社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.
SS96-3 (1996-05)

プログラム実行中にメモリをデータから暗黙的かつ効率的に解放する一方式

高田 義広, 鳥居 宏次

奈良先端科学技術大学院大学 情報科学研究科

〒630-01 奈良県 生駒市 高山町 8916 番地の 5
e-mail: takada@is.aist-nara.ac.jp

あらまし　計算機プログラムの実行中には, 必要なくなったデータに割り当てられているメモリ領域を, 他のデータのために解放する必要がある. このようなメモリの解放については, プログラマにメモリの解放を明示的に記述してもらわずに暗黙的に行なう方式が望ましい. そのような方式は, 従来より, 数多く提案されているが, システムを実現する上でそれぞれに問題がある. 例えば, ゴミ集めを行なう方式では, メモリが不足する度にプログラムの実行が一時中断すると言う問題がある. 本稿では, 従来の方式を比較して問題を列挙すると共に, それらの問題を解決した新しい方式を提案する.

キーワード　プログラミング言語, 言語処理系, メモリの割り当て, メモリの解放, ゴミ集め

A Method for Freeing Memory Blocks Efficiently and Implicitly during Program Execution

Yoshihiro TAKADA and Koji TORII

Graduate School of Information Science
Nara Institute of Science and Technology

8916-5 Takayama-cho, Ikoma-shi, Nara 630-01
e-mail: takada@is.aist-nara.ac.jp

Abstract　During program execution, it is necessary to free memory blocks that have been allocated to unnecessary data, for use for other data. With respect to such memory free, it is preferable that programmers should not be required to describe memory free explicitly and a processor should free memory implicitly. Methods for such implicit memory free have already been proposed, but each of the methods has its own problem related to system development. For example, the methods of garbage collection have a problem of suspending program execution at each time when memory is exhausted. This paper summarizes problems of the previous methods and presents a new method that solves the problems.

key words　programming language, language processing system, memory allocation, memory free, garbage collection

1 はじめに

計算機プログラムの実行中には, プロセッサの参照する各データに対して, それを保持するためのメモリ領域が割り当てられる. ただし, 計算機のメモリの容量には限りがあるので, 必要なくなったデータに割り当てられているメモリ領域は, 他のデータのために解放する必要がある. これをデータの処分と言うことにする.

データの処分には, 大別して2通りの方式がある. 1つは, プログラマにプログラムの中にデータの処分を明示的に記述してもらい, プロセッサがそれに従って処分する方式である. もう1つは, プログラマにそのような命令を記述してもらわずに, プロセッサやコンパイラなどの言語処理系が何らかの方法で自動的に処分する方式である. プログラマにとっては, プログラムの作成の労力やプログラムに誤りを混入させる危険性を考えると, 暗黙的にデータを処分する後者の方式が望ましい. そこで, 従来より, 暗黙的な処分の方式が, プログラミング言語を設計したり言語処理系を開発する上で工夫されている. [2, 4]

ところが, 従来の暗黙的な処分の方式には, 3章で述べるように, それぞれに問題があった [1]. 従来の代表的な方式としては, 例えば, 次の2つがある.

1つは, 各データについて, それを直接的に参照できるプログラムの部分に着目して, プログラムの実行がその部分を越えた時点で暗黙的に処分する方式である (3.1). 例えば, 手続きに局所的に宣言されている変数ついては, 手続きの中だけで名前によって直接的に参照されるので, 手続きの終了時に暗黙的に処分する. ある手続きの実行中に名前などによって直接的に参照できるデータの集まりは, その手続きの環境と呼ばれている. つまり, この方式では, どの環境にも含まれないデータを, 保持する必要のないデータとみなして処分する. これは, 手続き型のプログラミング言語の局所変数などに一般的に用いられている方式である.

この方式には, 環境に含まれなくても保持する必要のあるようなデータを扱えないと言う問題がある. つまり, 環境に含まれなくてもポインタを通して間接的に参照されるようなデータを扱えない. 例えば, ポインタを用いて構成されるデータ構造の一部となるようなデータである. 実際のシステムにおいては, そのようなポインタを含むデータ構造が頻繁に使われている. もし, そのようなデータ構造が扱えないとすれば, 実現できるシステムの機能が大幅に制限されることになる.

もう1つは, 各データについて, 環境に含まれるかどうかだけでなく, ポインタを通して参照できるかどうかも調べる方式である (3.2)[5]. つまり, 名前による参照も, ポインタによる参照も, 平等に, 参照のための1個の路であると考えて, そのような路が1個も存在しないデータを暗黙的に処分する. ただし, 特定のデータの参照のための路が存在するかどうかを調べることは, 全てのデータについて参照のための路が存在するかどうかを調べることと同程度に時間がかかる. そこで, この方式では, メモリが不足した時点で, 全てのデータについて参照のための路が存在するかどうかを一括して調べて, データを処分する. この処理は, ゴミ集めと呼ばれている.

この方式では, 環境に含まれなくてもポインタを通して参照されるデータも扱える. その代わり, ゴミ集めが負荷の高い処理であるために, メモリの不足する度に, プログラムの実行が一時中断してしまう. この問題は, 特に, 即応性を要求されるシステムの開発において重大である.

そこで, 本稿では, 暗黙的にデータを処分する新しい方式として, 以上の問題を同時に解決する方式を提案する. つまり, ゴミ集めのような負荷の高い処理を必要とせず, しかも, 環境に含まれてなくてもポインタを通して参照されるようなデータも適切に処分できる方式を提案する.

提案する方式では, 各データについて, そのデータの参照のための路の中の特定の1個が主たる路であると考える. そして, その主たる路が存在しなくなる時点で, そのデータが参照されなくなるとみなし, 暗黙的に処分する. 主たる路は, 名前による参照だけに限らず, ポインタによる参照でもよい. 従って, この方式では, 環境に含まれていなくてもポインタを通して参照されるデータも扱える. そして, 参照のための路が1個も存在しないかどうかを調べる必要がないので, ゴミ集めのような負荷の高い処理も必要ない. (4章)

提案する方式は, 特定のプログラミング言語に依存する方式ではない. C, C++, Pascal, Ada などのほとんどの手続き型言語に対して, 構文を一部拡張して言語処理系を作り直すことにより導入できる. ただし, 本稿の以降の部分では, 方式を具体的に示せるように, Pascal 風言語を例にとる.

```
procedure calc ();
      type vec = record X, Y, Z: integer;
                 end;
      procedure init ( var P: vec;
                       X, Y, Z: integer);
      begin P.X := X; P.Y := Y; P.Z := Z;
      end;
      procedure operator− (var L, R: vec) : vec;
      begin var P: vec;
            P.X := L.X − R.X;
            P.Y := L.Y − R.Y;
            P.Z := L.Z − R.Z;
            return P;
      end;
      .....
begin .....
      var P, Q: vec;
      init(P, .....);
      init(Q, .....);
      var R: vec = P − Q;
      .....
end;
```

図 1: 対象言語のプログラムの例

2 対象言語

本稿で例にとるプログラミング言語について述べる．この言語は，Pascal に似せているが，C++ や Ada に見られる新しい特徴を採り入れている [3]．例えば，演算を定義できること，手続きの途中で変数を宣言できること，手続きの戻り値に名前を付けられることなどである．

図 1 は，この言語で書かれたプログラムの例である．このプログラムは，ベクトルを表す構造データ型 vec を用いて何らかの計算を行なう手続き calc の記述である．手続き calc の記述の先頭では，vec 型の構造が type 文によって記述されている．その後に，vec 型のデータを初期化する手続き init，ベクトルの減算を表す 2 項演算 '−' が記述されている．その後に，手続き calc の本体が記述されている．

以下では，各構文要素について少し詳しく述べる．

2.1 手続き

図 1 の手続き init と 2 項演算 '−' とを比べるとわかるように，手続きと演算の定義に大きな相違はない．そこで，この言語では，通常の手続きも演算も区別せずに，単に手続きと呼ぶ．代入も，C 言語などにおいてと同様に，手続きの 1 つであると考える．また，戻り値のない手続きも，戻り値のある関数も区別せずに，手続きと呼ぶ．

2.2 変数の宣言

手続きの中では変数を宣言できる．変数は，図 1 においてのように，'var' と言う予約語によって宣言される．手続きの実行中には，実行が宣言の箇所にかかった時点で変数にメモリ領域が割り当てられる．その後は，手続きが終了するまで，その変数を名前によって直接的に参照できるようになる．

一般に，ある手続きの実行中に名前などによって直接的に参照できるデータの集合は，その手続きの環境と呼ばれている．変数が宣言されると環境は次のように変化する．まず，宣言によって変数にメモリ領域が割り当てられる時点で，その変数が環境に加えられる．その後は，環境を通してその変数が参照される．もし，同じ手続きの中で同じ名前のデータが環境に加えられると，その変数は環境から除かれる．最後に，その手続きが終了すると，その変数が環境から除かれ，環境自身も無くなる．

2.3 手続きの引数

手続きの引数には，参照渡しされる引数と値渡しされる引数との 2 種類がある．図 1 では，引数の宣言に 'var' と言う予約語が付けられている引数が参照渡しされ，付いていない引数が値渡しされる．引数が値渡しされる時はデータのコピーが生成されて渡され，参照渡しされる時はデータのコピーが生成されない．つまり，参照渡しの時には，そのデータがその手続きの環境に加えられるだけである．値渡しの時には，コピーが環境に加えられる．

値渡しの時に作成されるコピーについては，当然ながら，メモリ領域が割り当てられるので，必要なくなった時点で処分する必要がある．

2.4 手続きの戻り値

手続きの戻り値にも，参照渡しされる戻り値と値渡しされる戻り値との 2 種類がある．図 1 においては，'−' と言う手続きの vec 型の戻り値が値渡しされる．calc や init と言う手続きは，戻り値を持たない．参照渡しされる戻り値はない．ある手続きが参照渡しの戻り値を持つことを記述するには，procedure 文の末尾に，戻り値の型と合わせて 'var' と言う予約語を付ける．

2.5 戻り値への命名

参照渡しされる場合でも値渡しされる場合でも，戻り値には名前を付けることができる．図 1 においては，下から 3 行目で，'$P - Q$' と言う手続き呼び出しの結果として値渡しされるデータに，Rと言う名前が付けられている．戻り値に名前を付けるこ

とと代入とは, 全く異なることに注意して頂きたい. 例えば, '$R := P - Q$' と言う文は, ':=' と言う手続きを呼び出すことを意味する. そして, その手続きは, '$P - Q$' と言う手続き呼び出しの戻り値を Rへコピーする. 戻り値に Rと言う名前を付けることは, 手続き呼び出しでないし, データのコピーを伴わない.

手続き A から手続き B を呼び出した時の戻り値に関する環境の変化は, やや複雑であり, 次のようになる. まず, 手続き B の中の return 文の実行により, 戻り値が手続き A の環境に加えられる. 値渡しされる場合はデータのコピーが環境に加えられる. この瞬間には, 戻り値に名前がない. そして, 手続き B が終了して, 手続き B の環境は無くなる. 手続き A においては, 環境に加えられた戻り値に名前が付けられる場合とそうでない場合とがある. 名前が付けられる場合, その戻り値は, それ以降も直接的に参照でき, 手続き A の終了時まで環境に残る. 名前が付けられない場合, その戻り値は, 次の文が実行されるまでに環境から除かれる.

2.6 データの種類

この言語におけるデータは, ポインタデータ, 構造データ, 単純データの 3 種類に大別される. ポインタデータとは, メモリ中の他のデータを指すポインタのデータである. 型 t のデータを指すポインタデータの型は, '↑t' と表記する. 構造データとは, 配列やレコードのように構造を持つデータである. 単純データとは, ポインタデータと構造データを除くデータである. 例えば, interger 型データや文字型データである.

3 従来の方式

2 章で述べた言語に従来の代表的な方式を適用した場合について述べ, それぞれの問題を指摘する.

3.1 環境だけに注目する方式

3.1.1 概要

この方式では, どの環境にも含まれなくなったデータは保持する必要がなくなったとみなす. そして, 各データについて, どの環境にも含まれなくなる時点で暗黙的に処分する.

3.1.2 詳細

手続きに局所的に宣言されている変数については, その手続きの終了までに環境から除かれるので, その時点で処分する. 手続きへ値渡しされる引数についても, 同様である. ただし, 手続きへ参照渡しされる引数については, その時点では処分しない. 手続き A から手続き B を呼び出す時に参照渡しされる引数は, 手続き A の環境にも手続き B の環境にも含まれるので, 手続き B の環境から除かれただけでは処分しない. 手続きへ値渡しされる戻り値については, 戻り値に名前が付けられる場合と付けられない場合とで処分が異なる. 名前が付けられない場合, 次の文が実行されるまでに処分する. 名前が付けられる場合, その手続きの終了までに処分する.

```
var S: array [1..N] of ↑vec;
var Cnt: integer;
procedure push (var X: vec);
begin put(S, Cnt, X);
      Cnt := Cnt + 1;
end;
procedure pop ();
begin Cnt := Cnt − 1;
      get(S, Cnt);
end;
procedure calc ();
begin .....
      var A: vec;
      .....
      push(A);
end;
```

図 2: 対象言語のスタックのプログラムの例

3.1.3 プログラム作成上の制約

この方式では, 各データについて, 環境に含まれないと参照されないことを前提としているので, プログラムがプログラムを作成する時にもその制約が課せられる. つまり, 有効な名前がなくなったデータをポインタを通して間接的に参照するようなプログラムを作成してはならない.

例として, 図 2 のプログラムを考える. このプログラムは, スタックに vec 型のデータを push する手続きの記述であるが, この方式では正しく実行されない. A と言う vec 型のデータが push されると, そのデータは Sと言う配列よりポインタが張られるようになる. ところが, 手続き calc が終了すると, A が環境から除かれる. そのために, 張られたポインタはそのままでも, A が処分されてしまい, 不整合が生じる. この方式では, このようなプログラムを作成してはならない.

3.1.4 問題

この方式の問題は, 前節で示したように, 環境に含まれなくてもポインタを通して間接的に参照される可能性のあるデータを扱えないことである. つまり, この方式だけでは, ポインタを用いて構成される複雑なデータ構造を実現できない.

3.2 ゴミ集めを行なう方式

3.2.1 概要

この方式では, 各データについて, 環境に含まれるかどうかだけでなく, ポインタを通して参照できるかどうかも調べて処分する. つまり, 名前などによる直接的な参照も, ポインタによる間接的な参照も, 平等に, 参照のための1個の路と考えて, そのような路が1個も存在しないデータを処分する. ただし, 各データについて, 参照のための路がなくなった直後に処分する訳ではない. メモリが不足した時点で, 全てのデータについて参照のための路が存在するかどうかを一括して調べて処分する. このような処理は, ゴミ集めと呼ばれている. Smalltalk や Lisp などの言語の処理系によく用いられている方式である.

3.2.2 詳細

代表的なゴミ集めの手順は次の通りである [5]. まず, その時点でどれかの環境に含まれるデータに何らかの印を付ける. そして, 印のあるデータからポインタの張られているデータにも, 再帰的に, 印を付ける. この時, 印のないデータについては, 参照のための路が存在しない. そこで, 次に, メモリ中の全てのデータを調べて, 印のないデータを処分する.

3.2.3 プログラム作成上の制約

この方式では, 環境に含まれないデータをポインタを通して間接的に参照することが許される分だけ, 3.1 の方式よりも, プログラムの作成時に課せられる制約が少ない. そのために, 3.1 の方式では正しく実行されない図 2 のプログラムも, この方式では正しく実行される. 手続き calc が終了して A と言うデータが環境から除かれても, Sと言う配列からポインタが張られている A は, 処分されないからである.

3.2.4 問題

この方式の問題は, ゴミ集めの度に, プログラムの実行が一時中断することである. 3.2.2 に示したように, ゴミ集めは時間のかかる処理であり, しかも, プログラムの実行を停止しておかなければ実行できない. この問題は, 特に, 即応性を要求される実時間システムにおいて重大である.

3.3 参照カウンタ方式

3.3.1 概要

この方式では, ゴミ集めを行なう方式と同様に, 参照のための路が1個も存在しないデータを処分する. ただし, そのような処分するべきデータを見つけるために, より効率的な方法を採る. 各データについて, そのデータの参照のための路の数を表すカウンタを暗黙的に用意して, そのカウンタが 0 になった時点でデータを処分する.

3.3.2 詳細

この方式では, 参照のための路が変化する次のような時点で, 参照カウンタを自動的に更新する.

- 環境にデータが加えられたり, 除かれたりする時. または, 環境が無くなる時.
- ポインタデータが書き変えられたり, 処分されたりする時.

そして, 各データについて, 参照カウンタが 0 になった時点でデータを処分する.

3.3.3 プログラム作成上の制約

この方式では, コミ集めを行なう方式と同様にプログラムの作成の時に課せられる制約が少ない. また, 参照カウンタの作成や更新は, 暗黙的に行なわれるので, プログラムの作成に影響しない.

3.3.4 問題

この方式では, 3.2.2 のような負荷の高い処理を必要としないので, 即応性の問題がない. その代わり, 次の重大な問題がある. それは, いくつかのデータの間でポインタがループを成している場合, それらのデータについては, 参照のための路が存在しなくても参照カウンタが 0 にならず, 適切に処分されないことである [5]. そのような不要なデータが蓄積されると, 次第にメモリが不足して致命的な実行時エラーが発生する危険性がある.

なお, この方式では, 参照カウンタのためのメモリ領域を割り当てなければならないので, 他の方式と比べて多くのメモリを必要とする.

3.4 明示的な方式と組合わせた方式

3.4.1 概要

この方式では, 環境に含まれなくなると参照されなくなるデータと, 環境に含まれなくてもポインタを通して間接的に参照されるデータとを, プログラマに区別してもらう. そして, 環境に含まれなくな

```
var S: array [1..N] of ↑vec;
var Cnt: integer;
procedure push (var X: vec);
begin put(S, Cnt, X);
      Cnt := Cnt + 1;
end;
procedure pop ();
begin Cnt := Cnt − 1;
      delete get(S, Cnt);
end;
procedure calc ();
begin .....
      var A: vec = new vec;
      .....
      push(A);
end;
```

図 3: 組合わせ方式のプログラムの例

ると参照されなくなるデータについては, 環境から除かれる時点で暗黙的に処分する. 環境に含まれなくてもポインタを通して参照されるデータについては, 明示的な方式で処分する. つまり, データの処分のための命令をプログラマにプログラム中に記述してもらい, それに従って処分する. C, C++, Pascal, Ada などの言語によく用いられている方式である.

3.4.2 詳細

この方式では, 明示的なデータの処分のために, 動的にデータにメモリ領域を割り当てたり処分したりするための命令を定義する. 本稿では, それらの命令を 'new' と 'delete' と言う名前にする. (C 言語では, malloc と free と言う手続きが定義されている.) そして, 環境に含まれなくてもポインタを通して参照されるデータについては, プログラム中に記述されるそれらの命令に従って, メモリ領域を割り当てたり処分したりする. その他のデータについては, 3.1 の方式と同様に, どの環境にも含まれなくなる時点で処分する.

図 3 は, 図 2 のプログラムをこの方式に合わせて書き直したプログラムである. このプログラムでは, A と言うデータが明示的に処分され, 他のデータは暗黙的に処分される.

3.4.3 プログラム作成上の制約

この方式では, プログラマがプログラムを作成する時に, まず, 各データについて, 環境に含まれなくてもポインタを通して参照されるかどうかを判断することが要求される. そして, 環境に含まれなくても参照されると判断されたデータについては, プログラマの責任で, メモリ領域の割り当てや処分のための命令を適切に記述することが要求される.

また, 環境に含まれなくなると参照されなくなると判断されたデータについては, 3.1 の方式と同様に, 環境に含まれなくなったデータがポインタを通して参照されることのないよう注意する必要がある.

3.4.4 問題

この方式では, 3.2.2 のような負荷の高い処理が必要ないし, 環境に含まれなくても参照される可能性のあるデータも扱える. 代わりに, 2 個の重大な問題がある.

1 つは, 明示的に処分するデータと暗黙的に処分するデータとがプログラム中に混在することになるので, プログラムの作成が他の方式に比べて非常に複雑になることである.

もう 1 つは, 致命的な実行時エラーを引き起こす誤りを, プログラムに混入させやすいことである. 例えば, 明示的に処分するべきデータについて, 処分の命令を記述し忘れると, 次第にメモリが不足して, 致命的な実行時エラーが発生する. また, 処分するべきでない時点で誤って処分の命令を記述すると, やはり, 致命的な実行時エラーが発生する.

4 提案する方式

4.1 概要

提案する方式は, 次の仮説に基づいている.

> プログラマは, 保持する必要のある各データについて, そのデータの参照のための路の中の特定の 1 個を, 主たる路であると考えている. 各データに対する主たる路は, プログラムの実行中に他の路と入れ換わることがあるが, そのデータを保持する必要のある限り, とにかく, 必ず 1 個だけ存在する. 逆に言うと, 主たる路が存在しないデータは, 保持する必要がない.

例として, 図 2 のスタックのプログラムを考える. S や Cnt と言うデータについては, その名前によってしか参照されないので, それが参照のための主たる路である. A と言う vec 型のデータについては, スタックに push される前後で参照のための主たる路が入れ換わる. push される前には, 名前によってしか参照できないので, それが主たる路である. push された後には, そのデータの名前には関係なく, S と

言う配列からポインタが張られていることが重視される．つまり，そのポインタによる参照が主たる路となる．

以上の仮説に基づいて，提案する方式では，次のようにデータを処分する．まず，各データの参照のための主たる路がどのように移動するかが分かるように，プログラマにプログラム中に記述を加えてもらう．そして，その記述に基づいて，各データについて，主たる路が存在しなくなる時点を調べて，その時点で暗黙的に処分する．

以下では，本方式の詳細について述べる．

4.2 データの所有権

各データについて，参照のための主たる路が存在すること，および，その主たる路が入れ換わることがあることは，データの所有権と言う概念を導入すると直観的に理解しやすい．つまり，データを所有する権利が，そのデータの参照のための路の間を移っていくと考える．この概念を用いると，本方式は次のように言い換えられる．

プログラマは，各データについて，その所有権が，参照のためのどの路に属するかを常に意識してプログラムを作成していると思われる．そこで，各データの所有権がどのように移動するかが分かるように，プログラマにプログラム中に記述を加えてもらう．そして，その記述に基づいて，各データについて，所有権の消滅する時点を調べて，その時点で暗黙的に処分する．

この方式では，環境に含まれなくてもポインタを通して参照されるデータもそうでないデータも，区別なく扱われ，どちらのデータも適切な時点で処分できる．また，後に述べるように，所有権の消滅する時点はプログラム中の記述から容易に調べられるので，データを処分するために負荷の高い処理を行なう必要もない．

以下では，データの所有権について，少し詳しく述べる．

各データの所有権は，そのデータにメモリ領域が割り当てられる時点で発生する．そして，そのデータの参照のための路のどれか1個に属する．各データの所有権は，次のような時点で，参照のためのある路から他の路へ移動することがある．ただし，複製されたり分割されることは決してない．

- データが引数や戻り値として参照渡しされる時．
- あるデータを指すようにポインタデータが書き変えられる時．
- あるポインタデータからポインタを読み取る時．

参照のためのある路に属する所有権は，他の路へ移動する前に，その路が存在しなくなると消滅する．具体的には，次のような場合に所有権が消滅する．

- そのデータが環境から除かれる時．例えば，手続きが終了する時．
- そのデータへのポインタを含むポインタデータが書き換えられたり処分されたりする時．

4.3 構文の拡張

本方式では，データの所有権がどのように移動するかをプログラマにプログラム中に記述してもらうことが重要である．そのために，言語の構文に次の2つの拡張を加える．

1つは，手続きの引数や戻り値が参照渡しされる場合に，所有権の移動が伴うかどうかを示すための構文である．本稿では，所有権の移動を伴わない場合，引数や戻り値の宣言に，'var' の代わりに 'ref' と言う予約語を付けることにする．所有権の移動を伴う場合，'var' と言う予約語を以前の通りに付けることにする．

もう1つは，各ポインタデータについて，それが所有権の属するポインタかどうかを示すための構文である．本稿では，所有権の属さないポインタデータには，その宣言におけるデータ型の指定に 'ref' と言う予約語を付けることにする．

以上の拡張によって，各データの所有権がどのように移動するかが分かるようになる．拡張した構文に合わせて図1のプログラムを書き直したプログラムを図4に示す．手続き calc において宣言されている P と Q と言うデータについては，所有権の移動を伴う参照渡しの対象になっていないので，それらの所有権は手続き calc の環境に属したまま，手続き calc の終了時に消滅する．手続き '_' において宣言されている P と言うデータについては，所有権の移動を伴う参照渡しの対象になっている．手続き '_' を呼び出した時の所有権の移動は次の通りである．まず，P と言うデータにメモリ領域が割り当てられて，その所有権が手続き '_' の環境に発生する．そして，'return P' が実行されると，その所有権が手続き calc の環境に移動する．そこで，そのデータには，R と言う名前が付けられるので，手続き calc の環境に残されることになる．そして，それ以上移動しなければ，手続き calc の終了時に消滅する．

```
begin begin kill procedure calc ();
      type vec = record X, Y, Z: integer;
                 end;
      procedure init ( ref P: vec;
                       ref X, Y, Z: integer);
      begin P.X := X; P.Y := Y; P.Z := Z;
      end;
      procedure operator− (ref L, R: vec) : vec var;
      begin var P: vec;
            P.X := L.X − R.X;
            P.Y := L.Y − R.Y;
            P.Z := L.Z − R.Z;
            return P;
      end;
      .....
begin .....
      var P, Q: vec;
      init(P, .....);
      init(Q, .....);
      var R: vec = P − Q;
      .....
end;
```

図 4: 提案する方式のプログラムの例

4.4 プログラムの作成における制約

この方式では, ゴミ集めを行なう方式などにおいてと同様に, 環境に含まれなくてもポインタを通して参照されるデータを扱えるので, プログラムの作成時に課せられる制約は比較的少ない. 一方, 各データの所有権に関する記述が要求される分だけ, プログラムの作成が多少複雑になる.

ただし, 4.1 の仮説が妥当であるとすれば, データの所有権に関する記述は容易に作成できると考えられる. 仮説が妥当であるとすれば, この方式を採るかどうかに関わらず, プログラマは, 各データについての所有権の移動を常に意識しながらプログラムを作成していることになる. 従って, それを記述することは難しくない.

5 まとめ

プログラムの実行中にデータを暗黙的に処分することについて, 従来の代表的な方式の問題をそれぞれ指摘した. そして, それらの問題を解決した新しい方式を提案した. 従来の方式と新しい方式との特徴を表 1 にまとめておく. 表からも分かるように, 提案した方式では, ゴミ集めのような負荷の高い処理を必要としない. そして, 環境に含まれなくなってもポインタを通して間接的に参照されるようなデータも適切に処分できる.

表 1: 方式の比較

特徴 A : 環境に含まれなくてもポインタを通して間接的に参照されるデータを扱える.
特徴 B : プログラムの実行の中断するような負荷の高い処理が必要ない.
特徴 C : プログラムに致命的な誤りの混入する危険性が少ない.
特徴 D : 不要なデータが処分されずに残らない.
特徴 E : プログラムの作成が容易である.
特徴 F : 余分なメモリ領域を必要としない.

	環境だけに注目する方式	ゴミ集めを行なう方式	参照カウンタ方式	明示的な方式と組合わせた方式	所有権方式
特徴 A	×	○	○	○	○
特徴 B	○	×	○	○	○
特徴 C	×	○	○	×	△
特徴 D	○	○	×	△	○
特徴 E	○	○	○	×	○*
特徴 F	○	△	×	○	△

(*更なる評価を要する.)

今後は, 本方式を適用した言語処理系を実装して, 本方式の有効性を評価する予定である.

参考文献

[1] 阿部, 高田, 鳥居: “プログラミング言語において動的データと宣言的データとを統一的に扱う方式”, 情報処理学会 第 51 回全国大会, 1M-2 (1995).

[2] Aho, A. V., Ullman, J. D. 著, 土居 範久 訳: “コンパイラ”, 培風館 (1986).

[3] Lippman, S. B.: “C++ Primer 2nd Edition”, Addison Wesley (1991).

[4] 佐々 政孝: “プログラミング言語処理系”, 岩波 (1989).

[5] 寺田 実: “ごみ集めの研究動向”, 情報処理学会誌, Vol. 35, No. 11, pp. 1000 – 1005 (1994).