# 屑集め機能を備えた Pascal 処理系の実現†

宮本 衛市††

Pascal では，標準手続き *new* を呼び出して動的変数を生成し，不要になった動的変数は標準手続き *dispose* で処理系に返却する．しかし，この *dispose* の呼出しは，返却するのがパラメータで指定した動的変数のみであり，しかも可変部つきレコード変数のときには，生成したときと同じタグ値を明示しなければならない．リンクでつながれた動的なデータ構造を返却するためには，そのための手順を書く必要があり，場合によってはそのためのリンクも必要となり，アルゴリズムばかりでなく，データ構造さえも不透明にしかねない．そこで，プログラマは動的変数の生成にのみ関知すればよいように，屑集め機能を備えた Pascal 処理系を実現した．屑集めは LISP 系の処理系では必須の機能であるが，これを Pascal 処理系で実現するためには，任意のデータ構造を有する動的変数のつながりをヒープ領域中で追跡する必要がある．そのため，ポインタ型とそれを取り巻く構造型を型記述子を用いて記述しておき，変数と型記述子を対にして，つながりの根をなす静的変数から出発して動的変数を追跡し，参照可能か否かの印付けを行い，屑集めを行う．屑集め機能を備えた処理系のもとでコンパイラ自身を実行させ，実行時間および記憶容量のオーバヘッドを実測した結果，それらはわずかであり，しかも処理系への追加プログラム量もわずかであり，プログラマの負担軽減とあわせ，屑集め機能の有効性を確認した．

## 1. はじめに

Pascal では，標準手続き *new* を呼び出して動的変数を生成し，不要になった動的変数は標準手続き *dispose* で処理系に返却する．動的変数は任意のデータ型をもつことが許されるが，通常の使い方としては，ポインタ型のフィールドをもつレコード型の変数を生成し，リスト構造，木構造あるいはネットワーク構造などのリンクでつながれたデータ構造を構築するために使用する．このとき，可変部を有するレコード型の場合には，特定の場合を指定して動的変数に割り当てる領域を節約することもできる．

本論文で問題とするのは，不要になった動的変数の領域の再生処理である．*dispose* を呼び出して動的変数を処理系に返却しても，返却されるのはパラメータのポインタ変数が指す動的変数だけである．さらに，その動的変数が可変部を指示して生成したレコード変数であれば，同一の場合を指示して返却しなければならない．しかも，その指示は定数でなければならないため，タグフィールドの値で分類する *case* 文をおき，*dispose* の呼出しを列挙しておくなどの対策が必要になる．一方，不要になったのがリンクでつながれたデータ構造であれば，それを構成する要素を一つずつ返却するための手順を書かなければならず，場合によってはそのためのリンクを用意しておくなど，アルゴリズムばかりでなく，データ構造さえも複雑にしてしまう．

Pascal 処理系のなかには，標準手続きとして *release* をもつ処理系[1]や，Pascal-P コードに基づく処理系では *mark* と *release* によるヒープ領域の再利用が図られている[2]．これらは指定した動的変数の生成以降に生成したすべての動的変数を一括して返却してしまうものであり，ヒープ領域が一種のスタックとして使われることを前提としている．しかし，ヒープ領域は必ずしもスタックとして使われるとは限らず，*release* の使用は一般に危険を伴う．一方，ヒープ領域が十分にあれば，強いて不要になった動的変数を返却する必要はないわけで，ファイルで支援した仮想的な大容量のヒープ領域を用意しておく処理系も作られている[3]．これは動的変数による恒久的なデータ構造の構築には有益であろうが，動的変数の生成消滅を繰り返すような問題に対しては，処理時間のオーバヘッドが大きすぎるきらいがある．

一方，LISP 処理系に対しては種々の屑集めの方法が提案されており[4),5]，Pascal 処理系に対する適用も提案されているが，非現実的として構想だけに終わっている[6]．そこで，われわれは実際に屑集めの機能を備えた Pascal 処理系を実現し，屑集めに要する処理時間および所要メモリのオーバヘッドを，コンパイラ自身を屑集めを備えた処理系のもとで稼動させて実測

† Implementation of a Pascal Processor Equipped with Garbage Collection Facilities by EIICHI MIYAMOTO (Division of Information Engineering, Graduate School of Engineering, Hokkaido University).

†† 北海道大学大学院工学研究科情報工学専攻

してみた．翻訳させたのは再びコンパイラ自身であるが，少なくともコンパイラが要求するようなデータ領域の大きさのもとでは，層集めに要するオーバヘッドはわずかであり，*dispose* などを使って不要になった動的変数をプログラム上で返却する危険や煩わしさにくらべれば，Pascal 処理系が層集め機能を備えることはきわめて有効であると考える．さらに，層集め機能を処理系に備えるために追加するプログラム量もわずかであり，しかも標準手続き *dispose* あるいは *release* と共存することにも何ら問題はない．

## 2. 層集め機構

Pascal プログラムを実行させるために，プログラム領域とは別にデータ領域を設定し，**図 1** に示すように，一方からは手続き・関数の呼出しに伴う仮パラメータや局所的変数を割り付けるスタック領域として，他方からは動的変数を割り付けるヒープ領域として使う．この両領域のあいだにあって，まだいずれにも割り付けられていない領域を空き領域と呼ぶ．今回実現した処理系のヒープ領域の管理は，次の 2 通りである．

a）自由リスト方式

b）詰替え方式

前者は不要になった動的変域の領域を再利用するために，ヒープ領域中の再利用可能な領域をつないで自由リストとして保持する．動的変数の生成要求があると，処理系はまず自由リストの中から割り当てる領域を探すが，該当する領域が見つからないときにはヒープ領域に接する空き領域から割り当てる．一方，後者はヒープ領域の上端を指すポインタを置くだけで，動的変数の生成要求があると，処理系はヒープ領域に接する空き領域から領域を割り当てる．したがって，ヒープ領域は増大の一途をたどり，層集めの時点で参照可能な領域はヒープ領域の下端に詰め替え，ヒープ領域を縮小する．

いずれの方式にしても，ヒープ領域中にある動的変数が層集めの時点で参照可能であるか否かを判別する必要がある．この操作がいわゆる印付けである．

### 2.1 Pascal における印付け

Pascal では，動的変数は任意のデータ型をもつことができる．その動的変数を参照するポインタ変数は単独の変数としてのみでなく，レコード変数のフィールドとして，あるいは配列変数の要素として組み込まれ，さらにそれらが入れ子になった複雑な構造もありうる．層集めの時点では，主プログラムを含め，すべての呼び出されている手続き・関数で宣言されている変数のうち，動的変数を参照可能なすべての変数とそのデータ型を掌握している必要がある．

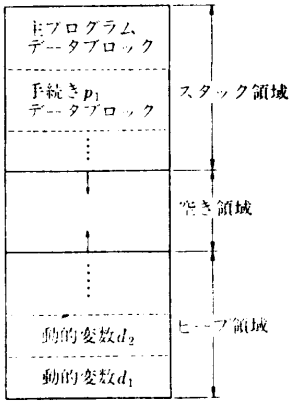


**図 1**　Pascal におけるデータ領域の割付け
Fig. 1　Allocation of data area in Pascal.

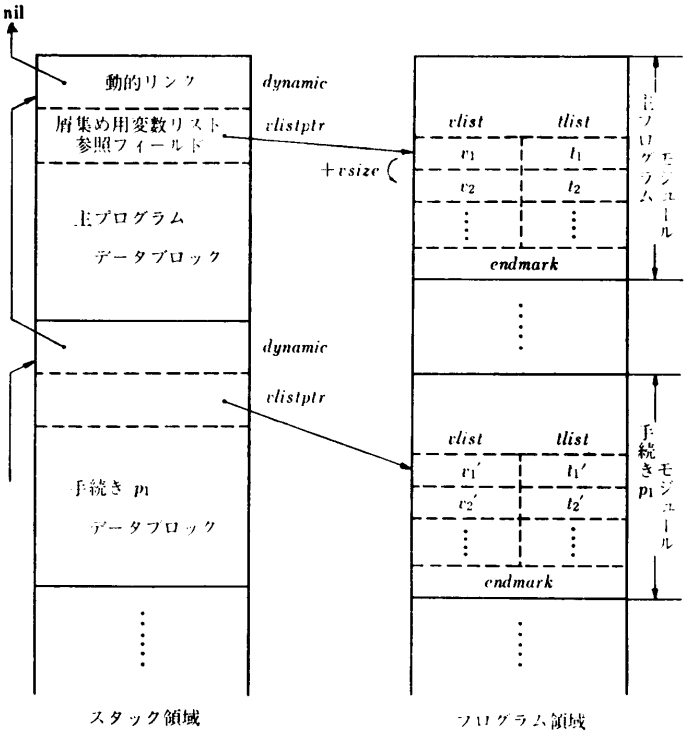


**図 2**　層集め用変数リストの参照方式
Fig. 2　Reference method to variable lists for the garbage collection.

```
address=0..max_address;
kind=(ptr, rec, fld, tag, recend, arr);
descriptor=
    record
      case typekind: kind of
  ptr: (reftype: address);   {reference type}
  rec: (fstfld: address);    {first field}
  fld: (fldadr: address;     {field address}
        fldtype: address;    {field type}
        nextfld: address);   {next field}
  tag: (tagadr: address;     {tag field address}
        tagvalue: integer;   {tag value}
        fstfield: address;   {first field}
        nextcase: address);  {next tag case}
recend: ( );
  arr: (comptype: address;   {component type}
        index: integer;      {number of component}
        size: integer)       {size of component}
    end;
```

**図 3** レコード型による型記述子の定義
Fig. 3 Definition of type descriptors by record type.

そのため**図 2** に示すように，動的変数の参照に関係するすべての変数とそのデータ型のリストを，翻訳時にプログラム領域に配置しておき，スタック領域中に割り当てられた主プログラム・手続き・関数の各データブロックの所定の欄から参照できるようにしておく．ただし，この変数リストには局所的変数ばかりでなく，手続き・関数のときには値パラメータ，さらに関数のときには関数値も動的変数の参照に関係があれば含めておく．図2で，変数リスト *vlist* には各変数のデータブロック内相対番地，型リスト *tlist* にはプログラム領域内におかれた型記述子の所在番地を与える．

屑集めのために必要なデータ型に関する情報は，ポインタ型とそれを取り巻く構造の記述である．**図 3** は屑集めのためのデータ型の記述を行うための型記述子をレコード型の定義で示したものである．タグフィールド *typekind* で分類するように，型記述子は6種からなる．*ptr* 記述子はポインタ型のための参照先の型記述子を，*rec* 記述子はレコード型の型記述の入口を，*arr* 記述子は配列型のもつ要素型，要素の数，および要素の占有する記憶容量を記述する．残りの三つはレコード型の型記述のためのもので，*fld* 記述子はフィールドのレコード内相対番地，フィールドの型記述子，および次のフィールドの型記述子を示し，*tag* 記述子はタグフィールドのレコード内相対番地，タグ値，最初のフィールドの型記述子，およびこのタグ値に対応しないときの次のタグ値に対する型記述子を示し，*recend* 記述子は *fld* または *tag* 記述子のリストの終了を示す．

**図 4** はポインタ型を含むデータ型と，それを型記述

```
type listp=↑listrec;
     listrec=
         record
           case atom: Boolean of
     true:  (name: array [1..10] of char);
     false: (left, right: listp)
     end;
     listarr=array [0..10] of listp;
a1: (ptr, a2)
a2: (rec, a3)
a3: (tag, 0, 0, a4, a7)
a4: (fld, d1, a1, a5)
a5: (fld, d2, a1, a6)
a6: (recend)
a7: (recend)
a8: (arr, a1, 11, sp)
```

**図 4** 型記述子によるデータ型の記述例
Fig. 4 An example of the description of data types by type descriptors.

子で記述した例である．同図で，$a_i$ $(1 \leq i \leq 8)$ は型記述子の所在番地を，$d_1$ および $d_2$ はそれぞれ *left* および *right* のレコード内相対番地を，$s_p$ はポインタ型に割り当てる記憶容量を表す．型記述はポインタ型に関係するフィールドのみを記述するため，タグフィールドが *true* になる場合の記述は必要ないので，*false* の場合のみ記述する．なお，$a_6$ の *recend* 記述子はフィールドのリストの終了に対する指示であり，$a_7$ の *recend* 記述子はタグフィールドの場合選択の終了に対する指示である．

屑集めの起動がかかると，その時点でのスタック領域の各データブロックから示される変数とその型記述子のリストをもとにして，**図 5** に示すようなアルゴリズムで参照可能なすべての動的変数に印付けを行う．ただし，ポインタ変数が未定義であってはならないので，新しくデータブロックを作るとき，および動的変数を生成するときには，あらかじめ初期化しておく．実現した処理系では，ポインタ変数のみ初期化するのはかえって手間がかかるので，手続き・関数の呼出しのときには局所的変数に割り当てた全領域に，動的変数の生成のときには割り当てた領域に0を格納している．また，標準 Pascal から唯一の制限事項として，タグフィールドがなく，タグ型のみを指定した可変部をもつレコード型を禁止する．タグフィールドがなければ，可変部の場合分けができないからである．各動的変数には参照可能か否かのフラグをもたせておくが，これもあらかじめ参照不可に初期化しておく．

図5は Pascal で記述しているが，番地計算に基づいてメモリを参照していくので，実際の処理系ではアセンブラ言語で記述したライブラリとして組み込んでいる．同図では型 *address* を添字とする配列変数

```
procedure marking;
  var vptr: address;
  procedure vmark(vad, typ: address);
    var ctr: integer; refad: address;
  begin
    case typekind(typ) of
ptr: begin refad:=memory(vad+basead);
       if (refad<>nil) or (refad<>0) then
         if not flag(refad) then
           begin flag(refad):=true;
             if reftype(typ)<>0 then
               vmark(refad, reftype(typ))
           end
     end;
rec: begin typ:=fstfld(typ);
       repeat
         case typekind(typ) of
   fld: begin vmark(vad+fldadr(typ), fldtype(typ));
              typ:=nextfld(typ)
         end;
   tag: if memory(vad+tagadr(typ))=tagvalue(typ)
          then typ:=fstfield(typ)
          else typ:=nextcase(typ);
recend: typ:=0
         end {case}
       until typ=0
     end;
arr: for ctr:=1 to index(typ) do
       begin vmark(vad, comptype(typ));
         vad:=vad+size(typ)
       end;
     end {case}
   end; {vmark}
 begin {marking}
   basead:=current_basead;
   repeat
     vptr:=vlistptr(basead);
     while vlist(vptr)<>endmark do
       begin vmark(vlist(vptr)+basead, tlist(vptr));
         vptr:=vptr+vsize
       end;
     basead:=dynamic(basead)
   until basead=nil
 end; {marking}
```

図 5 印付けアルゴリズム
Fig. 5 Marking algorithm.

*memory* でメモリを表している．手続き *marking* の本体はスタック領域のデータブロックから変数リストの入口番地を求め，そこに並べられているすべての変数に対し，参照可能な動的変数の印付けを手続き *vmark* に指示する．この印付けを主プログラムを含め，呼出し中のすべての手続き・関数に適用するため，呼出し関係を保持しているデータブロック内の動的リンクをたどりながら，主プログラムまで印付けの指示を繰り返す．なお，*basead* はデータブロックの入口番地を表す．

一方，手続き *vmark* では，指示された変数の相対番地と型記述子の所在番地を対にして，リンクでつながれた動的変数を追跡し，参照可能の印付けを行っていく．そのため，型記述子の *typekind* に従って処理を進めていくが，*rec* および *arr* の場合は各要素ごとに追跡するための場合分けであり，*ptr* の場合が参照可能な動的変数の発見に対応する．後者の場合，まずポインタの値を求め，それが **nil** または 0 でなく，かつ参照先の動的変数にすでに参照可能の印が付いていなければ，印付けを行った後，動的変数とその型記述子の番地を与えて手続き *vmark* を再帰的に呼び出す．一方，動的変数にすでに印が付いていれば，その先につながれているすべての動的変数はすでに追跡ずみであるので，以後の追跡は必要ない．

### 2.2 自由リスト方式の屑集め

この方式では，ヒープ領域内で回収した動的変数を連続した領域ごとにリストにしておき，動的変数の領域要求に対しては，まずこのリストの中から割り当てる．このリストを自由リストと呼ぶが，割当てにさいして，自由リストの前方に小領域が残り，むだな走査の繰返しを避けるため，前回割り当てた次の領域から走査を始める．自由リストの中から割り当てることができないときには，ヒープ領域に接する空き領域から割り当て，その分だけヒープ領域を拡大する．また，割当て不能な小領域が残り，領域が多数に分割されるのを防ぐため，実現した処理系では要求領域を 16 バイトで丸めた領域を割り当てている．以下の本節と次節では詳細な屑集めの手順を文献 5) に委ね，Pascal 処理系にとっての問題点を述べるにとどめる．

屑集めは次の 2 段階で行う．

1) 参照可能なすべての動的変数に印を付ける．
2) ヒープ領域中で自由リストに組み込まれていない全領域を走査し，参照可能でない領域を自由リストに組み込む．

上記操作のため，動的変数には要求領域のほかに，**図 6** に示すように参照可能か否かを示す印と，割り当てられた領域の大きさを保持する欄を設ける．後者は領域割当て時に書き込んでおき，屑集めの第 2 段階でヒープ領域を走査するときに用いる．

本方式を Pascal の屑集めに適用する場合，プログラムによっては致命的な問題を生じる．それは屑集めによってヒープ領域内の不要な領域は回収されるが，空き領域が増えるとは限らないことである．すなわち，スタック領域の近くに位置する動的変数が参照可能ならば，屑集めによってもスタック領域のための割当て領域が増えないこともありうるのである．このための対策としては，空き領域の大きさが一定値以下に

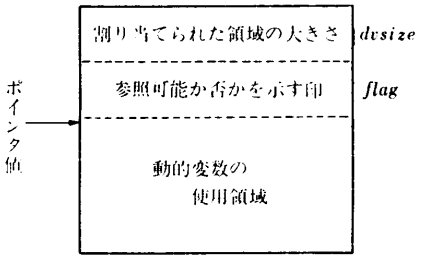


**図 6** 自由リスト方式における動的変数の構成
Fig. 6 Construction of a dynamic variable in the free list mode.

なったら屑集めを開始させ，スタック領域のための一定の空き領域を確保しておくことである．

屑集めの頻発を防ぐため，*new* の呼出しによる屑集めのとき，一定値以上の領域が新たに回収されないときには警告を発し，実行を中断する．また，手続き・関数の呼出しによる屑集めが1度起動されると，*new* の呼出しによる屑集めが発生するまで前者による屑集めの起動を中止するが，空き領域が消滅してしまう直前で再度の屑集めを試みる．

### 2.3 詰替え方式

この方式では，動的変数の領域要求があるとヒープ領域に接する空き領域から順次割り当てていく．屑集めのときには参照可能な動的変数はヒープ領域の下端に詰め替えるので，屑集めは空き領域が消滅する直前で行えばよい．詰替え方式のときは**図7**に示すように，動的変数に四つの欄を設け，次の4段階で屑集めを行う．

1) 参照可能なすべての動的変数に印を付ける．
2) ヒープ領域の全域を走査して，参照可能な動的変数のリストを欄 *nextvar* を用いて作るとともに，詰替え後の番地をヒープ領域の上端からの相対番地で欄 *newad* に書き込む．
3) すべてのポインタ変数を，詰め替えた後の番地に書き換える．
4) 第2段階で作った参照可能な動的変数のリストをもとに，参照可能なすべての動的変数をヒープ領域の下端に詰め替える．

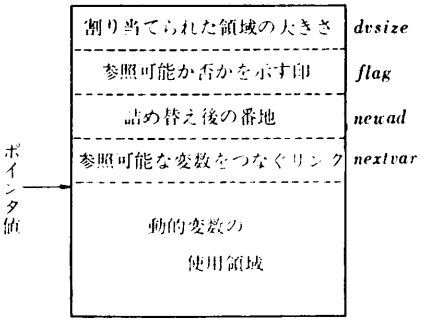


**図 7** 詰替え方式における動的変数の構成
Fig. 7 Construction of a dynamic variable in the compactification mode.

なお，図7に示す欄 *nextvar* は，第4段階の処理の便宜を図るためであり，この段階でもヒープ領域の全域を走査すれば省くことができる．第2段階で詰替え後の番地をヒープ領域の上端からの相対番地で書き込むのは，ヒープ領域の走査を上端から行うため，詰替え後の絶対番地が得られないからであって，第2段階が終了して初めて詰替え後のヒープ領域の上端 *heaptop* が確定する．第3段階は第1段階の印付けとほとんど同じ処理であり，ただ図5の手続き *vmark* の *case* 文で *ptr* のとき，*refad* が **nil** でも0でもないときに冒頭で，

$$memory(vad+basead) := newad(refad) + heaptop$$

の代入文を追加するだけでよい．欄 *flag* は屑集めの開始時点ではすべて初期化されているものとし，第1段階で付けた印は第2段階で初期化し，第3段階で付けた印は第4段階で再び初期化する．本方式でも，一定値以上の領域を回収できないときには，警告を発し実行を中断する．

詰替え方式では，内部的なポインタ値が屑集めにより変わってしまうので，ポインタ値に関する最適化処理を行っているときには注意が必要である．たとえば，

$$p\uparrow.q\uparrow.a := b;\ new(r);\ p\uparrow.q\uparrow.f := r;$$

において，ポインタ値 $p\uparrow.q$ をレジスタに保持しておいても，*new* の呼出しに伴う屑集めで $p\uparrow.q$ の値は変わってしまうからである．これを避けるためには，*new* あるいは手続き・関数の呼出しをまたいだポインタ値に関する最適化処理はやめればよいが，**with** 文で指定されたレコード変数のうち動的なものは，その番地計算をやり直さなければならない．しかし，**with** 文は一般に入れ子をなしており，それらの番地計算をやり直すように翻訳することははなはだ困難でもあるし，能率も悪い．そこで実現した処理系では，**with** 文で指定されたレコード変数の番地を作業用のポインタ変数にもたせるように翻訳し，屑集めのときに活動中の **with** 文に対応したすべての作業用ポインタ変数を屑集めルーチンに教え，その書き換えをさせている．そのため，**図8**に示すように，作業用のポインタ変数の番地をプログラム領域にリストにして配置

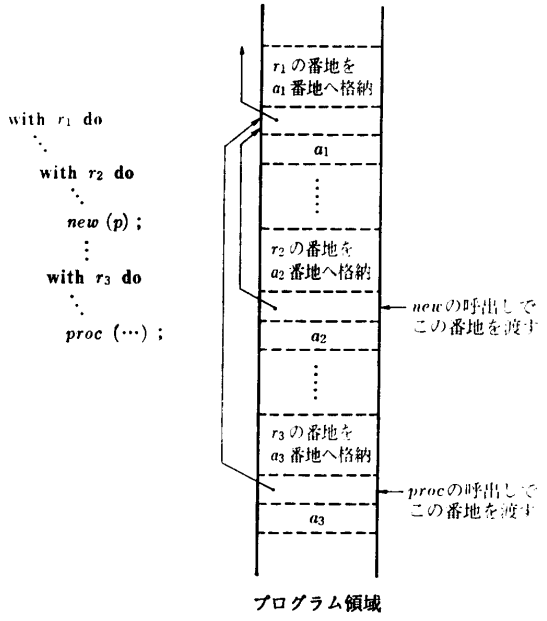


**図 8 with 文に現れたレコード変数の連結構造**
Fig. 8 Linking structure of record variables written in **with** statements.

しておき，*new* または手続き・関数の呼出し時に，最も内側の **with** 文に対応したポインタ変数の番地を置いた個所を呼出し先に伝える．屑集めルーチンはスタック領域の動的リンクを通して，呼出し中のすべての手続き・関数の活動中の **with** 文の作業用ポインタ変数をたどることができ，屑集めの第3段階でこれらの変数を書き換える．ただし，**with** 文で指定されたレコード変数は動的変数とは限らないので，ヒープ領域を指している変数のみを対象とする．また，レコード変数の番地をレジスタに保持している場合には，*new* または手続き・関数から戻ってきた後に，作業用のポインタ変数でレジスタを設定し直す．

## 3. 屑集めの実測例

屑集め機能を備えた Pascal 処理系のもとで実行させるプログラム例として選んだのは Pascal コンパイラ自身である．コンパイラはほとんどの表を動的データ構造を用いて記述しており，しかもヒープ領域をスタックとして使っているので，もともとは標準手続き *release* を用いて不要になった領域を返却していた．今回，この手続き呼出しを削除し，屑集め機能を備えた処理系のもとで稼動させ，その翻訳時間を計測した．なお，翻訳させたのは再びコンパイラ自身であり，実行させるコンパイラに割り当てるデータ領域を調節し，屑集めの実施回数をかえてみた．使用した計算機は HITAC-200 H であり，結果を**表 1** に示す．同表の CPU 時間はユーザに課金される時間であって，OS に関係する CPU 時間は含んでいない．また，入出力関係の状態によって，この CPU 時間は若干変動する．

**表 1 屑集めのもとでの実行時間例**
Table 1 An example of execution times under the garbage collection.

(i) 自由リスト方式

| データ領域 (kB) | CPU 時間 (sec) | 屑集め回数 |
|---|---|---|
| 1,000 | 18.76 | 0 |
| 800 | 18.95 | 1 |
| 400 | 18.79 | 2 |
| 350 | 18.73 | 3 |
| 300 | 18.73 | 4 |
| 250 | 18.72 | 6 |
| 200 | 19.07 | 10 |
| 170 | 19.42 | 18 |
| 150 | 19.90 | 37 |

(ii) 詰替え方式

| データ領域 (kB) | CPU 時間 (sec) | 屑集め回数 |
|---|---|---|
| 1,200 | 18.63 | 0 |
| 1,000 | 18.83 | 1 |
| 600 | 18.75 | 1 |
| 500 | 18.81 | 2 |
| 400 | 18.73 | 3 |
| 300 | 18.82 | 5 |
| 200 | 19.63 | 14 |
| 180 | 19.84 | 20 |
| 160 | 21.26 | 40 |

同表からわかるように，いずれの方式でも 10 回以下の屑集めを実施したのでは，それに要した CPU 時間のオーバヘッドはほとんど計測時の変動分にうずもれてしまう程度である．自由リスト方式で 37 回の屑集めで約 6 %，詰替え方式の 40 回の屑集めで約 14% のオーバヘッドを有しているにすぎない．

印付けを行うときに作業用スタックを必要とし，実現した処理系ではつながれた動的変数が現れるたびごとに 20 バイトを要する．前述の翻訳例で要したスタック量は最大で 6,960 バイトであり，データ領域がむしろ小さいときに発生している．これはデータ領域が小さいと屑集めが頻発し，リンクでつながれた大きなデータ構造がヒープ領域に存在するときに屑集めが行われる機会が増えたためと思われる．

## 4. おわりに

動的変数を多用するプログラムで，不要になった変数の返却も考えなければならないのは，プログラマに

余分な負担を負わせ，ときには虫の原因ともなる．実現した処理系では，動的変数の参照に関係する変数とその型記述子のうめ込みに，コンパイラへ Pascal のソース・プログラムで約100行の追加であり，屑集めルーチンがアセンブラ言語で約200ステップ程度である．屑集めは標準手続き *dispose* あるいは *release* と共存することに何ら問題はないので，プログラマが不要になった動的変数を陽に指定することを排除する必要はない．ゆえに，Pascal の仕様，処理系の手直し，屑集めのオーバヘッド，および屑集めによるプログラマの負担軽減を考えると，Pascal 処理系が屑集め機能を備えることはきわめて有効であり，かつ現実的であることを確認した．

**謝辞** 北海道大学工学部竹村伸一教授からは本研究を進めるに当たって貴重なご教示をいただき，また桃内佳雄助手からは本文をまとめるに当たり懇切なご助言をいただいた．ここに記して深謝申し上げます．

## 参考文献

1) Ammann, U.: Compiler List for Pascal 6000-3.4, *ETH Zür.* (1974).
2) Nori, K. V., Ammann, U., Jensen, K. and Nägeli, H. H.: The PASCAL 〈P〉 Compiler: Implementation Notes, Berichte des instituts für Informatik, *ETH Zür.*, Nr. 10, p. 57 (1974).
3) 中内伸二，萩原兼一，都倉信樹，鈴木直也：拡張されたヒープ領域管理機能をもつ PASCAL 処理系 ELPH の作成，電子計算機研究会資料，EC 82-19, pp. 73-82 (1982).
4) Cohen, J.: Garbage Collection of Linked Data Structures, *ACM Comput. Surv.*, Vol. 13, No. 3, pp. 341-367 (1981).
5) 日比野靖：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 (1982).
6) Ammann, U.: Run-time Storage Organization, Berichte des Instituts für Informatik, *ETH Zür.*, Nr. 25, p. 46 (1978).

(昭和58年2月25日受付)
(昭和58年4月19日採録)