セルのcdrのコピー・セルが使用される。このコピー・セルを先取りした後、原木のたどりはcar方向へ進むため、原セルのcdrの所ポインタの値をスタックにプッシュすることが必要となるので、この値を先取りしたコピー・セルのcarに格納する。このコピー・セルのcdrには、変数Stackが現在指しているセル(初期値はNIL)へのポインタを格納し、新たに変数スタックがその先取りされたコピー・セルを指すことによりスタックの先頭とする。(図2-(a)参照)

原木をたどり、葉(leaf)を構成する原AAセルに出会った時には、自由リストから取り出したコピー・セルのcar, cdrに原Aポインタ

```
procedure TreeCopy1(root) :
begin pointer oldcell, copycell, copy, Stack ;
  oldcell <- root  ; copycell <- New_Cell(avail) ;
  copy <- copycell ; Initialize_Stack() ;
  until oldcell = NIL do
    begin pointer oldcar, oldcdr, copycar, copycdr ;
      oldcar <- car(oldcell) ;
      oldcdr <- cdr(oldcell) ;
      case CellType(oldcar, oldcdr) of
        AA : car(copycell) <- oldcar ;
             cdr(copycell) <- oldcdr ;
             copycell <- Pop&Reclaim_Stack() ;
             oldcell  <- Get_Next_Cell(copycell) ;
        AF : copycdr  <- New_Cell(avail) ;
             car(copycell) <- oldcar ;
             cdr(copycell) <- copycdr ;
             copycell <- copycdr ;
             oldcell  <- oldcdr ;
        FA : copycar  <- New_Cell(avail) ;
             car(copycell) <- copycar ;
             cdr(copycell) <- oldcdr ;
             copycell <- copycar ;
             oldcell  <- oldcar ;
        FF : copycar  <- New_Cell(avail) ;
             copycdr  <- New_Cell(avail) ;
             car(copycell) <- copycar ;
             cdr(copycell) <- copycdr ;
             car(copycdr)  <- oldcdr ;
             Push_Stack(copycdr) ;
             copycell <- copycar ;
             oldcell  <- oldcar ;
      caseend
    end ;
  return copy
end

procedure CellType(oldcar, oldcdr) :
  begin
    return
      if atom(oldcar)
        then if atom(oldcdr) then AA else AF
        else if atom(oldcdr) then FA else FF
  end

procedure Initialize_Stack() : Stack <- NIL

procedure Empty_Stack() : return Stack = NIL

procedure Push_Stack(cell) :
  begin
    cdr(cell) <- Stack ; Stack <- cell
  end

procedure Pop&Reclaim_Stack() :
  begin
    if Empty_Stack() then return NIL
      else begin pointer t ;
             t <- Stack ;
             Stack <- cdr(Stack) ;
             return t
           end
  end

procedure Get_Next_Cell(copycell) :
  begin
    if copycell = NIL then return NIL
      else begin pointer nextcell ;
             nextcell <- car(copycell) ;
             return nextcell
           end
  end
```

図3. アルゴリズム TreeCopy1

をそのまま格納した後、スタックはポップ(pop)され、次のコピーの作成先へ制御が移る。この時、解放されたスタックの先頭のスタック要素(図2-(b)ではコピー・セル⑤)は、原FFセルのcdrの原セルに対応するコピー・セルとして再使用されるので、自由リストから新たなコピー・セルが取り出されることはない。

原セルのタイプがFAまたはAFの場合には、コピー・セルに格納されるFポインタは次に自由リストから取り出されるコピー・セルを指し、また、Aポインタの値は原Aポインタ自身であるので、容易に原セルに対応するコピー・セルを作成できる。

以上は2進木のコピーについての議論であるが、TreeCopy1は容易に n 進木のタグなし自由リストへのコピー・アルゴリズムに拡張することができる。その場合のポイントは、先取りしたコピー・セル中に、そのコピー・セルに対応する原セルへの値を格納し、更にこれらのコピー・セルを、木のたどりに対応する順序に連鎖(chain)化することである。

## 3. 連続領域への適用：アルゴリズム TreeCopy2

タグなし自由リストへのコピーに比較して連続領域へのコピーは利用できる情報が多いこと、また、前節で述べたアルゴリズムTreeCopy1は、Clark[1]の連続領域への2進木のコピー・アルゴリズムの一般化であることは既に述べた。したがって、アルゴリズムTreeCopy1を容易に連続領域への2進木のコピー・アルゴリズムとして使用するように変えることができる。

連続領域へのコピーの場合には、コピー・セルを自由リストから1セルづつ取り出してくる必要はなく、n番地より始まる連続領域の各番地が、自由リストを構成する各セルに対応する。図4に示す連続領域へのコピー・アルゴリズムTreeCopy2の原理は、前節で述べたアルゴリズムTreeCopy1と同一である。TreeCopy2は、Clark[1]のアルゴリズムと同一の処理速度を有するだけでなく、それに加えて、Clarkのアルゴリズムにはない次の様な特徴をもつ。すなわち、FFセルのcar, cdrセルが同時に連続領域にとられるため、carがm番地ならば、cdrはm+1番地にとられることになる。(この逆も可能である。)

なお、図4中で使用される補助関数は、図3のそれと全く同一であるので、図3を参照されたい。

```
procedure TreeCopy2(root) :
  begin pointer oldcell, copycell, copy, Stack ;
    oldcell <- root ; copycell <- n ; copy <- copycell ;
    Initialize() ;
    until oldcell = NIL do
      begin pointer oldcar, oldcdr ;
        oldcar <- car(oldcell) ; oldcdr <- cdr(oldcell) ;
        case CellType(oldcar, oldcdr) of
          AA : car(copycell) <- oldcar ; cdr(copycell) <- oldcdr ;
               copycell <- Pop&Reclaim() ;
               oldcell  <- Get_Next_Cell(copycell) ;
          AF : car(copycell) <-oldcar ; cdr(copycell) <- n+1 ;
               n <- n+1 ; copycell <- n ; oldcel<- oldcdr ;
          FA : car(copycell) <- n+1 ; cdr(copycell) <- oldcdr ;
               n <- n+1 ; copycell <- n ; x <- oldcar ;
          FF : car(copycell) <- n+1 ; cdr(copycell) <- n+2 ;
               copycell <- n+1 ; n <- n+2 ; Push(n);
               oldcell <- oldcar ;
        caseend
      end ;
    return copy
  end
```

図4. アルゴリズム TreeCopy2

## 4. 解析

ここでは、Clark[2], 長谷川[3]と同様に、アルゴリズム自身はマイクロプログラムで記述されているか、又はキャッシュ・メモリ中に存在しているものとして解析を行う。従って、アルゴリズム自身の性能は、原セルおよびコピー・セルがメモリより読み出されたり、書き込まれたりする、メモリへのアクセス回数に依ることになる。また、carとcdrは1語中に含まれ、最適化に依り、car(x), cdr(x)は、同時に読み書きされるものと仮定する。

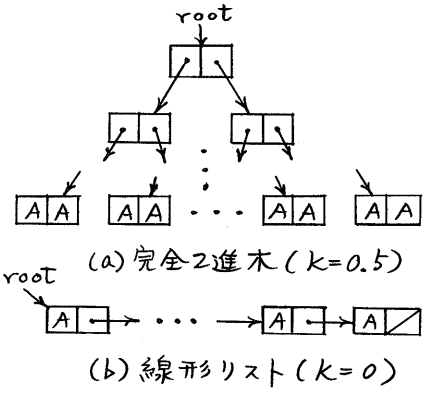



図5. 2進木の例

各パラメータを次の様に定める。

n：2進木を構成するセル数.

k：タイプ1セルの全セル数nに対する割合.（$0 \leq k \leq 0.5$）

紙面の都合により導出過程を省き、結果のみを記すると以下の様になる。

タグなし自由リストへの2進木のコピー・アルゴリズムに関しては、TreeCopy1でのメモリへのアクセス回数 $T_{TC1}$ は、

$$T_{TC1} = (3+2k)n.$$

TreeCopy1での内部スタックを外部補助スタックに置き換えた従来の基本的なコピー・アルゴリズムの場合には、そのメモリへのアクセス回数 $T_{AUX}$ は、

$$T_{AUX} = (3+4k)n$$

となる。また、Lee[4]のアルゴリズムおよびLindstrom[5]のアルゴリズムの場合には、そのメモリへのアクセス回数 $T_{Lee}$, $T_{Lindstrom}$ は、それぞれ、

$$T_{Lee} = 5n+1, \quad T_{Lindstrom} = (8-k)n-5$$

となる。表1に、図5の完全2進木および線形リストの場合についてのタグなし自由リストへの2進木のコピー・アルゴリズムの効率比較を示す。表1中で、$k=0$ の場合に、TreeCopy1と補助スタックを用いたアルゴリズムとが、同一のメモリ・アクセス回数となっているのは、線形リスト（$k=0$）の場合には、スタックが全く使用されないからである。

| アルゴリズム | セルのアクセス回数 | |
|---|---|---|
| | $k=0.5$ | $k=0$ |
| TreeCopy1 | $4n$ | $3n$ |
| 補助スタックを用いたアルゴリズム | $5n$ | $3n$ |
| Lee[4]のアルゴリズム | $5n+1$ | $5n+1$ |
| Lindstrom[5]のアルゴリズム | $7.5n-5$ | $8n-5$ |

表1. タグなし自由リストへの2進木のコピー・アルゴリズムの効率比較

連続領域への2進木のコピーに関しては、TreeCopy2のメモリ・アクセス回数 $T_{TC2}$ とClark[1]のアルゴリズムのそれ、$T_{Clark}$ とは、次の様に同一となる。

$$T_{TC2} = T_{Clark} = 2(1+k)n$$

この結果、上記の全アルゴリズムのメモリ・アクセス回数に関して、

$$T_{TC2} = T_{Clark} < T_{TC1} \leq T_{AUX} < T_{Lee} < T_{Lindstrom}$$

の関係が成り立つ。

## 5. あとがき

従来、固定作業領域アルゴリズムは、補助スタックなどを補助作業領域として使用するアルゴリズムに比較して、速度の点で劣るのではないかという見方が存在した。しかし、ことコピーに関しては、この様な見方は新アルゴリズムの出現により次々に塗り替えられつつある。このことは、内部スタック方式を基本とするTreeCopy1, TreeCopy2や長谷川[3]のタグ付き自由リストや連続領域への任意のリストのコピー・アルゴリズムに限るものではなく、ポインタ循環法を基本としたコピー・アルゴリズムで、補助スタックを使用した場合よりも高速なアルゴリズムも存在している。

現在、タグなし自由リストへ任意のリストをコピーする固定作業領域アルゴリズムとしては、Robson[7]のアルゴリズムが最速である。これに対して、任意のリスト中に予測不能セル(inscrutable cell)[2],[3]が存在しない場合には、Robsonのアルゴリズムのパス1を、ポインタ循環法には依らずに、本論文で述べたコピー・セルの先取りによる内部スタック方式に基づいたやり方に置き換えることにより、より高速なアルゴリズムを実現することができる。また、このやり方を予測不能セルを含む任意のリストのコピーの場合に適用すると、パス1終了時に、予測不能セルと同数のコピー・セルを自由リストから余分に取り出してしまう。しかし、この余分に自由リストより取り出したコピー・セルをパス2で使用することにより、Robsonのアルゴリズムよりもより高速な固定作業領域アルゴリズムを実現できる。また、このアルゴリズムは、余分に自由リストから取り出したコピー・セルを最終的に自由リストへ戻すことが可能であるだけでなく、補助スタックを使用した任意のリストのコピー・アルゴリズムよりも高速であると予想される。このアルゴリズムの詳細については、別稿において述べる。

さらに、共有部分木(shared subtree)を含む木のタグなし自由リストへのコピーに関しても、Robsonのアルゴリズムよりも高速なアルゴリズムを求めてあるが、このアルゴリズムについても、詳細は別稿において述べる。

今後なすべき事としては、本論文で示したTreeCopy1, TreeCopy2に関しては、その正当性の証明が残されている。また、コピー・アルゴリズム全般に関しては、アルゴリズムの並列化に関する研究が必要であろう。

謝辞　本研究の機会を与えられた石井治ソフトウェア部長、棟上昭男情報システム研究室長、ならびに、文献(4)の存在を筆者にいち早く教えて下さった電子計算機部計算機方式研究室弓場敏嗣主任研究官に深謝する。

## 参考文献

(1). Clark, D.W.: A fast algorithm for copying binary trees. Inf. Proc. Letters 4, 3 (Dec. 1975), 62-63.
(2). Clark, D.W.: A fast algorithm for copying list structures. C.ACM 21, 5 (May 1978), 351-357.
(3). 長谷川 洋: "高速リスト・コピー・アルゴリズム", 信学技報, AL 78-81 (1979年1月).
(4). Lee, K.P.: A linear algorithm for copying binary trees using constant workspace. C.ACM 23, 3 (May 1980), 159-162.
(5). Lindstrom, G.: Scanning list structures without stacks or tag bits. Inf. Proc. Letters 2, 2 (1973), 47-51.
(6). Lindstrom, G.: Copying list structures using bounded workspace. C.ACM 17, 4 (April 1974), 198-202.
(7). Robson, J.M.: A bounded storage algorithm for copying cyclic structures. C.ACM 20, 6 (June 1977), 431-433.