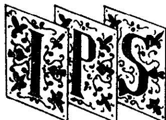
Vol. 35 No. 11　　情 報 処 理　　Nov. 1994

解 説



ごみ集めの基礎と最近の動向

1. ごみ集めの基本アルゴリズム†

日 比 野 靖††

1. はじめに

われわれが計算をするとき，大抵の場合，問題そのものや，計算結果よりも，計算の中間結果を記録するのに多くの計算用紙を費やす．しかし，その経過をよく調べてみると，始めのほうの記録は消してしまってもよいことが多い．このことは，使った計算用紙の枚数よりも，相当少ない紙数で問題が解けることを示している．

計算用紙の場合は，わざわざ消ゴムで消したりしないが，計算機を用いて計算を進めるときは，限られた資源である“メモリ空間”を有効に使うために，消ゴムに当たるものがほしくなる．数値計算のように，扱うデータの構造が変化しない場合は，メモリ空間の割付けをプログラマが行ってもあまり負担にならないが，記号処理のように計算の進行に従って構造そのものが変化するような対象を扱う場合には，プログラマの責任でメモリ管理を行うのは大変な負担である．

本来，プログラミングは対象としている問題の本質的なアルゴリズムのみを記述するのが理想であり，メモリの管理などはシステムにまかせ，処理概念の抽象化を進めるのが望ましい．多くの言語処理系，たとえば Lisp に代表される記号処理言語の処理系では，“ごみ集め”（ガーベジコレクション）という方法が使われている．

本稿では，ごみ集めのアルゴリズムを原理的な立場から整理し，解説する．最近の話題である，“実時間ごみ集め”や“分散ごみ集め”については，本稿に続き，湯浅太一氏，市吉伸行氏が解説される．

また，ごみ集めのアルゴリズムの古典的な研究については，Cohen[1] によるよい概説があるので，あわせて参照してほしい．

2. 予備知識

2.1 リスト構造

リスト処理に用いられる内部データ構造は，ポインタ連鎖によるリスト構造 (linked list structure) である[2]．

リスト構造を構成するデータ要素は“セル (cell)”または“ノード (node)”と呼ばれ，複数の“リンク（またはポインタ）フィールド”からなる．

特別の場合として，リンクフィールドがセルを指示するポインタでなく，「データ」* を格納していることがある．処理系により，これを区別する方法はことなるが**，ここでは，ガーベジコレクションのアルゴリズムを議論するのに十分なように「空リンク (null link)」で代表させる（アルゴリズム中では 0 で表現）．

以下では，Pascal ふうの記述を用いる．linki=0 のとき，空リンクであるとし，また，簡単のため，しばらくの間セルの大きさは一定であるとする．

```
D1. type cell=record link 1 : 0..M, ***
                     link 2 : 0..M, ・・
                     linkn : 0..M end;
var SPACE : array [1..M] of cell;
```

リスト構造は，D1 で定義したセルをリンクで任意に結んだものである．したがって，単純な「木 (tree)」やその集合である「林 (forest)」だけではなく，同一のセルが複数のセルから参照されている場合（多重参照）や，リンクをたどってい

† Fundamental Algorithms for Garbage Collection by Yasushi HIBINO (Japan Advanced Institute Science and Technology, Hokuriku).

†† 北陸先端科学技術大学院大学情報科学研究科

* 正確には，データまたはセル領域以外の領域へのポインタ．

** セル領域とそれ以外のアドレスを区別したり，ポインタとデータとの区別をするタグを用いる．

*** セル空間の大きさを表す整数値．

くと，もとのセルにもどってくる場合（循環リスト (circular list) を含んでいる．

このことが，ごみ集めを単純なものにしていない理由である[1]．

2.2 ルートと到達可能

リストの主発点となる特定のセルを“ルート”という．ごみ集めにおける重要な概念は，“到達可能”ということである．あるセルaからセルbへ到達可能であるとは，aからbへのリンクによる有限長のパスが存在することである．

ルートから到達可能なセルを“生きているセル (active cell)”という．

処理の対象となっているリスト（の集合）を管理するために，ルートへのリンクを保持する．ここではこのために，特別の配列 ROOT を用意する．

D2. var ROOT : array [1..N] of 0..M; N*

3. 基本アルゴリズム

3.1 アルゴリズムの分類

リスト処理の基本操作のうち，リスト構造を変化させる操作は，(1)新しいセルの生成**，(2)リンクの書換え，(3)ROOT の書換えである．

セルの供給源から新しいセルを得てリストを生成すると，セルが消費される．またリンクの書換えや ROOT の書換えが行われると，到達不可能なリストが生じる．計算の進行のある時点でセル空間を見ると，ROOT から到達可能なセルと，到達不可能なセル，および未使用のセルが混じりあった状態になる．未使用のセルを自由セルと呼ぶ***．ただし，本稿では，自由セルの管理方法（セルの供給源の管理法）にはふれない．

計算が進行すると自由セルがなくなる（セルの供給源が空になる）．この時点でガーベジコレクタ（以下 GC と略す）が起動される．ごみ集めの方法は，基本的には次の二つのフェーズに分けられる．

(1) 仕分けフェーズ：ROOT から到達可能なセルとそれ以外のものを区別する．

(2) 回収フェーズ：到達不可能なセルを回収し，自由セルとする（セルの供給源にもどす）．

自由セルがなくなったとき，この二つのフェーズを順次実行するのが古典的な「一括型のごみ集め」である．

仕分けフェーズのアルゴリズムは，次の三つに分類される．

(1. 1) 目印付け法：ROOT からリンクをたどり到達可能なセルに目印を付ける．

(1. 2) 参照カウンタ法：セルごとに自分を指しているリンクの数を示す参照カウンタを設け，リスト構造の変更操作を行うたびに，参照カウンタの増減を行う．参照カウンタが“0”のものが到達不可能なセルである．

(1. 3) 移動（複写）法：セル空間を二つの部分空間に分け，一方の部分空間（現空間）で自由セルがなくなったとき，ROOT から到達可能なセルをもう一方の部分空間（新空間）へ複写する．

回収フェーズの仕事は，自由セルの管理手法，セルの大きさの種類（単一長か，多種類か）により分類される．また仕分けフェーズのアルゴリズムによりその内容が大きく異なる．

回収フェーズの分類をしておく．

(2. 1) 詰め替え（領域圧縮 compaction）を行わず，自由リストに戻す．セルの大きさが一定のときは，これだけで十分なことが多い．

(2. 2) 詰め替えを行う．詰め替えの程度により次のように分けられる．

(2. 2. 1) 任意順：詰め替え後の並び順になんらの制約がない．

(2. 2. 2) 直線化：直接リンクのあるセルが連続した位置を占めるように詰め替える．

(2. 2. 3) 横すべり：セルの並び順を保持したまま，セル空間の一方の端に集める．

ここまでは，GC はセルの供給源が空になったとき，起動されるとして話を進めてきた．しかし，これは古典的な一括型のごみ集めの概念である．セル空間が大きくなり，またそれが仮想空間上に展開されているような環境* では，一括型のごみ集めでは，次のような問題を生じる．

一度，GC が起動されると，長時間の処理の中断が生じる．会話処理や，実時間処理の応用では，これは致命的な欠陥となる．

---

* ルートの個数を表す整数値．

** 未使用のセルを溜めてあるセルの供給源から，セルを取り出す操作．

*** 訳語としては，「空きセル」のほうが適切であると思うが慣習に従う．

* 今日では普通の環境である．UNIX ワークステーションで LISP を使用している場合を考えよ．

このような問題に対する一つの方法は，リスト処理の中にごみ集めの処理を分散してしまおうというものである．さらに直接的な方法は，ごみ集めをリスト処理と並行して行おうとするものである．

前者を即時型，後者を並行型と呼んでいる．
これらについては，本稿に続き，湯浅氏，市吉氏が解説される．

3.2 目印付け

目印を付けるため，セルに対応して1ビットの情報を付加する必要がある（tag ビット）．この目印ビット（mark bit）をセル自身に設ける方法と，すべての目印ビットを一カ所に集めたビットテーブル（bit table）を用いる方法がある．

ビットテーブルを用いる場合は，セルのアドレスからビットテーブルの対応するビット位置を求める処理が必要である．ここでは，セル自身に目印ビットを設けることにする．

```
D11. type cell=record mark : Boolean ;
                      link 1 : 0..M ;
                      link 2 : 0..M end ;
     var SPACE : array [1..M] of cell ;
```

目印付けの方法は，次の三つが代表的なものである．

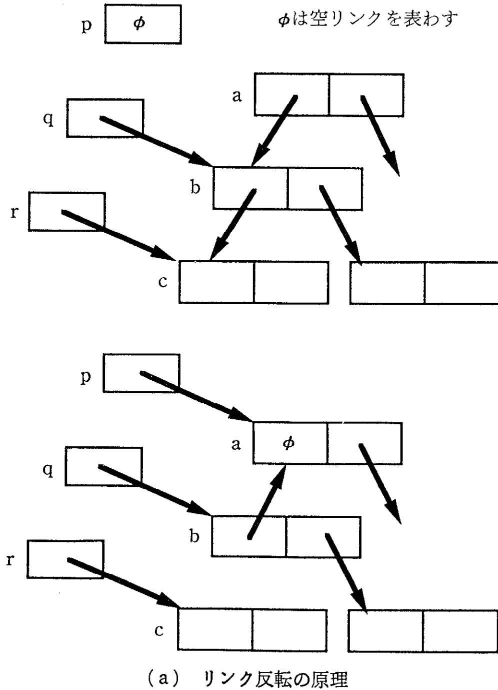
M1．リストたどり法
M2．リンク（ポインタ）反転法
M3．走査法

[M1．リストたどり法][2)]

図-1 にセルのリンクが二つの場合のアルゴリ

```
procedure mark(x) {recursive trace version}
begin
  if x≠0 then
    if SPADE[x].mark=false then
      begin
        SPACE[x].mark:=true;
        mark(SPACE[x].link 1);
        mark(SPACE[x].link 2)
      end
end
```

図-1　再帰的リストたどりによる目印付け



（a）　リンク反転の原理

```
D12. type cell=record mark: Boolean;
                      tag: Boolean;
                      link 1: 0..M;
                      link 2: 0..M end;
     var SPACE: array[1..M] of cell;
procedure mark(x) {link reversal}
begin
  if x=0 then goto finish;
  p:=0; q=x;
11: SPACE(q).mark:=true;
  r:=SPACE[q].link 1;
  if r≠0 then
    begin
      SPACE[q].link 1:=p;
      p:=q; q:=r; goto 11
    end;
  r:=SPACE[q].link 2;
12: if r≠0 then
    begin
      SPACE[q].tag:=true
      SPACE[q].link 1:=p;
      p:=q; q:=r; goto 11
    end;
13: if p=0 then goto finish;
  if SPACE[q].tag=false
    then
      begin
        r:=SPACE[p].link 1;
        SPACE[p].link 1:=q;
        q:=p; p:=r; goto 12
      end
    else
      begin
        r:=SPACE[p].linkd 2;
        SPACE[p].tag:=false;
        SPACE[p].link 2:=q;
        q:=p; p: =r; goto 13
      end;
finish:
end;
```

（b）　リンク反転法による目印付け

図-2

ズムを示す．これはよく知られた，前順序 (pre-order) の木たどり (tree traverse) アルゴリズムである．この例のように再帰的手続きを用いるかわりに，陽にスタックを用いてもよい．しかし，いずれにしろ，木をたどるためには木の高さに比例するスタック領域がいる．ガーベジコレクタが起動されたときは，すでにメモリ資源がない状態なので，この性質は望ましいものではない．この問題に対して，スタックの消費量を減らすさまざまのアルゴリズムが提案され，実用に供されている[3]．

スタックを，まったく用いないアルゴリズムとして次に述べる，リンク反転法と走査法がある．

[M 2．リンク（ポインタ）反転法][4]

図-2(a) に示すように，セルaからリンク a. link 1（セルb）をたどり，セルbを訪れるときセルbのリンク b. link 1 にアドレスaを保持する．b．link 1 の内容cは，作業域rに保持する．

もどるときは，逆向きリンクが link 1，link 2 のいずれに保存されているかを知る必要がある．そのためセルに1ビットの情報を付加する必要がある．**図-2**(b) にアルゴリズムを示す．

[M 3．走査法][5]

この方法もスタックを必要としない．まず始めに ROOT から目印を付け，あとはセル空間の走査を繰り返し，目印を到達可能なセルへ伝播させていく．

最も素朴な方法を**図-3**(a) に示す．この方法では，走査の方向とリンクの方向が逆のときは，一つ先のセルしか目印が伝播しないから，セル空間全域へのアクセスを繰り返すことになり能率の良いものとはいえない．

やや能率を改善した古典的アルゴリズムを**図-3**(b)に示す．いずれも目印付けを行うのに，最悪の場合には，目印を付けるセルの数を n，メモリサイズを M とすると，$n \times M$ に比例した時間がかかってしまい，実用上は使いものにならない．

ただし，図-3(a) のアルゴリズムは，Dijkstra ら[14]の並列ごみ集めアルゴリズムの中で採用されている．

3.3 回収フェーズ

回収フェーズのアルゴリズムは単純である．セル空間を走査し，目印の付いていないセルをセルの供給源にもどす．わすれてならないことは，目

```
procedure mark(x) {naive scanning version}
begin
  for i:=1 to N do {N is size of ROOT}
    SPACE[ROOT[i]].mark:=true;
  old_count:=0;
  repeat
    count:=old_count;
    for i:=1 to M do
      if SPACE[i].mark then
        begin
          if SPACE[i].link 1≠0 then
            begin
              SPACE[SPACE[i].link 1].mark:=true;
              count:=count+1
            end;
          if SPACE[i].link 2≠0 then
            begin
              SPACE[SPACE[i].link 2].mark:=true;
              count:=count+1
            end
        end
    until old_count=count
end
```

（a） 素朴な方法

```
procedure mark(x) {improved scanning version}
begin
  for i:=1 to N do {N is size of ROOT}
    SPACE[ROOT[i]].mark:=true;
  k:=1;
  repeat
    next:=k+1;
    if SPACE[k].mark then
      begin
        if SPACE[k].link 1≠0 then
          begin
            SPACE[SPACE[k].link 1].mark:=true;
            next:=min (next, SPACE[k].link 1)
          end;
        if SPACE[i].link 2≠0 then
          begin
            SPACE[SPACE[k].link 2].mark:=true;
            next:=min(next, SPACE[k].link 2)
          end
      end
    k:=next
  until k>M {M is size of SPACE}
end
```

（b） 改良した方法

図-3　走査法による目印付け

```
procedure reclaim
begin
  for i:=1 to M do
    if SPACE[i].mark
      then
        SPACE[i].mark:=false
      else
        restore_to_cell_pool (SPACE[i])
end
```

図-4　回収フェーズの手順

印の付いているセルの目印を消すことである．単純な回収フェーズの手順を図-4 を示す．

大きさが一定のセルの場合の回収では，上述のように単純でよい．

しかし，セルの大きさが，多種類の場合や，セルの大きさが一定でも，仮想記憶空間で問題になるようにセルの配置の局所性が問題になる場合は，後述する「詰め替え (compaction)」をこの回収フェーズの中で行う．

4. 参照カウンタを用いる方法[6)]

リスト構造は，単純な「木」または木の集まりである「林」ではない．木の中間のノードを別の木から参照していることもある．

これが多重参照といわれる状況である．この多重参照があるために，回収してよいセルと，まだ生きているセルとの区別が難しくなるのである．

3. では一般的な目印付けでこの問題を解決した．もう一つの解決法は，参照カウンタを用いる方法である．

参照カウンタを用いる方法では，「リスト構造の変更操作」すなわち，「セルの生成」，「リンクの書換え」，「ROOT への代入」などのたびに，参照カウンタの更新をしなければならない．

これは，リストの基本操作のオーバヘッドとなる．一方，参照カウンタが“0”になれば，ただちにそのセルを回収してよいことが分かるので，ごみ集めに要する時間を，基本操作の中に分散できる．したがって，この方法は即時型のごみ集めに向いている．

ただし，参照カウンタ法では本質的に循環リストの回収ができないという性質があるので，一括型のごみ集めとの併用が前提となる[1)]．理由は，**図-5** から明らかであろう．

参照カウンタ法のもう一つの問題は，カウンタの大きさである．理論的には，すべてのセルから同時に参照される可能性もあるから，リンクフィールドと同じ大きさにする必要がある．しかし，経験的には大部分のセルの参照回数は 1 であることが知られている[10)]．この性質を用いたのが，Deutsch & Bobrow の「多重参照表」を用いる方法[11)]である．この方法では，参照カウンタが 2 以上のセルを登録する多重参照表 (MRT) を用いる．詳細は省略する．

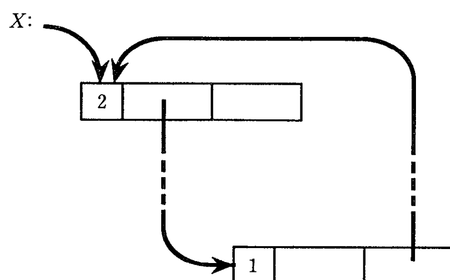


図-5 循環リストと参照カウンタ

5. 移動（複写）法

ごみ集めの目的は，本来限られたセル空間を有効に使うことである．移動（複写）法のようにセル空間を二つの部分空間に分けて使うというのは奇妙に思われるかもしれない．しかし近年のように，大きな仮想記憶空間上にセル空間がとられる場合[15)]は魅力的な方法の一つである．

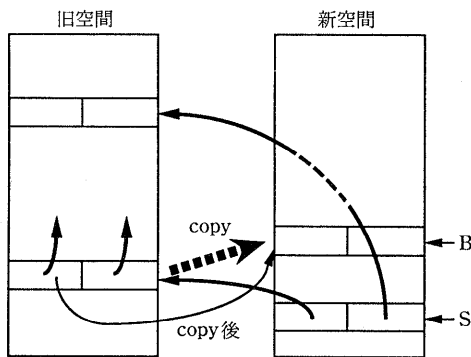
もともと移動（複写）法は，セル空間の生きているセルを 2 次記憶へ書き出し，再び読み込むという方法から生まれた[7)]．移動法の最大の特徴は移動により，セルの仕分けと詰め替えが同時に終了することである．

ここでは，Minsky-Fenichel-Yochelson-Cheney-Arnborg によるアルゴリズム[12)]を示す．新（移動先）空間を指す二つのポインタ S（走査用）と B（空き領域の底）とを設ける．**図-6**(a) にその原理を示す．

まず，ROOT から直接リンクのあるセルを新空間へ移す．手順は次のように行う．B の指す新空間のセルへ，ROOT の指す旧空間のセルの「内容」をそのまま複写する．旧空間のセルの link 1 には新空間の移動先アドレスを記入しておく．この段階では，新空間のセルの「内容」は旧空間を指している．

次に，S を B まで順次動かし，新空間の走査をしてリンクの修正をする．修正は次のように行う．リンク p が旧空間を指していれば，旧空間の該当セルの link 1 を調べる．これを q としよう．もし，q が新空間を指していれば，q は p の移動先であるから，リンク p を q に修正する．

q が旧空間を指しているときは，p が指しているセルは，まだ移動していないセルであるから移動手続きを行う．具体的なアルゴリズムを**図-6**(b) に示す．



（a） 移動法の原理

```
procedure moving_collector;
begin
  S:=1; B:=1; {0 means null pointer}
  for i:=1 to N do
    ROOT[i]:=move (ROOT[i]);
  while S<B do
    begin
      SPACE[S].link 1:=move (SPACE[S].link 1);
      SPACE[S].link 2:=move (SPACE[S].link 2);
      S:=S+1
    end
end;
procedure move(p);
if newspace(p)
  then return p
  else
    begin
      if oldspace (SPACE[p].link 1)
        then SPACE[p].link 1:=copy(p);
      return SPACE[p].link 1
    end;
procedure copy(p);
begin
  SAPCE[B].link 1:=SPACE[p].link 1;
  SAPCE[B].link 2:=SPACE[p].link 2;
  q:=B;
  B:=B+1;
return q
end;
```

（b） 移動法のアルゴリズム

図-6

6. 詰め替えのアルゴリズム

セルの大きさが一定の場合は，セル空間の断片化が生じないので，詰め替えは行わなくてよい．しかし，一定長セルでも仮想空間での局所性が問題となる場合や，セルの大きさが多種類の場合は詰め替えのアルゴリズムが重要となる．

詰め替えを行うときの条件として，**3.** で述べたように，次のような分類ができる．

（1） 任意順：詰め替え後の並び順になんらの制限をしない．

（2） 直線化：直接リンクのあるセルが連続した位置を占めるように詰め替える．

（3） 横すべり：セルの並び順序を保持したまま，セル空間の一方の端に集める．

直接リンクのある一連のリストが連続した位置を占めるように配置できれば，リストを単位として2次記憶との書出し・読込みが容易となるという意味で（2）が重要である．しかし，近年の仮想記憶環境では，ワーキングセットを小さくしたいという要請がより重要であり，その観点からは（3）が重要である．

ここでは，効率のよいアルゴリズムの研究が多数なされている（3）のタイプのアルゴリズムを取り上げる．

［移動先アドレスの計画による横すべり］

図-7 に横すべり型の詰め替えの概念を示す．すでに仕分けフェーズの処理が終了しているものとする．

$I_i(0 \leqq i \leqq n)$ は「生きているセル」のかたまりを，H_i $(1 \leqq i \leqq n)$ は「生きていないセル」のかたまりを表す．I_i を「島」，H_i を「穴」と呼び，H_i を代表するアドレス（たとえば穴の左端のアドレス）を h_i と表す．

横すべりの結果，島は左側に集められるものとする．各島をどれだけ横すべりさせるかは，穴の大きさの和として計算できる．これを各島の「移動量」とよび，s_i で表すことができる．

もし，あるセルのリンク p が島 I_i の中を指しているとすると，I_i の移動量は s_i であるから，移動後の新しいリンク p' は，$p-s_i$ とすればよい．したがって，移動の手順は次のようになる．

（1） 計画：あらかじめ各島の移動量を求める．

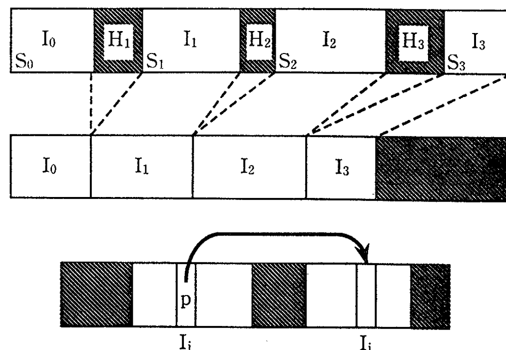


図-7 横すべり型の詰め替え

（2）　修正：計画に従ってリンクを修正する．

（3）　移動：実際にセルを移動させる．

これが横すべり型詰め替えの原理である[8]．

問題はあるリンクの値 p から，その移動量 s_i を知る方法である．p がある島 I_i の中を指すことは，$h_i \leqq p < h_{i+1}$ なる関係から見いだせる．最も簡単な実現法は，「穴の表」を作り，それを走査する方法である．表の内容は，穴の代表アドレス h_i と穴の大きさである．この表のために必要となるメモリが心配となるが，表を穴に埋め込めば余分のメモリを必要としない．

穴の表の走査を繰り返すのは能率が悪いので，表の代わりに h_i により「二分木」を作り，二分木検索法により能率を上げる方法や，ハードウェアによる方法が提案されている[16]．

［リンク反転による横すべり］

横すべり型詰め替えアルゴリズムとしては，前述の計画型とは異なる原理による興味深いアルゴリズム[9]がある．特徴はリンクの反転を行うこと，セル領域の走査が2回で済むことである．

原理は次のようなものである．図-7 によって，説明する．セル a からセル b を指しているとき，セル a は自由に動かせるが，セル b を動かしたいときは，セル a のリンクを修正しなければならない（図-8(1)）．

この原理に従うと，セル a とセル b のリンクを反転して（すなわち，セル b にセル a のアドレスを書き，セル b のもとの内容 X は，セル a に保存して）おけば，セル b は自由に動かせることになる．ここで，移動先のアドレス n が分かれば，

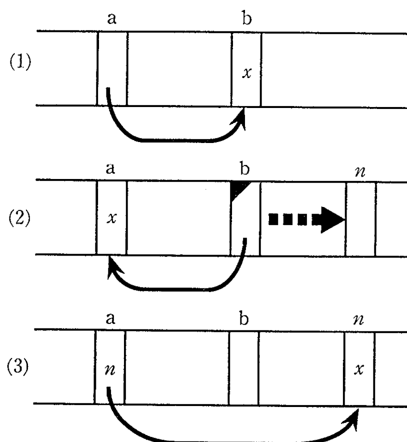


図-8　リンク反転による詰め替え

そこにセル b を動かす（図-8(2)）．

最後に，セル b にリンク（はじめに反転してあった）をたどって，セル a に移動先アドレス n を書き，セル a に保存されていたセル b のもとの内容 x を b の移動先 n に移す（図-8(3)）．以上でセルから指されていたセル b を n に移す作業が完了する．なお，リンクが反転しいることを示すのには，目印（タグ）を用いる．

1回目の走査は，アドレスの上昇方向に行い，正方向のリンクについて，リンクの反転を行う．

2回目の走査はアドレスの降下方向に行い，逆方向リンクの反転と，移動とを行う．移動先アドレスの計算は，走査を行うときに目印（タグ）の付いているセルを数えることにより求めることができる．

7. まとめ

ごみ集めのアルゴリズムを原理を中心に紹介した．

通常のメモリ管理法では，メモリの確保と返却が明確に意識されており，メモリセルの状態は，「未使用」か「使用中」かの二つの状態しかない．これに対して，ごみ集めを前提としたメモリ管理法では，メモリの確保は意識されるとしても，きちんと返却はせず，無意識に捨て置く（棄却する）だけである．したがって，メモリセルの状態は，「未使用」，「使用中」，「使用済み」の三つの状態が混在した状態にある．

ガーベジコレクタの仕事は，これらセルの状態を区別し，仕分けすることによって，「使用済みセル」を回収し「未使用セル」に再生することである．

このセルの状態の仕分けのための目印付け方法として，三つのアルゴリズムを紹介した．スタックを用いるリストたどり法，余分な作業域を必要としないリンク（ポインタ）反転法，最も素朴な走査法である．これらは，いずれも一括型の回収法の適したものである．

一方，逐次型あるいは即時型に適した，アルゴリズムとして，目印付けとは別の，参照カウンタを用いる方法を示した．しかし，参照カウンタ法は，循環リストが回収できないという本質的な問題があり，一括型との併用が前提となることを指摘した．

また，仮想記憶環境でのごみ集めに適した移動(複写) 型のアルゴリズムを取り上げた．このアルゴリズムは，最後に述べた「詰め替え」を同時替に行うことができ，仮想記憶のワーキングセットを小さくするのに役立つ．

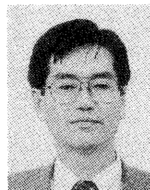
最後に，セル空間の断片化を解消するため，あるいは仮想空間での局所性を高めるために必要となる，詰め替えアルゴリズムの原理を述べた．

本稿は，ごみ集めの基礎の範囲に限ったので，個々のアルゴリズムの改良などについてはふれていない．くわしくは，文献を参照してほしい．

参考文献

1) Cohen, J.: Garbage Collection of Linked Data Structure, Computing Surveys, Vol. 13, No. 3, pp. 341-367 (1981).
2) Knuth, D. E.: The Art of Computer Programming Second Edition, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass. (1973).
3) Kurokawa, T.: A New Fast and Safe Marking Algorithm, Software Practice and Experience, No. 11, pp. 671-682 (1981).
4) Schirr, H. and Waite, W.: An Efficient Machine Indepependent Procedure for Garbage Collection in Various List Structure, Comm. ACM, Vol. 10, No. 8, pp. 501-506 (1967).
5) Dijkstra, E.D. et al.: Structured Programming, Academic Press, London (1972).
6) Collins, G. E.: A Method for Overlapping and Erasure of List, Comm. ACM, Vol. 3, No. 12, pp, 665-667 (1960).
7) Minsky, M. L.: A Lisp Garbage Collector Algorithm Using Serial Secondary Memory Storage, Memo 58 (rev.), Project MAC, MIT, Cambridge, Mass. (1963).
8) Wegbreit, B.: A Generalized Compactifying Garbage Collector, Computer J. Vol. 15, No. 3, pp. 204-208 (1972).
9) Moriis, F. L.: A Time and Space Efficient Garbage Collection Algorithm, Comm. ACM, Vol. 21, No. 8, pp. 662-665 (1978).
10) Clark, D. M. and Green, C. C.: An Empirical Study of List Structure in Lisp, Comm. ACM, Vol. 21, No. 2, pp. 78-86 (1977).
11) Deutsch, L. P. and Bobrow, D. G.: An Efficient, Incremental, Automatic Garbage Collection, Comm. ACM, Vol. 19, No. 9, pp. 522-526 (1976).
12) Baker, H. G.: List Processing in Real Time on a Serial Computer, Comm. ACM, Vol. 21, No. 4, pp. 280-294 (1978).
13) Lieberman, H. and Hewitt, C.: A Real-Time Garbage Collector That can Recover Temporary Storage Quickly, MIT CS Lab. Memo, Cambridge, Mass. (1980).
14) Dijkstra, E. W. et al.: On-the-Fly Garbage Collection: An Exersize in Cooperation, in Lecture Notes In Computer Science, Vol. 40, pp. 43-56, Springer-Verlag, New York (1976).
15) White, J.: Address Memory Management for Gigantic Lisp Environment or, GC Considered Harmful, Conf. Record 1980 LISP Conference, pp. 119-127, Stanford Univ. Stanford CA, (1980).
16) Terashima, M. and Goto, E.: Genetic Order and Compactifying Garbage Collectors, Inf. Processing Letter, Vol. 7, No. 1, pp. 27-32 (1978).
17) 日比野：ガーベジコレクションとそのハードウェア，情報処理，Vol. 23, No. 8, pp. 730-741 (Aug. 1982).

(平成6年7月21日受付)



日比野 靖（正会員）

1945年生．1970年東京工業大学工学部電子物理工学科卒業．1972年同大学院修士課程修了．同年日本電信電話公社（現，NTT）入社．1986年日本電信電話(株)基礎研究所情報通信基礎研究部第二研究室長．1987年同社ヒューマンインタフェース研究所言語メディア研究部知能分散処理研究グループ・グループリーダ．1992年北陸先端科学技術大学院大学情報科学研究科教授．1988年より東京工業大学工学部非常勤講師．専門はコンピュータ・アーキテクチャ．NTT研究所では LISP マシン ELIS の研究開発を行う．遍在コンピューティング (ubiquitous computing), VLSI アーキテクチャに興味をもつ．電子情報通信学会会員．