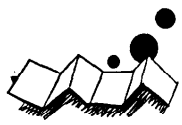
解　説

ガーベジコレクションとそのハードウェア†



日　比　野　靖††

はじめに

われわれが計算をする時，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，始めの方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時は，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．多くの高級言語処理系，特に Lisp に代表される記号処理言語の処理系では，“ガーベジコレクション（屑集め）”という方法が使われている．

この解説では，ガーベジコレクションのアルゴリズムを原理的な立場から整理し，動作環境やハードウェア化の観点から考察をくわえる．

ガーベジコレクションの多くのアルゴリズムについては，Cohen によるよい 概説[1] があるので，あわせて参照していただきたい．

1. 準　備

1.1 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（またはノード）”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**は異なるが，ここでは屑集め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．以下の記述***では，$link_i = 0$ のとき，空リンクであるとする．また，簡単のため，しばらくの間セルの大きさは一定とする．

```
D 1. type cell=recored link1 : 0..M, ****
                       link2 : 0..M,
                        :
                       linkn : 0..M end ;
     var CELLSPACE : array [1..M] of cell ;
```

リスト構造は，上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，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くと，もとのセルにもどってくる場合（循環リスト）を含む．

1.2 ルートと到達可能

リストの出発点となる特定のセルをルートという．

屑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 (activecell) 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．

† Garbage Collection and its Hardware by Yasushi HIBINO (Musashino Electrical Communication Laboratory N. T. T.).

†† 日本電信電話公社武蔵野電気通信研究所

* 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領域のアドレスで区別したり，ポインタとデータの区別をするタグを用いる．

*** Pascal 風の記述をする．

**** セル空間の大きさを表わす整数値．

D 2.

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

2. 基本アルゴリズム

2.1 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．ある時点でセル空間をみると，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，この解説では自由セルの管理方法にはふれない．)

計算が進行すると自由セルがなくなる．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．

屑集めの方法は，基本的には次の2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．

セルがなくなったとき，この2つのフェーズを順次実行するのが，古典的な一括形の屑集めである．

仕分けフェーズのアルゴリズムは，次の3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．参照カウンタが零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2つの部分空間に分けて，一方の部分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へ複写する．

回収フェーズの仕事は，自由セルの管理方法，セルの大きさの種類等により，種々の水準がある．また仕分けフェーズのアルゴリズムにより内容が異なる．

(2.1) 詰換え (compactification) を行わず，自由リストにもどす．

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) 詰換えを行う．詰換えの程度により，次の水準に分けられる．

(2.2.1) 任意順：詰換え後の並び順に何ら制約がない．

(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詰め換える．

(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 はセルがなくなった時起動するとして話を進めてきた．しかし，これは古典的な一括形の屑集めの概念である．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，一括形の屑集めでは，次のような問題を生じる．

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

このような問題に対する1つの方法は，リスト処理の中に屑集め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，屑集めをリスト処理と並行して行おうとするものである．

ここでは前者を即時形，後者を並行形と呼ぶことにする．

表-1 は，前述のセルの仕分け法と，これらの処理形態とにより，屑集めのアルゴリズムを整理したものである．

2.2 目印付け

目印を付けるために，セルに対応して1ビットの情報を付加する必要がある．この目印ビットをセル自身

表-1 ガーベジコレクションのアルゴリズムの分類

仕分け ＼	一括形	即時形	並列形
目印付け	リストたどり[2),3)] リンク反転[4)] 走査[5)]	藤田・西田[12)] 服部他[13)] Deutsch & Bobrow[11)]	Steele[18)] Dijkstra[19)] Kung & Song[21)] 日比野[23)] 薄他[25)]
参照カウント	Collins[6)]		
移動法	Minsky[7)]	Baker[14)] Lieberman & Hewitt[15)]	

* ルートの数を表わす整数値．

** 訳語としては“空きセル”の方が適切と思うが習慣に従う．

```
procedure mark (x) {recursive trace};
begin
    if x≠0 then
        if space [x]. mark=false then
            bigin
                space [x]. mark:=true;
                mark (space [x]. link1);
                mark (space [x]. link2)
            end
end
```

図-1 再帰的リストたどりによる目印付け

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を用いる方法とがある．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．

目印付けの方法は，次の3つに分けられる．

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2つの場合の手続きを示す．これはよく知られた，前順序 (pre-order) の木たどり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．セルの回収再成が要求された時は，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へらす様々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．

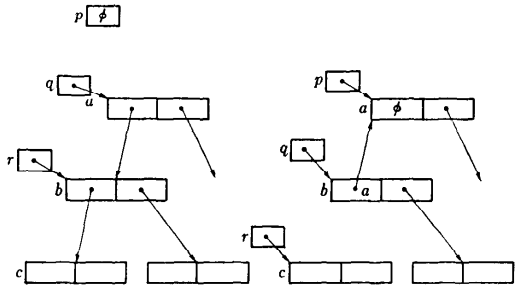
[M2. リンク（ポインタ）反転法][4]

図-2(a) に示すように，セル a からリンク $a.\mathrm{link}_1$ $(=b)$ をたどり，セル b を訪れるときセル b のリンク $b.\mathrm{link}_1$ にアドレス a を保持する．$b.\mathrm{link}_1$ の内容 c は，作業域 r に保持する．

もどる時は，逆向きリンクが link_1, link_2 のいずれに保存されているか知る必要がある．このため，セルには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．

[M3. 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**(図-3)**．



(a) リンク反転の原理

```
procedure mark (x) {link reversal};
begin
    if x=0 then finish;
    p:=0; q:=x;
11: space [q]. mark:=true;
    r: =spce [q]. link1;
    if r≠0 then
        begin
            space [q]. link1:=p;
            p:=q; q:=r; go to 11
        end;
    r:=space [q]. link2;
12: if r≠0 then
        begin
            space [q]. tag:=true;
            space [q]. link2:=p;
            p:=q; q:=r; go to 11
        end;
13: if p=0 then finish;
    if space [q]. tag=false
      then
        begin
            r:=space [p]. link1;
            space[p]. link1:=q;
            q:=p; p:=r; go to 12
        end
        else
        begin
            r:=space [p]. link2;
            space [p]. tag:=false;
            space [p]. link2: =q;
            q:=p; p:=r; go to 13
        end;
finish:
end
```

(b) リンク反転による目印付け

図-2

走査の方向とリンクの方向が逆のときは，1つ先のセルにしか目印が伝播しないから，走査の方向を反転させるなどの工夫が必要である．

この方法はセル空間全域へのアクセスを繰り返すので能率のよいものとはいえないが，Dijkstra ら[19] の並列アルゴリズムの中で採用されている．

なお，リンク反転法と走査法は実用性に疑問があり，実際には用いられていない．また，大きな仮想空間上

```
procedure mark {scanning};
begin
    for i:=1 to N do
        space [ROOT[i]]. mark:=true;
    repeat
        flag:=false;
        for i:=1 to M do
            if space [i]. mark=true then
                begin
                    if space [i]. link1≠0 then
                        begin
                            space [space [i]. link1]. mark:=true;
                            flag:=true
                        end;
                    if space [i]. link2≠0 then
                        begin
                            space [space [i]. link2]. mark:=true;
                            flag:=true
                        end
                end
    until flag=false
end
```

図-3　走査法による目印付け

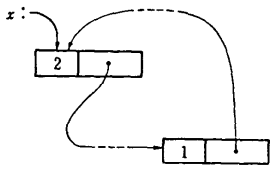


図-4　循環リストと参照カウンタ

のシステムでは，「リストたどり」にスタックを必要することが，さしたる問題ではなくなってきている．

2.3　参照カウンタを用いる方法[6]

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への代入などの度に，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わかるので，屑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形の屑集めに向いている．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，一括形の屑集めとの併用が前提となる[11]（**図-4** 参照）．

2.4　移　動　法

屑集めの目的は，本来限られたセル空間を有効に使うことである．移転法のようにセル空間を2つの部分空間に分けて使うというのは奇妙に思えるかもしれないが，近年のように大きな仮想空間上にセル空間がとられる場合は魅力的な方法の1つである．

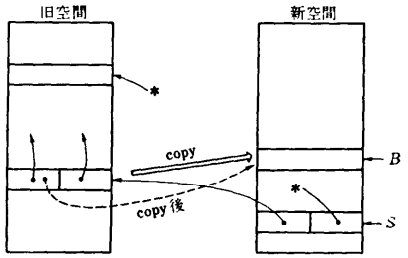
もともと移動法はセル空間の生きているリストを2次記憶に書き出し，再び読み込むという方法から生れた[7]．移動法の最大の特徴は移動により，セルの仕分けと詰め換えが同時に終了することである．

ここでは**3**章で述べる即時形アルゴリズムに使われている方法を示す[14]．

新(移動先)空間を指す2つのポインタ S（走査用）と B（自由領域の底）とを設ける（**図-5**(a) 参照）．

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 link1 に新空間の移動先アドレスを記入しておく．



(a)　移動法の原理

```
procedure moving-collector;
begin
    for i:=1 to N do
        ROOT [i]:=move (ROOT[i]);
    while S<B do
        begin
            space [S]. link1:=move (space [S]. link1);
            space [S]. link2:=move (space [S]. link2);
            S:=S+1
        end
end;
procedure move (p);
if newspace (p)
    then return p
    else
        if oldspace (space [p]. link1) then
            begin
                space [p]. link1:=copy (p);
                return space [p]. link1
            end;
procedure copy (p);
begin
    space [B]. link1:=space [p]. link1;
    space [B]. link2:=space [p]. link2;
    q:=B; B:=B+1; return q
end
```

(b)　移動法のアルゴリズム

図-5

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．つぎに，S を B まで動かし，新空間を走査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時は，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．

アルゴリズムを 図-5(b) に示す．

2.5 詰め換えのアルゴリズム

セルの大きさが一定の場合 (single-sized cell) は，セル空間の断片化が生じないので，詰換えは行わなくてもよい．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 (varisized cell) は，詰換えのアルゴリズムが重要となる．

ここでは，特に有用な横すべり形の詰換えアルゴリズムをとりあげる．

[移動先アドレスの計画による横すべり]

図-6 に横すべり形の詰換えの概念を示す．すでに仕分けフェーズの処理が終っているものとする．$I_i(0\leq i\leq n)$ は生きているセルのかたまりを，$H_j(1\leq j\leq n)$ は生きていないセルのかたまりをあらわす．

I_i を島，H_i を穴と呼び，H_i を代表するアドレス (たとえば左端のアドレス) を h_i と表わす．

横すべりの結果，島は左側に集められるとする．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呼び s_i で表わすことにする．

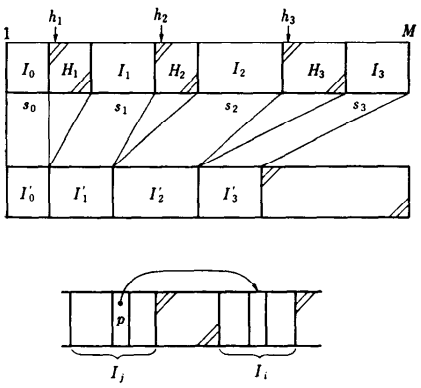


図-6　横すべり形の詰換え

もし，あるセルのリンク p が島 I_i の中をさしているとすると，I_i の移動量は s_i であるから，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ておく．

(2)　修正：計画に従って各セルのリンクを修正する．

(3)　移動：実際にセルを移動させる．

これが横すべり形詰換えの原理である[8]．問題は p から s_i を求める方法である．p が島 I_i の中を指すことは $h_i<p<h_{i+1}$ なる関係から見い出せる．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と大きさである．

穴の表の走査を繰り返すのは能率が悪いので，表のかわりに h_i により二分木を作り，二分木検索により能率を上げる方法や，5章でとりあげるハードウェアによる方法[28]が提案されている．

[リンク反転による横すべり]

横すべり形詰換えアルゴリズムとしては，前述の計画形とは異なる原理による興味深いアルゴリズムがある[9]．特徴はリンクの反転を行うことと，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 参照)．

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．この原理に従うと，セル a とセル b のリンクを反転して (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 q に保存して) おけば，セル b は自由に動かせ

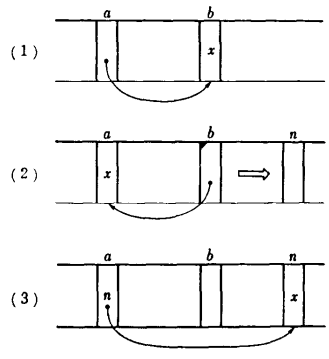


図-7　リンク反転による詰換え

* 表を穴におけば余分のメモリを必要としない．

ることになる．ここで移動先のアドレス n がわかれば，そこにセル b を動かす．b のリンクをたどって，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．以上でセル a 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，リンクの反転していることを示すのには，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下降方向に行い，逆方向リンクの反転と，移動を行う．移動先アドレスの計算は走査を行うときに目印の付いている語を数えることにより求められる．

3. 即時形のガーベジコレクション

3.1 多重参照表による方法

即時形のガーベジコレクションとしては，参照カウンタを用いるものが知られている．

参照カウンタ法には，循環リストが回収できないという問題があることを2.1で指摘した．参照カウンタ法のもう1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[11] の多重参照表を使う方法である．

この方法では，参照カウントが2以上のセルを登録する多重参照表(MRT)，参照カウントが零のものを登録する零参照表(ZCT)，変数からの参照を登録する変数参照表(VRT)という3つの表を用いている．各表への登録，検索はハッシングによる．ZCT，VRT はROOT に対する操作（すなわち，変数への代入や，中間結果を記憶するスタックのプッシュ，ポップ等）を行う時に，表へのアクセス回収を減らす目的で設けたもので，次のような使い方をしている．

ZCT には，真にどこからも参照されていないセルと，ROOT に束縛されているセルの両方が混在している．すなわち，新たに ROOT への束縛を行うときも，ZCT から該当セルを除去しない．VRT は ZCT の中にあるこの ROOT に束縛されているセルを区別するための表であるが，その更新は ROOT に対する操作の度には行わず，実際にセルの回収に入るときにまとめて行う．

回収できるセルは，ZCT にあって VRT にないセルである．Deutsch & Bobrow は ZCT と VRT とを走査する別のプロセッサを付加することを提案している．

3.2 共有リスト表による方法

複数のリストが部分リストを共有するようになる(すなわち多重参照セルを生じる）必要条件は，それらのリストに対して，リンクの書換え，ROOT への代入を行うことである．

藤田ら[12]はこの性質を用いた方法を提案している．この方法は，共有リスト表を用い，目印付けによるセルの仕分を行う．

以下の説明の前提として，共有リスト表には共有部リストをもつ可能性のあるリストのルートが対の形で登録されているものとする．

2つの ROOT (x と y) に束縛されているリストの間でリンクの書換えが行われる場合を考える．たとえば x の途中のセル q のリンク(リスト a を指す)が，y の途中のセル b を指すように変更されるとしよう(図-8 参照)．リスト a は屑になる可能性があるが，x と共有関係にある別のリスト z があれば，リスト a は無条件に回収できない．

そこで共有リスト表(T)のリスト対を調べ，リスト x と共有リストをもつ可能性のあるリストをすべて集める．このリストの集合を W とする．W は次のようにして作る．

はじめ x と対をなすリスト z_1 をさがす．次に z_1 と対をなすリスト z_2 をさがす．以下順次くり返して，W に加えていく．このようにして作った W は，x と共有リストをもつ可能性のある最小集合である．

表 T には新たにリスト対 (x, y) を登録する．

次に，W に登録されているリストに目印付けを行う．

リスト a からたどって目印の付いていないセルが

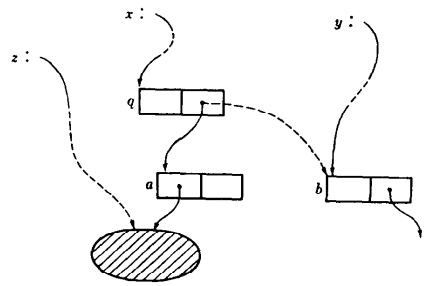


図-8 リンクの書換えと共有リスト

回収可能な屑セルである.

藤田らは,共有リスト表がふくらむのを防ぐ方法も示している.

3.3 部分空間に分けたガーベジコレクション

[ページ外参照カウンタを用いる方法]

大きな仮想空間の上にセル空間がとられる場合の問題を,ページ単位の屑集めにより解決しようという提案が,服部ら[13]によりなされている.

この方法は,参照カウンタ,多重参照表と目印付けを併用するもので,目印付けをページ内に限定すること,参照カウンタの更新をページ外からの参照に限定*していることである.

セルに対する参照は,他のページおよびページ内のセルからのものと,ROOT(スタック)からの参照がある.参照カウンタの更新オーバヘッドを少なくするために,ROOT からの参照とページ内参照は参照カウンタに反映させず,通常の目印付けにより屑セルの仕分けをする.各セルに参照カウンタフィールドを設けるが,この更新を行うのはページ外からの参照の時だけである.参照カウンタのあふれは多重参照表に登録する.

目印付けは ROOT と対象ページ内の参照カウントが非零のもの** を起点として対象ページ内でだけ行う.

目印付けが終了したら目印のないセルを回収するが,その時,そのセルが他のページを参照している場合には,相手先セルの参照カウンタを1減らす.しかし,相手先セルがページアウトされていた時には,ページイン処理を行わずに多重参照表の該当項目を1減ずる.表に該当項目がない時は,項目を設け −1 を登録する.

このアルゴリズムは次のような特徴をもつ.

(1) 実メモリ上のページを選んでセルを回収でき,しかもページ単位の処理であるから即時性が保証できる.

(2) 参照カウントの更新オーバヘッドが少ない.また,目印付けを行うので,ページ内の循環リストは回収できる.

(3) 屑集めに入ったページをリストプロセスが使わないことにすれば,リストプロセスと GC プロセスの並行処理ができる.その場合,細かいレベルでの同期操作が不要である.

[即時性を保証した移動法]

Baker[14] は移動法を用いた即時屑集めアルゴリズムを提案している.

基本アルゴリズムのところで述べたように,セル空間は2つの部分空間に分けられる.セルの生成操作(Lisp のcons)の中で現在使用中の部分空間に自由セルがないことがわかると,新しい部分空間へセルの移動を行う.この時,即時性を保証するために,移動させるセルの個数を k 個以下に制限する.

したがって,このアルゴリズムでは,新,旧両空間に生きているセルがあることになる.この問題に対して,リスト処理側からは常に新空間をアクセスするという方策をとる.

このため,リンクをたどる操作(car, cdr)時に,対象セルが旧空間にあるときはセルの移動をしなければならない.リンクの書換え操作(replace)には余分の処理は必要ない.なぜなら,リンクの書換えの対象となるセル,およびリンクの書換え先のセルは,ROOT の移動かリンクたどり操作に伴うセルの移動かによりすでに移動が終了しているからである.

[多数の部分空間を用いる移動法]

Baker のアルゴリズムは,セル空間を2つの部分空間に分けているだけなので,大きな仮想空間での効率が問題となる.Hewitt ら[15]は,Baker のアルゴリズムを一般化して多数の部分空間(region)を用いる方法を提案している.

この方法は新しい部分空間ほど屑の発生が多く,古い空間ほど屑の発生が少ないという性質に着目したもので,部分空間を世代番号で管理し,屑集めのコストを部分空間の古さに比例させようというものである.アルゴリズムを要約すると次のようになる.

(1) セルの生成は常に現空間で行う.現空間があふれれば,現空間を廃棄空間として,新空間に移り新しい世代番号を与える.

(2) 廃棄空間の生きているセルを別の新しい空間に移す.この空間には廃棄空間と同一の世代番号を与える.

(3) 廃棄する部分空間へのリンクを修正する.

リンクの修正のとき,一般にはすべての部分空間を走査する必要があるが,この方法では一世代前の部分空間までしか走査を行わなくてもよいことを保証して

---

* この他にアトムヘッダからの参照があるが,Lisp のインプリメントに依存するので,ここではふく.

** 参照カウンタが非零であっても,後述するように,多重参照表に −1 が登録されている項目もあるので,真に参照カウントが非零のものも求めるには,参照カウンタと多重参照表を照合しなければならない.しかし,照合を行わなくても安全である.

いる．これが保証されるためには，すべてのセルのリンクが次の性質を満たす必要がある．

P1. 注目している部分空間より古い世代へのリンク（後向きリンク）は，任意の世代を指してもよい．

P2. 注目している部分空間より新しい世代へのリンク（前向きリンク）は一世代新しい世代のみである．

これらの性質を満たすため，リンクの書換え操作のとき，次のような処置をする．

注目セルの空間と書換え先セルの空間の世代番号が2以上はなれている時は，注目セルを一世代ずつ新しい世代に移していく．このようにして前向きリンクが常に一世代以内に限られるようにする．

4. 並列形のガーベジコレクション

4.1 リストプロセスと GC プロセスの同期

一括形のガーベジコレクションの欠点を最も直接的な方法で解決しようというのが，並列形のガーベジコレクションである（表-2 参照）．

並列形の屑集めでは，リストプロセスと GC プロセスとが同時にリスト構造を変更するので，両プロセスの間の同期のとり方など困難な問題を多数含んでいる[16),17)]．

実際のところ，並列アルゴリズムを屑集めに適用して効率を上げるところまで至っていないのが現状である．

並列形の屑集めアルゴリズムの提案は，古くは Minsky のもの[2)]があるが，Steele[18)] のアルゴリズムは並列形の屑集めアルゴリズムを具体的に示した初めてのものである．このアルゴリズムでは，リストプロセスと GC プロセスの2つのプロセスの間で，臨界領域 (critical section) を用いて同期をとりながら，処理を進める．目印付けは再帰的リストたどりによっており，GC プロセスも独自のスタックを持つ．

表-2 並列ガーベジコレクションアルゴリズム

提案者(年)	目印付け	目印	特徴	実装
Steele (1975)	リストたどり	1ビット*	多種類のセマフォアを用いる	—
Dijkstra (1976)	走査	白，灰，黒	臨界領域を最小化	—
Kung & Song (1977)	リストたどり	白,灰白,灰,黒	双頭待ち列の使用	—
日比野 (1978)	走査とリストたどり	1ビット	走査要求フラッグの使用	○
薄 他 (1978)	リストたどり	1ビット	メモリ管理を分離	○

* この他に詰換えのためのタグがある．

GC プロセスは，目印付け，詰換え，回収の3つのフェーズを繰り返し実行している．

並列形の屑集めにおいて，目印付けにより ROOT から到達可能なセルを仕分けようとするとき，何が問題になるかといえば，GC プロセスが到達可能なセルに完全に目印を付けたことを，どうやって保証するかということである．なぜなら，GC プロセスがある順序でセルをたどり目印を付けている間にも，リストプロセスが動いているので，時々刻々 ROOT は変化し，またリスト構造も変わっているからである．

Steele のアルゴリズムでは，次のような方法でこの問題を解決している．

リストプロセスが，リスト構造の変更操作を行う時は，GC プロセスが目印付け，詰換え，回収のどのフェーズにあるかを臨界領域の中で判定し，目印付けフェーズなら，次のような処置をする．

（1） セルの生成時には，リストプロセス側でそのセルに目印を付けておく．

（2） リンクの書換え時には，書換えを行ったセルを GC プロセスのスタックへ積み上げる．

これらの処理は複数のセマフォアを用いた臨界領域の中で行われており複雑なものとなっている．

このアルゴリズムが正しく動作するか否かはわからないが，リストプロセスと GC プロセスが並列動作する場合の問題点と，その解決法を具体的に示した功績は大きい．

4.2 走査法による並列ガーベジコレクション

臨界領域の必要な操作を最小限にすること，また，アルゴリズムの正当性を保証することを目的とした，Dijkstra[19),20)] のアルゴリズムがある．このアルゴリズムでは効率を問題にしていない．

GC プロセスは目印付けフェーズと回収フェーズの2つのフェーズから成り，目印付けは走査法を用いている．

目印は，白，灰，黒の3色に変わる．変化は白から灰，灰から黒のように段階を追って進む．白からは灰色に変えるが，灰色や黒の場合は色を変えないという操作を影付け (shading) と呼ぶことにする．

GC プロセスは，まず ROOT が指しているセルに影付けする．次にセル空間全域の走査を繰り返し，灰色のセルを黒に，灰色のセルからリンクのあるセルを影付けする．目印付けの終了は灰色のセルがなくなった時である（自由リストも一般のリストと同様に目印付けする）．

リストプロセス側では，セルの生成時と，リンクの書換え時に，関連するセル* に影付けして GC プロセスに知らせる．灰色セルの存在する限り，GC プロセスは目印付けのための走査を繰り返すから，ROOT から到達可能なセルには必ず目印が付くことになる．

回収フェーズでは，白色のセルを回収する．

並列形の屑集めアルゴリズムが正当であるためには，任意の時刻で，ROOT から到達可能なセルの集合 A (すなわち生きているセル) と，目印の伝播可能なセルの集合 M との間に，常に $A \subseteq M$ が保証できねばならない．

Dijkstra のアルゴリズムでは，GC プロセスが目印付けを終了した時点でこの関係が成立しており，また，GC プロセスが回収フェーズにあってもリストプロセス側が影付けを行っているので，やはりこの関係が成立している．

このアルゴリズムは，3色の目印を使っているので，実現には目印として2ビット必要である．また影付け操作を非可分操作としなければならないので，そのためのハードウェアがないと効率が悪い．

4.3 目印付けの効率改善

Dijkstra のアルゴリズムは，灰色の目印の付いたセルの存在により，リストプロセスから GC プロセスに目印付けの再走査を要求している．そのため灰色セルがないことを確認するため，GC プロセスはセル空間を走査しなければならない．

この非効率を改善したものに，Kung[21] らの両頭待ち列 (deque) を用いるアルゴリズムと，筆者の提案した走査要求フラッグを用いるアルゴリズム[22]がある．

[Kung のアルゴリズム]

目印には，灰白，白，灰，黒の4色を用いる．灰白は自由リストを表わし，自由リストには目印付けを行わない．

両頭待ち列は，リストプロセス用と GC プロセス用の2つの口を持つ．GC プロセス用の口は，GC プロセスがリストたどりをする時にスタックとなる．

基本的なアイデアは，リストプロセスから GC プロセスへ目印付けをやってほしいリストがまだ残っていることを知らせるために，セルの生成時と，リンクの書換え時に，両頭待ち列にそのリストを挿入するというところにある．

GC プロセスは目印付けをスタックを用いて再帰的リストたどりにより行う．この方法によれば，Dijkstra の方法よりも目印付けが相当改善される．

[Hibino のアルゴリズム]

Dikstra と Kurg，いずれのアルゴリズムも，3色ないし4色の目印を用い，灰色セルの存在ということを利用しているのに対し，このアルゴリズムでは，1ビットの目印と，リストプロセスから GC プロセスへリスト構造の変更が生じたことを知らせる走査要求フラッグとを用いている．

GC プロセスは，目印付フェーズの最後で走査要求フラッグを調べ*，フラッグがオンなら再走査に入る．

目印付けは，セル空間の走査と再帰的リストたどりを併用する．すなわち，セル空間を走査中に目印の付いたセルを見つけると，そのセルをルートとして再帰的リストたどりを行う．したがって走査要求フラッグがオンでなければ，一回の走査で到達可能なセルに目印が付く．

このアルゴリズムは，小型のマルチプロセッサ上の Lisp に実装され，効率の測定が行われた[23),24)]．

筆者の知るかぎりでは，並列ガーベジコレクションのアルゴリズムで，実現されたものはこのアルゴリズムと，次に述べる薄ら[25)]のもののみである．

4.4 メモリ管理プロセッサの分離

薄らは Lisp の処理をインタプリタとメモリ管理というレベルで2つのプロセッサに分割する方法を提案している．

メモリ管理プロセッサは，セルの生成，リンクの書換え，ROOT への代入などリスト構造の変更操作と，屑集めを担当する．これにより，屑集めのアルゴリズムが簡単になっている．

目印付けは通常の再帰的リストたどりである．リスト構造の変更操作のときは，GC のスタックへ関連するリストを積み上げることにより，目印付けを保証している．

アルゴリズムに不明の点もあるが，特別に設計されたプロセッサ2台により実動したとの報告がある．

5. ガーベジコレクションのハードウェア化

5.1 仮想記憶とガーベジコレクション

リスト処理全体の中で屑集めに要する時間は，セルの消費速度と関係している．しかし，セルの供給能力以上に屑が発生することはないから，1回の再生回収で十分なセルが回収できる状態では，処理時間全体の中で屑集めの占める割合は，ある程度におさえられ

* 生成時には，そのセル．書換え時には書換え先のセル．

* GC プロセス側では，この操作のみが非可分操作である．

る．普通この割合は10% 内外であり[26]，50% を越えることはまずないと考えてよい．したがって処理速度の向上という面からだけハードウェアの効果を期待しても，よい結果は得られない．

むしろ今日，ガーベジコレクションの問題は，十分大きなセル空間がありながら，一度屑集めに入ると，結果的には十分なセルが回収されるとしても，利用者にとって堪えがたいほどの待ち時間が生じるという点である[18),27)]．セル空間が実記憶にとられている場合は，10^6 セル程度までは，秒のオーダで応答が期待できるから，実時間の応用以外では十分である．しかし，セル空間が仮想記憶上にあって，実記憶との大きさの比が数倍以上あるときは，古典的な一括型の屑集めでは，二次記憶のアクセス時間で処理時間が決まってしまう．仮想セル空間の大きさが 10^7〜10^8 セルの場合には，その時間は数10分にもなる．

したがってハードウェア化が期待されるとすれば，上記のような問題を解決することであろう．

このためには，一言でいえば，ワーキングセットを小さくする方向でのハードウェア化が望ましい．そのような性質を持つものは，4章でとりあげた即時形アルゴリズムである．このうちハードウェア化が興味深いものは，次のものである．

（1） 多重参照表を用いる方法(Deutch & Bobrow)
（2） 共有リスト対表を用いる方法（藤田，西田）
（3） ページ外参照カウンタを用いる方法（服部他）
（4） 多数の部分空間を用いる移動法 (Hewitt)

(1),(2),(3)は表への登録，参照にハッシングを用いており，ハッシングハードウェアの導入によりアルゴリズムが一層実用的になろう．(3),(4)は仮想記憶機構との関係をより密にすることにより，一層の効率改善がはかれると考えられる．

また，これらはいずれも，並列処理化の可能性の高いものであり[11)~14)]今後の研究が期待される．

5.2 詰換えアルゴリズムのハードウェア化

ワーキングセットを小さくするという観点からは，詰換えアルゴリズムが重要である．詰換えは，セル空間の断片化を防ぐために必要であるが，セル空間全域の走査を伴うので，走査の回数を減らすこと，および，セルへのアクセスをできるだけ減らすことが望ましい．

後藤ら[28)]の提案している横すべり形詰換えのハードウェアはこの点で興味深い．**図-9** にその概念を示す．

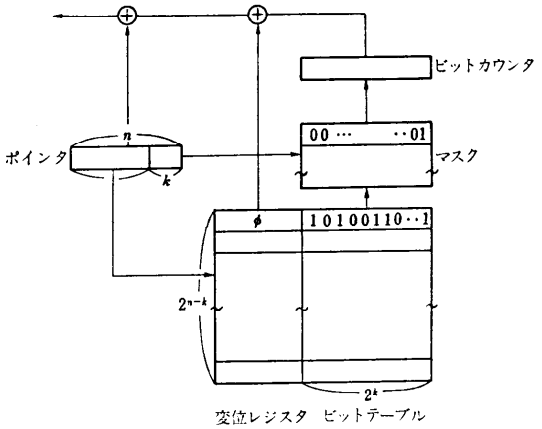


図-9 横すべり形詰換え用ハードウェア

横すべりの移動先アドレスを，目印ビットテーブルと変位レジスタから計算して求めている．

5.3 ビットテーブルの応用

後藤らのハードウェアは，ビットテーブルの巧みな応用である．ビットテーブルを用いるとセル自体へのアクセス回数を減らせるという点も，小さいことであるが見のがせない．ビットテーブルを効率よく使うにはセルアドレスからビット位置への変換ハードウェアがあるとよい．このようなハードウェアを実装した例として，瀧ら[29)]の Lisp マシンがある．さらにビット表の効果を上げるには，ビット表を高速の内部メモリに置くことである．メモリ量としては，セル当り1ビットとすれば，10^6 セル程度の場合，8バイト／セルとして，主記憶容量の 1.6% 程度となる．

5.4 並列ガーベジコレクションとハードウェア

最後に，並例ガーベジコレクションについて，ハードウェア化の面から考察する．

これまで提案，あるいは実装された並列アルゴリズムは，いずれもセルの仕分けに目印付けを用いたものであり，理論的には興味深いが，今日のガーベジコレクションの問題の解決になっていない．むしろ，前述したように，即時形アルゴリズムの中に並列化に適したものがある．

並列化を行う場合，ハードウェアの立場から注意すべき点は，メモリアクセス競合と，ガーベジコレクタとなるプロセッサの性能である．前者はリスト処理に限らず一般に多重プロセッサの問題であるが，リスト処理の場合，メモリアクセスが主体であるから特に注意が必要である．

後者は，プロセッサの性能が十分でないと，セルの

消費に対してセルの回収が間にあわず，“決してメモリを使い切ることがない (never run out of space)”条件[18]を保つことができなくなるという問題である．

屑集め自体の処理は単純なものであるから，プロセッサとしては小規模のハードウェアでよい[24]．

したがって，適切なアルゴリズムを用いれば，並列ガーベジコレクションを妥当なコストで実現できる可能性がある．

あとがき

ガーベジコレクションのアルゴリズムを原理に重点をおき紹介し，ハードウェア化について若干の考察をした．ガーベジコレクションのアルゴリズムについては多くの研究があり，その考え方も統一的に扱えないので，説明が散慢になったことをお許し願いたい．

この解説では，記号処理の立場からガーベジコレクションを扱ったが，ガーベジコレクションは，オペレーティング・システムや，オブジェクト指向形言語処理系などに広く適用されていくものと考えられ，プログラミングの概念の抽象化が進むに従ってますます重要性を増すであろう．

この解説でふれなかった範囲として，アルゴリズムの正当性の問題と[21),30),31)]，アルゴリズムの特性解析[1),2),16),18)]がある．並列処理に関しては，これらの研究が特に重要であると考えられる．

参考文献

1) Cohen, J.: Garbage collection of Linked data structure, Computing Surveys, 13 (3), pp. 341-367 (1981).
2) Knuth, D. E.: The Art of Computer Programming, Vol. I: Fundamental Algorithms, Addison-Wesley, Reading, Mass. (1973).
3) Kurokawa, T.: A new fast and safe marking algorithm, Software Practice and Experience, 11, pp. 671-682 (1981).
4) Schorr, H. and Waite, W.: An efficient machine-independent procedure for Garbage collection in Various list structure, Comm, ACM, 10 (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, 3 (12), pp. 655-667 (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. 15 (3) pp. 204-208 (1972).
9) Morris, F. L.: A time and Space efficient garbage collection algorithm, Comm. ACM, 21 (8), pp. 662-665 (1978).
10) Clark, D. M. and Green, C. C.: An empirical study of list structure in Lisp, Comm. ACM, 20 (2), pp. 78-86 (1977).
11) Deutsch, L. P. and Bobrow, D. G.: An efficient, incremental, automatic garbage collection, Comm. ACM, 19 (9), pp. 522-526 (1976).
12) 藤田米春，西田富士夫：改良形即時処理ガーベジコレクション，信学論 J64-D (3) pp. 222-227 (1981).
13) 服部 彰他：高速リスト処理に適したアーキテクチャについて，情処記号処理研資 18-9, pp. 95-101, (1982).
14) Baker, H. G.: List processing in real time on a serial computer, Comm. ACM, 21 (4), pp. 280-294 (1978).
15) Lieberman, H. and Hewitt, C.: A real-time garbage collector that can recover temporary storage quickly, MIT C. S. Lab. Memo Cambridge, Mass. (1980).
16) Wadler, P. L.: Analysis of an algorithm for real time garbage collection, Comm. ACM, 19 (9), pp. 491-500 (1976).
17) Gries, D.: On believing programs to be correct, Comm. ACM, 20 (1), pp. 49-50 (1977).
18) Steele, G. L.: Multiprocessing compactifying garbage collection, Comm. ACM, 18 (9), pp. 495-508 (1975).
19) Dijkstra, E. W. et al.: On-the-fly garbage collection: An exercise in cooperation, in Lecture Note in computer Science, 46, pp. 43-56, Springer-Verlag, New York (1976).
20) 和田英一：ソフトウェアからみたマルチプロセッシングシステム，信学技報，EC 76-71, pp. 1～4 (1976).
21) Kung, H. T. and Song, S. W.: An efficient garbage collection system and its correctness proof, Proc. 18th Annual Symposium on Foundation of Computer Science, pp. 120-131 (1977).
22) 日比野靖：並列ガーベジコレクションのアルゴリズムと Lisp への適用，信学技報，EC 78-32, pp. 21-30 (1978).
23) Hibino, Y.: A parallel garbage collection algorithm and its application to Lisp, Trans. IECE Japan, E63 (1), pp. 1-8 (1980).
24) Hibino, Y.: A practical parallel garbage collection algorithm and its implematation, Conf., Proc. 7th Annual Symposium on Computer

Architecture, pp. 113-120 (1980).
25) 薄 隆他：マルチマイクロプロセッサによるLispマシン，信学技報，EC 78-33, pp. 31-38 (1978).
26) 竹内郁雄：第2回 Lisp コンテスト，情報処理，20 (3), pp. 192-199 (1979).
27) White, J.: Address Memory Management for a Gigantic Lisp environment or, GC considered harmful, Conf. Record 1980 Lisp Conference, pp. 119-127, Stanford Univ. Stanford California (1980).
28) Terashima, M. and Goto, E.: Genetic order and compactifying garbage collectors, Inf. Process. Lett. 7 (1), pp. 27-32 (1978).
29) 瀧 和男他：Lispマシンの試作—アーキテクチャとLisp言語の仕様—，情報処理論文誌，20(6) pp. 481-486 (1979).
30) Gries, D.: An Exercise in proving parallel program correct, Comm. ACM. 20 (12), pp. 921-930 (1977).
31) Francez, N.: An application of a method for analysis of cyclic programs, IEEE Trans. Softw. Eng. 4 (5), pp. 371-377 (1978).

(昭和57年5月6日受付)