ICOT Technical Report: TR-0870

TR-0870

スタック領域が不要な深さ優先順コピー型ゴミ集め方式

中島　浩（京都大学）、近山　隆

April, 1994

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191～5
Institute for New Generation Computer Technology

スタック領域が不要な深さ優先順コピー型ゴミ集め方式

中島 浩 (京都大学工学部)　　近山 隆 (ICOT)

概要

代表的なゴミ集め方式のひとつである，コピー型のゴミ集めを改良した二つの方式を提案する。従来の方式が幅優先順にコピーを行なうのに対し，提案する方式はいずれも深さ優先順にコピーを行なう。この改良によって，ゴミ集め中およびゴミ集め後のメモリ・アクセスの局所性を大幅に改善できる。またデータ・オブジェクトだけを見て，その内容がポインタか否かを区別できる必要がないので，データ構造の設計に関する自由度が高い。

深さ優先順にコピーを行なうためには，未処理の要素を持つデータ構造を何らかの方法で記憶しておく必要がある。提案する方式の一つであるリンク法では，コピー前の領域に存在する未処理要素を持つデータ構造をリンクで結ぶことにより記憶する。またもう一つの方式である予約スタック法では，データ構造を指示するような未処理要素をコピー先領域の末端に配置されたスタックに記憶する。従って，いずれの方式においてもスタック領域を別途用意する必要はなく，従来の方式と同じ大きさのメモリ空間しか使用しない。

これら二つの方式と従来の方式の性能を評価し比較した所，ゴミ集めの対象となる領域が大きく，例えば実記憶容量を越えているような場合には，提案する方式ではページ・フォルトの回数が大幅に削減されることが明らかになった。

Depth-First Copying Garbage Collection without Extra Stack Space

Hiroshi Nakashima (Kyoto University)　　Takashi Chikayama(ICOT)

Abstract

In this papar, we propose two copying garbage collection methods. These two methods copy data structures in depth-first order, while conventional method copies in breadth-first order. This modification greatly improves memory access locality during both garbage collection and computation after. Restriction on data structure representation is also relaxed because the proposed methods don't require that a data object itself indicates whether it is a pointer.

For the depth-first copying, it is necessary to memorize information about non-leaf data structures so that their unprocessed elements are visited afterward. The first proposed method, the *link* method, links data structures which have unprocessed elements. Another method, the *reservation stack* method, pushes each unprocessed element being a pointer to another structure to a stack allocated at the end of the area to which data objects are copied. Therefore, neither of methods requires extra stack space.

We evaluated the performance of the proposed methods together with the coventional method. The result shows that the proposed methods have a great advantage over the conventional method when data objects are distributed in a large memory space, because the number of page faults are drastically reduced.

1 はじめに

自動メモリ領域管理機構は，ソフトウェア作成者からメモリ領域管理の負担を除き，より本質的な問題解決に専念させるために，殊に記号処理向きの言語の処理系では不可欠な機能となっている。

不要になったデータの占めるメモリ領域を自動的に再利用するゴミ集めには，さまざまな方式が提案され実際に用いられてきたが，代表的な方式の一つにコピー型のゴミ集め方式がある。これは，二つのメモリ領域を持ち，必要なデータだけを領域間でコピーすることによって，コピーされなかった不要なデータの占める領域を含め，コピー元の領域全体を再利用する方式である。この方式は不要なデータの占めるメモリ領域に一切アクセスしないため，必要なデータの割合が少ない場合に特に効率が良い。また，コピーの結果必要なデータがメモリ領域の一端に集まるので，連続した空き領域を提供でき，後のデータ割り付けが容易になり，ワーキングセットも小さくなるという長所を持つ。実装が比較的容易であることも大きな利点である。これらの長所は，メモリ領域が二つ必要なのでアドレス空間を2倍必要とするという欠点を充分に補うものであるため，最近の言語処理系ではコピー型の方式が多用されている。

さて，従来のコピー型ゴミ集め方式，即ち Fenichel と Yochelson によって最初に提唱された方式 [1] や，それを元に変形した諸方式 [2, 3] では，ネストしたデータ構造の領域間でのコピーが幅優先順に行なわれ，コピー後のデータも幅優先順に並ぶことになる。ところが，再帰的にデータ構造を処理するプログラムでは，データ構造を深さ優先順に扱うのが普通である。このため，データ構造はその作成時にはメモリ中に深さ優先順に並ぶように割り付けられ，後の利用でもこの順にアクセスされることが多い。このように深さ優先順に並んだデータ構造に対してゴミ集め時に幅優先順のコピーを行なうと，コピー時のワーキングセットが大きくなる。またコピー後の実行でも幅優先順に並んだデータを深さ優先順に用いることになるので，ワーキングセットが大きくなってしまう。

この問題を解決するために，本稿では深さ優先順にコピーを行なう二つの方式，リンク法と予約スタック法を提案する。深さ優先順にコピーを行なうためには，未処理の要素を持つデータ構造を何らかの方法で記憶しておく必要があるが，リンク法ではこのようなデータ構造をリンクで結ぶことにより記憶する。またもう一つの方式である予約スタック法では，データ構造を指示するような未処理要素をコピー先領域の末端に配置されたスタックに記憶する。従って，いずれの方式においてもスタック領域を別途用意する必要はなく，メモリ量やゴミ集めに要する手間のオーダは従来の方式と等しい。また，ゴミ集めのための特殊なデータ・タグを設けたり，各メモリ語に余分なビットを付加する必要もない。更に，提案する方式では常にデータ構造を指すポインタを用いながらコピーを行なうので，データ・オブジェクトだけを見て，その内容がポインタか否かを区別できる必要がない。

以下本稿では，まず2章において従来のコピー型ゴミ集め方式とその長所短所について簡単に述べる。次に3章では，提案する二つの幅優先順ゴミ集め方式について説明し，4章ではこれらの方式と従来の方式の性能比較を行なう。最後に5章では本稿の結論を述べる。

2 従来のコピー型ゴミ集め方式

提案する方式の説明の前提を与えるため，本章ではまず従来のコピー型ゴミ集め方式について概説する。

2.1 ゴミ集めの手順

コピー型ゴミ集め方式の基本的な手順は以下の通りである。

(1) 同じ大きさのメモリ領域をふたつ用意する。

(2) 通常のメモリ割り付けにはその一方だけを用いる。

(3) 使用中のメモリ領域（以下この領域を**旧領域**と呼ぶ）がいっぱいになったら，必要なデータだけをもう一方の領域 (**新領域**) にコピーする。不要なデータはコピーしないので，コピー後の領域には空きができる。

(4) 以降，コピーした先の領域をメモリ割り付けに用いる。

本稿で提案する方式も，このレベルでの手順は従来のコピー型ゴミ集め方式とまったく異ならない。違いはステップ3のコピーを行なう方式の詳細だけである。

従来の方式では，付録A.1に示した手順にしたがってコピーを行なう。なお，この手順で用いているデータ型，変数，手続き／関数は，以下のように定義されているものとする。

- *mem_word* ..... メモリ語を表すデータ型。ポインタ語に関してはポインタの型を示すタグと，指示するメモリ語のアドレスからなる。
- *mem_addr* ..... メモリ語のアドレスを表すデータ型。
- *new_base* ..... 新領域の先頭アドレスを保持する変数。
- *set_of_roots* ..... ルート（実行に必要と判っているポインタ群）のアドレスの集合。
- *ptag* ..... ポインタ型のタグを保持する変数。システムで用いるタグの中の任意のもので良い。
- *load*(a) ..... アドレス a の語の値を返す関数。
- *store*(a, w) ..... アドレス a に語 w を書き込む手続き。
- *tag*(w) ..... ポインタ語 w のタグを返す関数。
- *address*(w) ..... ポインタ語 w が指示する語のアドレスを返す関数。
- *make_word*(t, a) ..... タグが t，指示する語のアドレスが a であるようなポインタ語を返す関数。
- *is_pointer*(w) ..... 語 w がポインタであれば *true* を返す関数。
- *copied*(w) ..... ポインタ語 w が指示するデータ構造がコピー済であれば *true* を返す関数。付録A.1の場合は，データ構造の先頭語が新領域を指示するポインタ語であれば *true* を返す。
- *object_size*(w) ..... ポインタ語 w が指示するデータ構造の語数を返す関数。
- *copy_words*(ao, n, an) .... 旧領域アドレス ao から n 語を新領域アドレス an にコピーする手続き。

またデータ領域やデータ構造はアドレスが小さい方から大きい方へ向かって連続して伸びることを前提とし，その最も小さいアドレスの語を**先頭**，最も大きいアドレスの語を**末尾**と呼ぶ。

さて，この手順では，まず全てのルートから直接指されているデータを新領域にコピーし，続いてコピーされたデータから指されているデータをコピーする。実際にコピーを行なう手続き *copy_data* は，新領域（あるいはルート）のアドレス *new* の語がポインタ語であり，かつそれが指示しているデータ構造が未コピーであれば，データ構造の全体を新領域にコピーする。またコピー後，その先頭語をコピー先（新領域）へのポインタ語とすることによってコピー済を通知する。更に，コピー済／未コピーのいずれの場合にも，*new* のポインタ語はコピー先を指示するように書き換えられる。

新領域にコピーされたデータ構造から指されているデータのコピーは，*new* と新領域の末尾の次のアドレスである *new_tail* が等しくなるまで繰り返される。即ち新領域の *new* と *new_tail* の間は要素が処理されていないデータ構造を保持するFIFOキューであり，ネストしたデータ構造は幅優先順でコピーされる。このキューはデータ構造がコピーされるたびに大きくなるが，同じデータを二回コピーすることはないのでいつかは空になり（$new = new\_tail$），その時点で全てのデータがコピーされている。なお，*new* を順次増やしながら新領域をスキャンし，各々の語がポインタ語か否かを識別していることに注意が必要である。

この他，データ構造とそれを指すポインタ語を調べることにより，以下の二つが明らかになるというを前提を用いているが，これらは方式に関わらず必要なものであり，かつ妥当なものである。

(p1) データ構造の大きさ。
(p2) データ構造がコピー済であることを示す新領域へのポインタ語の位置。

なお (2) の語には，本来の値として新領域へのポインタ語と解釈される値が入っていてはならない。また前述の手順と以下の説明では，データ構造やポインタ語に関わらず先頭語を用いているが，一般にはどの語であっても良い。更に，複数語のデータ構造へのポインタ語はその先頭を指すことによってデータ構造の全体を指示すること，即ち；

(p3) データ構造の一部分のみを指すようなポインタは存在しない。

ことを前提としており，この制限は提案する方式についても同様である。この制限の緩和については紙数の都合で省略するが，[4] で論じているように

- 論理型言語の処理系で用いられる変数参照ポインタのような「透明な」ポインタの導入
- コピー済のデータ構造の先頭語を識別する特殊タグの導入

によって対処することができる。

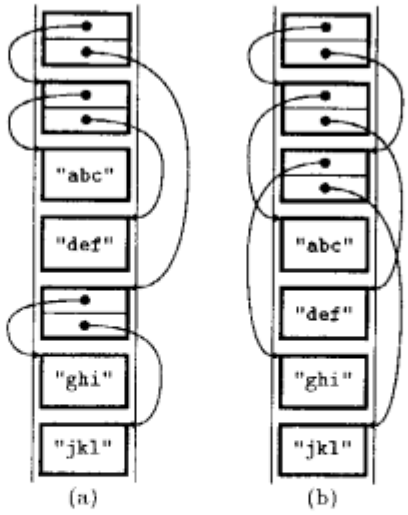

図 1: 従来方式での 2 進木のコピー

2.2 従来の方式の性質

従来のコピー型ゴミ集め方式は，以下の長所を持っている。

(+1) 不要なデータが占めるメモリ領域に一切アクセスしないので，必要なデータの割合が少ない場合に効率が良い。

(+2) ゴミ集めが必要なデータをメモリ領域の一端に集めるので；

- ゴミ集め後に空き領域が連続領域になり，データ割り付けが容易になる。
- コピー後のデータをアクセスする際に，データを移動しないゴミ集め方式に比べて，ワーキングセットが小さくなる。

(+3) 算法が簡潔で実装が比較的容易である。

一方，メモリ領域を二つ用いるために 2 倍のアドレス空間が必要とするというコピー型に共通の欠点があるほか，従来の方式に本質的な問題点として以下の二つが挙げられる。

(−1) ネストしたデータ構造については，幅優先順にコピーを行なうが，これは通常の再帰呼び出しなどによるプログラムのアクセス順である深さ優先順と一致していない。例えば Prolog において；

```
tree([L|R]):- left(L), right(R).
left(["abc" | "def"]).
right(["def" | "ghi"]).
```

によって四つの文字列を葉に持つ 2 進木構造；

```
[["abc" | "def"] | ["ghi" | "jkl"]]
```

を生成すると，図 1(a) のように配列する[1]。ところがこれを幅優先順にコピーすると，新領域では図 1(b) のような配列となる。このため，元のデータの並び順を保存する方式，例えばスライディング・コンパクション方式 [5] に比べると；

- 深さ優先順に作成したデータ構造を，ゴミ集めでコピーするとき
- ゴミ集めでコピーしたデータ構造を，深さ優先順にアクセスするとき

[1]多くの Prolog 処理系では"abc"などの文字列を文字コードのリストとして表現するが，ここでは文字列特有のデータ構造があるものとして説明する。

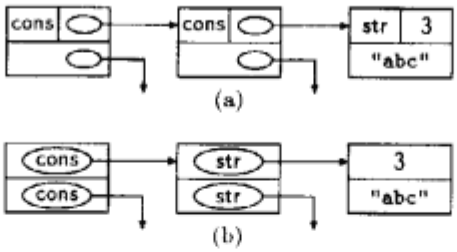

図 2: オブジェクト・タグとポインタ・タグ

に参照アドレスの局所性が低い。

(−2) 新領域をスキャンしながらコピーすべきデータ構造を見つけるため，ポインタとそうでないデータを識別するようなデータ表現が必要である。即ち，ある語がポインタ語か否かがデータ構造自体を調べるだけで明らかになる必要がある。この条件はデータ表現形式の設計に制約を加えるものであり，ゴミ集めの際に知る必要があるデータ構造の性質を，それを指すポインタのみが記述するような設計ができない。

例えば前述の 2 進木の例では，"abc"などの文字列をその長さとベタ詰めの文字コードの並びで表現しようとすると，それを誤ってポインタと解釈しないようにするための措置が必要となる。その一つとして図 2(a) のように，cons セルや文字列の先頭にデータ構造の型を示すタグを付け，cons セルでは 2 つのポインタを処理し，文字列では長さ分のデータをスキップする方法がある。この方法でのタグは，付加された語が属するオブジェクトの性質を示すためオブジェクト・タグと呼ばれる。しかしオブジェクト・タグは型識別のためのオブジェクトの参照，即ちデータ型を知るためにポインタが指示するメモリ語を読む必要がある。従って，図 2(b) のようにタグをポインタに付け，ポインタが指すオブジェクトの性質を示すようにするポインタ・タグに比べて一般に効率が悪い。

3 深さ優先順のコピー型ゴミ集め方式

従来の方式が持つ二つの問題点を解決するためには，深さ優先順にコピーすることと，新領域の単純なスキャンを不要とすることの，二つの改善が必要である。この二つは，ネストしたデータ構造のコピーのために新領域を FIFO キューとして用いることに密接に関連している。従って別途 LIFO スタックを設け，ネストしたデータ構造を深さ優先順にコピーすることにすれば，二つの改善を同時に実現できる。

例えば，複数の要素を持ち，かつ各々が他のデータ構造へのポインタでありうるようなデータ構造（以下**非終端構造**と呼ぶ）S の要素 e が，他の非終端構造 T を指すポインタである時，以下の情報をスタックに退避した後で T の処理を行なえば良い。

- S を指示するポインタ語。
- e の次の要素を知るための情報（S の先頭からのオフセットなど）。

スタックは T の処理が完了した時点でポップされ，e の次の要素から S の処理が再開される。その際，S の大きさやコピー先はスタックに退避した S へのポインタ語から求めることができる。またスタックが空であることは，一つのルートに関する処理が完了したことを意味する。

この方法（単純スタック法と呼ぶ）に基づくコピーの手順は，例えば付録 A.2に示すものとなる。なお説明を簡単にするために，データ構造の表現形式を以下のように仮定している。

(p4) データ構造は先頭に連続した非ポインタ語からなるヘッダを持つことができ，その大きさはデータ構造とそれを指すポインタ語を調べることにより明らかになる。関数 $copy\_header(w, a)$ は，ポインタ語 w が指示するデータ構造のヘッダを新領域アドレス a にコピーし，ヘッダの大きさを返す。

(p5) データ構造のヘッダを除いた各語はポインタ語である可能性があり，ポインタ語か否かはその語のみを調べれば明らかになる。

但しこれらの前提は単純スタック法にとって本質的ではなく，データ構造 S の語 w がポインタ語か否かを，S へのポインタと S の先頭からの w のオフセットから知ることができれば充分である。

さてこの手順では，非終端構造 S の末尾要素ではない要素 e_i を処理しているとき，処理の文脈は S を指すポインタ語 *pword* と，S の先頭から e_i へのオフセット *offset* が保持している。また補助的な文脈として S の先頭語の旧領域／新領域アドレスが *old_head* と *new_head* に，e_i を含めた未処理要素の数が n にそれぞれ保持されている。e_i が未コピーの非終端構造 T へのポインタ語であると，T の処理完了後に S の処理を e_i の次の要素から再開するために，手続き *push_stack* により文脈 *pword* と *offset* + 1 がスタックにプッシュされる。続いて補助的なものも含めた文脈が T の先頭要素のために設定され，先頭要素の処理を開始する。

一方，要素 e_i が非ポインタ語，コピー済またはコピー中のデータ構造へのポインタ，あるいは非ポインタ語のみからなるデータ構造（ヘッダのみの構造）であるときには，手続き *next_element* により次要素 e_{i+1} の処理に移る。但しこの時点でスタックが空であれば，コピー中の非終端構造は存在しないので，一つのルートに関する処理が完了したことになる。また e_{i+1} が末尾要素であると S の処理に戻る必要はないので，スタックをポップして S の「親」である非終端構造の文脈を補助的なものを含めて復元する。この一種の tail recursion 操作は要素数が 1 のデータ構造の処理にも適用され，文脈の退避／復元は行なわない。

さて単純スタック法の問題点は，スタックの深さが最悪の場合には旧領域と同じサイズになり，コピー型の欠点である大きなアドレス空間を更に拡大してしまうことにある。そこで，必要なメモリ領域は従来の方式と同一に保ったまま，深さ優先順にコピーする方式を二つ提案する。

3.1 リンク法

データ構造の表現形式に関する前述の仮定 (p4) と (p5) が成り立つ場合，非終端構造 S の要素 e_i の処理文脈として以下を用いることができる。

(1) e_i の旧領域アドレス。

(2) e_i の新領域アドレス。

(3) 未処理要素数

リンク法では，これらの文脈を図3に示すようにコピー中の非終端構造をリンクで結ぶ形で保存し，スタックを用いずに深さ優先順のコピーを行なう。即ち付録 A.3の手順に示す手続き *push_stack* では，e_i が非終端構造 T へのポインタである時，e_i の旧領域アドレス *old_link* を T の末尾要素に保存し，末尾要素自身は T の新領域の末尾に退避する。また e_i の値自身の参照が完了していることを利用し，e_i の新領域アドレス *new_link* は旧領域の e_i に保存される。

未処理要素数については，等価な情報として S の末尾要素が「親」の非終端構造でへのリンクであることを利用する。即ち，このリンクと他の要素の値とを区別できるようにし，これをセンチネルとして用いて一つの非終端構造の処理が完了したことを知る。この区別のために特殊なタグや余分なビットを用いなくても済むように，本来は旧領域のアドレスである *old_link* を，以下の式を用いて新領域アドレス *old_link'* に変換し，ポインタであることを示す適当なタグを付加して退避する。

$$old\_link' = old\_link - old\_base + new\_base$$

但し *old_base* は旧領域のベースであり，付録A.3では関数 *old_to_new* がこの変換を行なうこととしている。この変換によって，最終要素の値は新領域へのポインタに見えるが，前提 (p5) と 2.1で述べた前提 (p3) により，このようなポインタは末尾要素ではない未処理の要素の値として出現することはない。従って手続き *next_element* で用いている関数 $last\_word(w)$ は，w が新領域へのポインタ語であれば *true* を返すものであれば良い。

なお空のスタック，即ち非終端構造の連鎖の末端を示すために *old_link* を空ポインタ *NULL* に初期化しているが，この値は旧領域中に出現しないものであれば何でも良く，例えば絶対に使用されないアドレスを用いることができる。但しその場合には *new_link* の退避と復元，及び末尾要素の判定において，空ポインタに関する特別な操作が必要となる。一方，新旧各々の領域に 1 語ずつ，前述の変換式で相互に変換できるようなアドレスを確保し，この使用しないアドレスを空ポインタに用いれば，これらの特別な操作は不要となる。

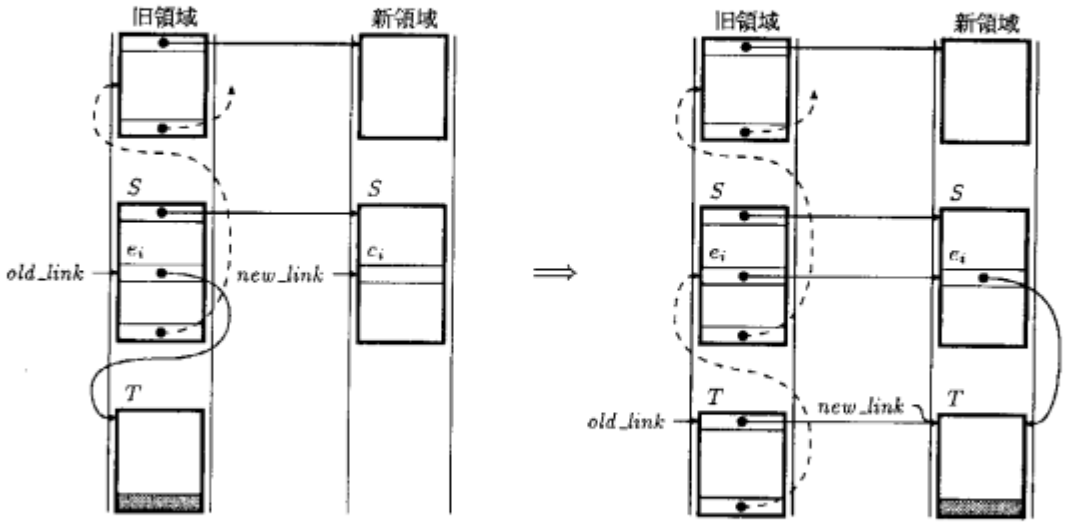

図 3: 非終端構造のリンク

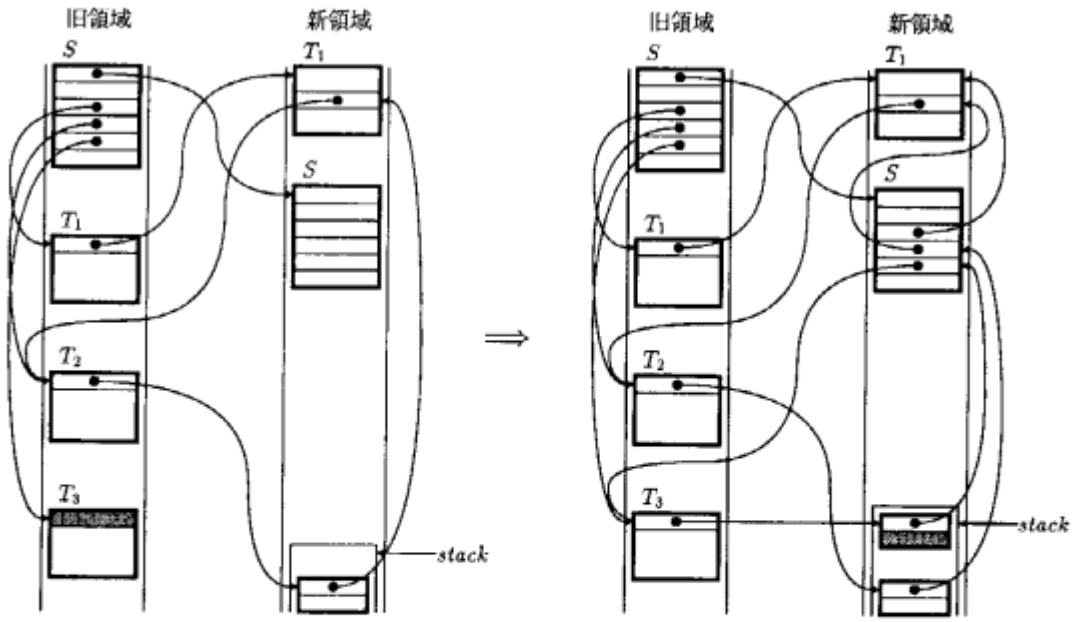

図 4: 予約スタック

3.2 予約スタック法

前述の単純スタック法では，非終端構造Sの要素として非終端構造Tが出現した時に，Sの処理を中断してTの処理を行ない，スタックにはSの処理に復帰するための情報を退避した。一方，提案する予約スタック法では，Sの処理を中断せずにTを処理するための情報をスタックに退避し，Sの処理完了後にTを処理する。この『予約スタック』は新領域のベースと反対側の末端に設け，コピーによって新領域の末尾*new_tail*が伸びる方向と逆の方向に伸びるようにし，余分なスタック領域を使用しないで済ませることができる。付録A.4に示す手順では，スタックの先頭である*stack*を新領域のベース*new_base*とその大きさ*new_size*の和に初期化して，スタックを新領域の末端に設定している。

予約スタックに退避する情報が，非終端構造一つ当たり2語以下であり，かつ退避されるのが一度だけであれば，新領域の末尾と予約スタックが衝突しないことを保証できる。即ち，Tの情報をスタックに退避する時点では，Tのコピー先は確保されていないが，Tがその後にコピーされることは確定している。またTは非終端構造であるので，明らかに2語以上の領域を占めている。従って，既にコピーされたデータ構造の語数とスタックの深さの和は，最終的にコピーされるべき全てのデータ構造の語数よりも小さく，この値は新領域の大きさよりも当然小さい。

さて，Tに関して予約スタックに退避する情報の一つとして，Tを指しているポインタtのコピー先アドレスがある。即ち，予約スタックへの退避の時点ではTのコピー先は確定していないため，後にTを処理する際にtがコピー先を指すように修正できなければならない。なお，T自身のアドレスはtが保持しているので，予約スタックから間接的に知ることができ，退避する必要はない。

一方，Tに関する情報が一度だけしか予約スタックに退避されないようにするためには，Tが「予約済」であることを何らかの方法で通知する必要がある。これを特殊なタグや余分なビットを用いずに行なうために，Tの先頭語を退避先のアドレスに変更し，かつその元の値を予約スタックに退避する。予約スタックは新領域に存在するので，非終端構造の先頭語の値によって，未コピー，コピー済，コピー予約済のいずれであるかを，以下のように判定することができる（図4）。

- 新領域へのポインタでなければ未コピー (T_3)
- 新領域へのポインタであるが，予約スタックへのポインタでなければコピー済 (T_1, $copied(w)$)
- 予約スタックへのポインタであれば，コピー予約済 (T_2, $reserved(w)$)

また，非終端構造Tがポインタtの処理によってコピー予約され，その後Tを指す別のポインタt'が見つかった場合，Tが処理される時にtとt'の双方が修正されるようにしなければならない。そこで，図4に示すように；

$$stack \rightarrow t' \rightarrow t \rightarrow T$$

の連鎖を形成し，Tの処理時に連鎖中の全てのポインタを修正できるようにする。ポインタの修正は手続き*pop_stack*の中で行なわれ，連鎖の末端は旧領域へのポインタの出現 ($old\_area(a)$) によって判別している。即ち，ポインタは新領域に存在するかルートであるので，旧領域に存在するTへのポインタによって，連鎖の末端を知ることができる。

以上をまとめると，予約スタックに退避すべき情報は，非終端構造を指すポインタのアドレスと非終端構造の先頭語の2語となり，前述の条件を満たす。即ち，1語のデータ構造を指すポインタについては，スタックを用いずに直接ポインタの先を処理してるので，予約スタックには2語以上のデータ構造に関する情報のみが退避される。なお，付録A.4の手順では前述の仮定 (p4) と (p5) を用いているが，これらの仮定は予約スタック法にとって本質的ではない。また，説明を簡単にするために非終端構造の処理を先頭要素から正順に行なっているが，コピーの順序を左優先にするために末尾から逆順に行なうこともできる。

4 性能評価

本章では従来の方式と提案した二つの方式の性能を，メモリのアクセス回数，小～中規模のメモリ領域を対象とした時の実行時間，及び大規模なメモリ領域を対象とした時のページ・フォルトの回数の，三つの指標に基づいて比較する。

4.1 メモリ・アクセス回数

従来の方式では，データ構造を旧領域から新領域へコピーした後で，新領域をスキャンしてその要素を処理する。従ってデータ構造中のポインタとなりうる要素の各々について3回のメモリ・アクセス，即ちコピーのための旧領域での読出と新領域への書込，及びスキャンによる新領域での読出が行なわれる。一方，提案した二つの方式はいずれも新領域のスキャンを行なわないので，要素ごとのメモリアクセス回数は旧領域での読出と新領域への書込の2回で済む。従って少なくとも要素数が充分大きければ，提案した方式の方が従来の方式よりも高速であると言うことができる。

実際にはコピー済通知などのために，要素数によらずデータ構造ごとに一定回数のメモリ・アクセスを必要とし，この回数は方式によって異なる。例えば，ただ一つのポインタから指され，かつポインタ語を2語以上含むようなデータ構造に関して，各々の方式における定数回のメモリ・アクセスは以下のようになる。

従来の方式 ＝2回

- コピー済通知のための先頭語の書込（1回）
- データ構造を指すポインタの修正（1回）

リンク法 ＝7回

- コピー済通知のための先頭語の書込（1回）
- 末尾語の退避のための読出／書込（各1回）
- *old_link* の退避／復元のための書込／読出（各1回）
- *new_link* の退避／復元のための書込／読出（各1回）

スタック法 ＝8回

- コピー済通知のための先頭語の書込（1回）
- コピー済予約済み通知のための先頭語の書込（1回）
- スタックの書込／読出（各2回）
- データ構造を指すポインタ修正のための読出／書込（各1回）

従って，全ての語がポインタ語でありうるようなn語のデータ構造の場合，各方式でのメモリ・アクセス回数は；

$$\begin{aligned} \text{従来の方式} &= 3n+2 \\ \text{リンク法} &= 2n+7 \\ \text{予約スタック法} &= 2n+8 \end{aligned}$$

となり，従来方式に比べて$n>5$でリンク法が，また$n>6$でスタック法が，それぞれメモリ・アクセス回数が少なくなる。

4.2 実行時間

ゴミ集めに要する時間を決定する要因として，前節で述べたメモリ・アクセス回数の他に，処理の複雑さとメモリ・アクセスの局所性などが挙げられる。提案した二方式は従来の方式に比べ，処理の複雑さの面では劣り，メモリ・アクセスの局所性の面では優るものと考えられる。これらを総合的に評価するため，各方式に基づくゴミ集めのプログラムを作成し，実行時間を測定した。

測定に用いたデータは$n=2^k$のバランスしたn進木である。木の各ノードであるn要素のデータ構造は$n+1$語からなり，先頭語に要素数nが，引き続くn語に各要素がそれぞれ格納される。この先頭語のコピーのためのメモリ・アクセス回数の増加は従来法で3，リンク法とスタック法では2である。但しLispやPrologなどの言語では，$n=2$のconsセルを2語で実装する方式が良く用いられるので，それに準拠した表現とした。即ちconsセルに限り，データ構造を指すポインタを他のものと区別し，要素数を省略している。また木の葉には非ポインタ語が直接格納されている。1語は32ビットであり，下位2ビットがデータ型を示すタグ，上位30ビットがアドレスなどに用いられる。

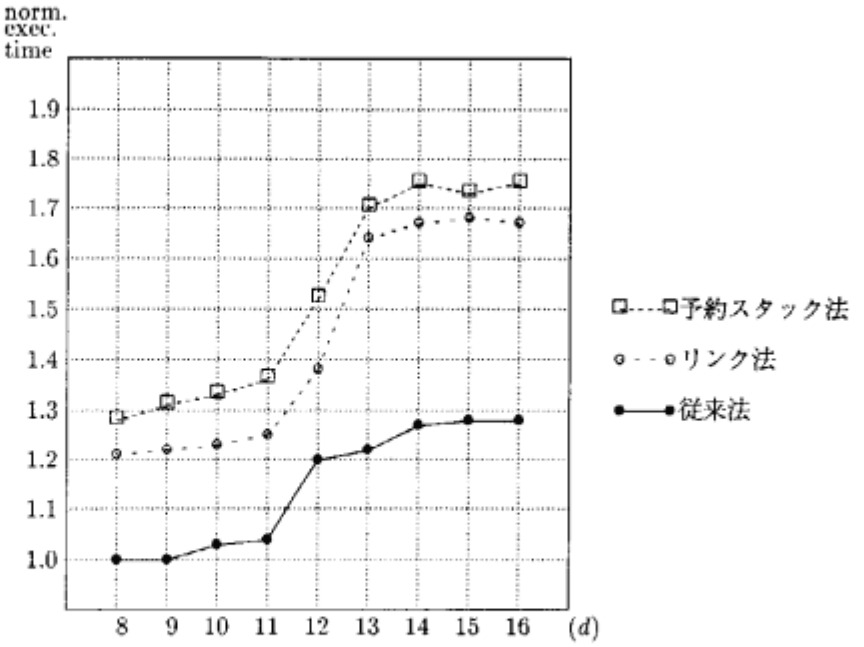

図 5: 2 進木の場合の実行時間

ゴミ集めの方法は基本的に 3 章で述べたものに基づいているが，リンク法と予約スタック法に関しては，tail recursion 最適化の技法を用いて不要なメモリ・アクセスを削除している。その結果評価に用いた n 進木ではメモリ・アクセス回数が若干減少し，$n > 2$ の場合のノードあたりの平均アクセス回数は；

$$\begin{aligned} \text{リンク法} &= 2n + 8 + (n-1)/n \\ \text{予約スタック法} &= 2n + 2 + 8(n-1)/n \end{aligned}$$

となる。また，cons セルの場合にはリンク法が 9, 予約スタック法が 8 となる。更に，予約スタック法は左優先にコピーするために要素の処理を逆順に行なっている。

また，プログラムにはゴミ集めを行なう部分の他に n 進木を操作する部分があり，木を左優先／深さ優先に探索しながら，木のコピーを生成する。例えば 2 進木の場合，以下の Prolog プログラムと等価な操作を行なう。

```
traverse([L1|R1],[L2|R2]):- !,
    traverse(L1,L2), traverse(R1,R2).
traverse(T,T).
```

実行時間の測定はこの操作も含めて行なった。

プログラムは C 言語によって作成し，主記憶容量 16 MB，キャッシュ容量 64 KB の SparcStation1 で実行した。またコンパイラは gcc 2.4.5 版を用い，コンパイル・オプションは "`-O2 -fomit-frame-pointer`" である。

図 5は，深さ d が 8 から 16 の 2 進木の操作とゴミ集めに要した時間を木の語数で割り，それを更に深さ 8 の時の従来方式の実行時間を 1 として正規化したものである。なお深さを 17 以上にすると特に従来方式でページ・フォルトが頻発し，有意な測定ができなかった。提案した手法は従来方式に比べ実行時間が長くなっているが，d が小さい時にはリンク法で約 20 %，予約スタック法で約 30 %の性能低下に留まっている。メモリ・アクセス回数は予約スタック法では従来法と同じであり，またリンク法でもノードあたり 1 回 (12.5 %) 多いだけであるので，実行時間差の多くの部分は処理の複雑さによるものである。またいずれの方式でも深さが 12 になると急に実行時間が増加するが，これはデータがキャッシュからあふれることが原因である。従来法の方が増加の度合が少ない理由は，新領域のスキャンの際にキャッシュのプリフェッチ効果が現れ，キャッシュ・ミスの頻度が少なくなることであると考えられる。

一方，図 6は n を変化させた場合の性能であり，d はページ・フォルトが頻発して測定不能になる直前まで大きくした。なお，$n = 2$, $d = 16$ の時の従来法の実行時間を 1 として正規化してある。図から明らかな

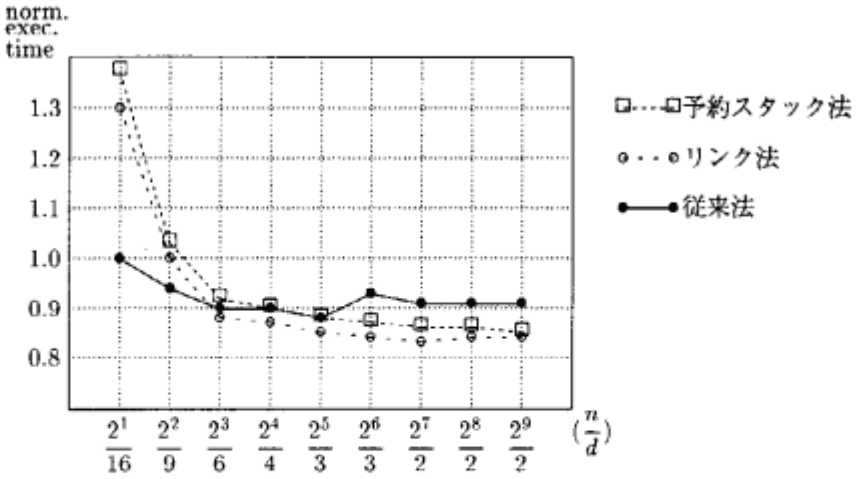

図 6: n 進木の場合の実行時間

表 1: 余分なページ・フォルトを起こさないための物理ページ数

従来法	リンク法	予約スタック法
3585	1026	2050

ように，n が増加するに従ってリンク法や予約スタック法の性能が相対的に良くなり，従来法と逆転する。この理由は前述のメモリ・アクセス回数の差が主なものであるが，リンク法や予約スタック法でもキャッシュのプリフェッチ効果が，木の葉の部分のコピーの際に現れることにも関係していると考えられる。

4.3 ページ・フォルトの回数

従来法の問題点であるメモリ・アクセス局所性の悪さは，ページ・フォルトが増加する形で顕在化するものと考えられる。即ち，幅優先で旧領域をアクセスするために，連続してアクセスする二つのデータ構造間のアドレス差が大きく，これらが同じメモリ・ページに存在する確率が低くなることが予想される。

そこで，各方式について深さが 20 の 2 進木（約 8 MB）を対象として，ページ・フォルトがどのように発生するかを評価した。但しこの処理を実際の計算機上で行なうと，ページ・フォルトの頻発のために有意な測定ができないので，アクセスするメモリ・アドレスのトレースを生成してページ・フォルトの発生状況をシミュレートした。なお，メモリ・ページの大きさは 4 KB とし，ゴミ集め開始時にはアクセス対象の全てのページがスワップ・アウトされているものと仮定した。ゴミ集めでは新旧各々の領域について 8 MB = 2048 ページをアクセスするため，4096 回のページ・フォルトが必然的に発生する。そこで，これを基準として更に余分なページ・フォルトがどれだけ発生するかを，物理ページ数を変化させながら測定した。

まず表 1は，余分なページ・フォルトが全く発生しないようにするために必要な物理ページ数を示したものである。表から明らかなように，リンク法ではアクセスするページの約 1/4，また予約スタック法では約 1/2 の物理ページがあれば，余分なページフォルトは全く発生しない。これに対して従来法では約 7/8 の物理ページが必要であり，アクセス局所性が悪いことが明らかになった。また，ゴミ集めの結果を深さ優先／左優先で探索する場合，リンク法や予約スタック法では 1 ページあれば余分なページ・フォルトは全く発生しないが，従来法では実に 2047 ページが必要であることも明らかになった。

次に，物理ページ数を表 1に示した値よりも少なくした時に，余分なページ・フォルトが何回発生するかを測定した。結果は図 7 に示す通りであり，リンク法や予約スタック法では物理ページ数の減少に対するページ・フォルトの増加は緩やかである。これに対し従来法では急激に増加し，アクセスするページ数の 1/2 である 2048 ページでも，実に約 18,000 回ものページ・フォルトが発生する。この原因は，ある一定の深さ以下では同じ深さにあるノードが全て別のページに存在し，アクセスするノードが変わるたびにページ・フォルトとなってしまうことにある。

以上のように従来法ではアクセスする領域が大きくなるとページ・フォルトが極めて高い頻度で発生す

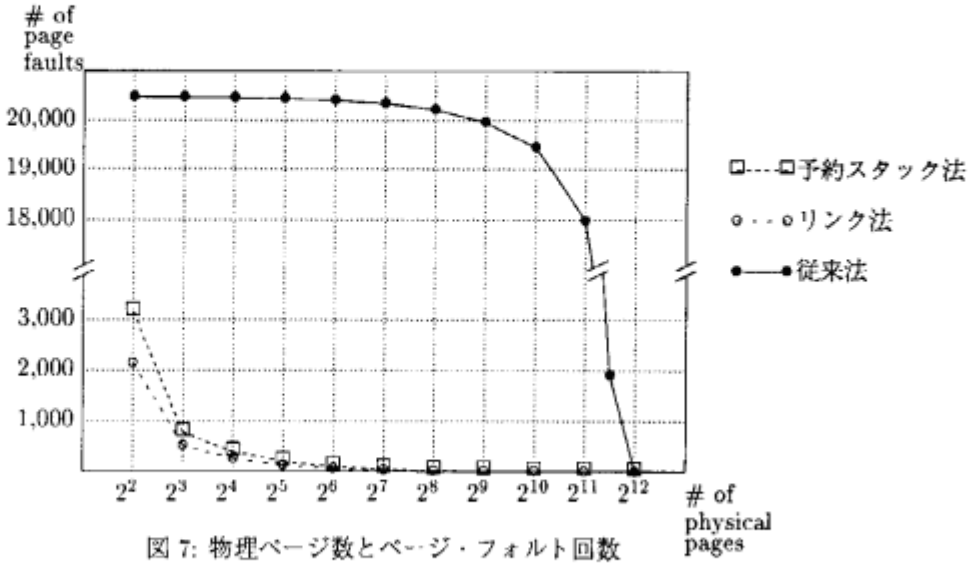

図 7: 物理ページ数とページ・フォルト回数

るため，実用的な時間でゴミ集めができないことが明らかになった。また，新旧領域の合計を物理メモリ容量以下にしたとしても，ページ・アクセス局所性の悪さは TLB のヒット率低下につながり，やはり大幅な性能低下を招くことは明らかである。

5 おわりに

本稿ではコピー型のゴミ集めを改良し，深さ優先順にコピーを行なう二つの方式を提案した。いずれの方式についても，不要なデータにアクセスしないことや，必要なデータがメモリ領域に集まるなど，従来方式の利点をそのまま受け継いでいる。また，ゴミ集めに要する手間のオーダや必要とするメモリ量も従来方式と等しく，特殊なデータ・タグや各メモリ語に余分なビットを必要することもない。

一方，従来方式が幅優先順にコピーをするために生じる参照アドレス局所性の低さは，深さ優先順にコピーすることにより解消しており，ページ・フォルト回数の劇的な減少と言う評価結果によって提案した方式の有効性を示すことができた。また，新領域のスキャンを必要としないため，ポインタ・タグを使用できるなど，従来方式に比べてデータ構造の設計に関する自由度が大幅に増したことも，提案した方式の大きな特質である。

参考文献

[1] Fenichel, R. and Yochelson, Y.: A Lisp Garbage Collector for Virtual Memory Computer Systems, *Communication of the ACM*, Vol. 12, No. 11, pp. 419–429 (1969).

[2] Baker, Jr., H. G.: List Processing in Real Time on a Serial Computer, *Communication of the ACM*, Vol. 21, No. 4, pp. 280–294 (1978).

[3] Unger, D.: *The Design and Evaluation of a High Performance Smalltalk System*, ACM Distinguished Dissertations, The MIT Press (1987).

[4] 中島浩, 近山隆: データ構造の一部を指すポインタを許容するコピー型ゴミ集め方式, Technical memorandum, ICOT (1994). (to be published.).

[5] Morris, F. L.: A Time and Space Efficient Garbage Collection Algorithm, *Communication of the ACM*, Vol. 21, No. 8, pp. 662–665 (1978).

付録

A.1 従来の方式の手順

```
procedure breadth_gc ;
  new_tail :  mem_addr ;  {(新領域の末尾)+1}
  new :       mem_addr ;  { 処理対象語の新領域アドレス }
  word :      mem_word ;  { 処理対象語の値 }
  old :       mem_addr ;  { ポインタ語が指す語の旧領域アドレス }
  size :      integer ;   { ポインタ語が指すデータ構造の語数 }
  procedure copy_data ;                                  { データのコピー }
  begin
    word ← load(new) ;
    if is_pointer(word) then begin                       { ポインタ語の処理 }
      old ← address(word) ;
      if copied(word) then                               { オブジェクトはコピー済 }
        store(new, make_word(tag(word), address(load(old)))) ;
      else begin
        store(new, make_word(tag(word), new_tail)) ;     { ポインタ語がコピー先を指すようにする }
        size ← object_size(word) ;
        copy_words(old, size, new_tail) ;                { オブジェクトを新領域にコピー }
        store(old, make_word(ptag, new_tail)) ;          { 先頭語にコピー済を通知 }
        new_tail ← new_tail + size ;                     { コピー先の確保 }
      end ;
    end ;
  end ;
begin
  new_tail ← new_base ;
  for ∀new ∈ set_of_roots do                             { 全てのルートが指すデータをコピー }
    copy_data ;
  new ← new_base ;
  while new < new_tail do begin                          { コピーされたデータが指すデータをコピー }
    copy_data ;
    new ← new + 1 ;
  end ;
end ;
```

A.2 単純スタック法の手順

```
procedure depth_gc ;
  ⟨ breadth_gc と同じ変数宣言 ⟩
  pword :       mem_word ;  { コピー中非終端構造へのポインタ語 }
  offset :      integer ;   { 処理対象要素のオフセット }
  old_head :    mem_addr ;  { コピー中非終端構造の先頭アドレス (旧領域)}
  new_head :    mem_addr ;  { コピー中非終端構造の先頭アドレス (新領域)}
  n :           integer ;   { 未処理語数 }
  hsize :       integer ;   { ヘッダの語数 }
  stack_empty : boolean ;   { スタックが空ならば true}
  procedure initialize_stack ;                           { スタックの初期化 }
  begin
    stack ← stack_base ;
```

```
        pword ← NULL ;
        offset ← 0 ;
    end ;
    procedure push_stack ;                          { スタックのプッシュ}
    begin
        store(stack, pword) ;                       { 非終端構造へのポインタ語を退避 }
        store(stack + 1, offset + 1) ;              { 次要素のオフセットを退避 }
        stack ← stack + 2 ;
        pword ← word ;                              { 非終端構造へのポインタ語と先頭アドレスを設定 }
        offset ← hsize ;
        old_head ← old ;
        new_head ← new − offset ;
        n ← size − offset ;                         { 未処理要素数を設定 }
    end ;
    procedure next_element ;                        { 次要素の処理 }
    begin
        if stack = stack_base then                  { スタックは空 }
            stack_empty ← true ;
        else begin                                  { コピー中の非終端構造あり }
            offset ← offset + 1 ;                   { 次要素の設定 }
            word ← load(old_head + offset) ;
            new ← new_head + offset ;
            n ← n − 1 ;                             { 未処理要素数を減らす }
            if n = 1 then begin                     { 末尾要素 }
                stack ← stack − 2 ;                 { スタックをポップ }
                pword ← load(stack) ;               { 非終端構造へのポインタ語を復元 }
                offset ← load(stack + 1) ;          { 要素のオフセットを復元 }
                if pword ≠ NULL then begin
                    old_head ← address(pword) ;     { 非終端構造の先頭アドレスを復元 }
                    new_head ← address(load(old_head)) ;
                    n ← object_size(pword) − offset ;   { 未処理要素数を復元 }
                end ;
            end ;
        end ;
    end ;
begin
    new_tail ← new_base ;
    initialize_stack ;                              { スタックを初期化 }
    for ∀new ∈ set_of_roots do begin                { 全てのルートを処理 }
        word ← load(new) ;                          { ルートを処理 }
        repeat begin
            stack_empty ← false ;
            if is_pointer(word) then begin          { ポインタ語の処理 }
                old ← address(word) ;
                if copied(word) then begin          { オブジェクトはコピー済 }
                    store(new, make_word(tag(word), address(load(old)))) ;
                    next_element ;
                end ;
                else begin
                    size ← object_size(word) ;
```

```
            store(new, make_word(tag(word), new_tail)) ;
                                                  { ポインタがコピー先を指すようにする }
            hsize ← copy_header(word, new_tail) ; { ヘッダをコピーし先頭要素のオフセットを得る }
            new ← new_tail + hsize ;              { 先頭要素のコピー先を設定 }
            word ← load(old + hsize) ;            { 先頭要素をロード }
            store(old, make_word(ptag, new_tail)) ; { 先頭語にコピー済を通知 }
            new_tail ← new_tail + size ;          { コピー先の確保 }
            if size = hsize then                  { 非ポインタ語のみからなるデータ構造 }
               next_element ;
            else if size - hsize > 1 then         { 非終端構造 }
               push_stack ;
         end ;                                    { 上記以外であれば 1 要素のデータ構造 }
      end ;
      else begin                                  { 非ポインタ語の処理 }
         store(new, word) ;
         next_element ;
      end ;
   end ;
   until stack_empty ;                            { スタックが空になるまで繰り返し }
end ;
end ;
```

A.3 リンク法の手順

```
procedure link_gc ;
   ( breadth_gc と同じ変数宣言 )
   old_link  :  mem_addr ;  { 旧領域へのリンク }
   new_link  :  mem_addr ;  { 新領域へのリンク }
   hsize     :  integer ;   { ヘッダの語数 }
   stack_empty : boolean ;  { スタックが空ならば true}
   procedure initialize_stack ;                   { スタックの初期化 }
   begin
      old_link ← NULL ;                           { リンクを空ポインタに初期化 }
   end ;
   procedure push_stack ;                         { スタックのプッシュ}
   begin
      store(new_tail - 1, load(old + size - 1)) ; { 末尾要素の退避 }
      store(old_link, new_link) ;                 { リンクの退避 }
      store(old + size - 1, make_word(ptag, old_to_new(old_link))) ;
      old_link ← old + hsize ;                    { リンクを先頭要素に設定 }
      new_link ← new ;
   end ;
   procedure next_element ;
   begin
      if old_link = NULL then                     { スタックは空 }
         stack_empty ← true ;
      else begin                                  { コピー中の構造体あり }
         old_link ← old_link + 1 ;                { リンクを次要素に設定 }
         new_link ← new_link + 1 ;
         new ← new_link ;                         { 次要素の設定 }
```

```
        word ← load(old_link) ;
        if last_word(word) then begin              { 末尾要素 }
            old_link ← new_to_old(address(word)) ; { 旧領域リンクの復元 }
            word ← load(new_link) ;                { 末尾要素の復元 }
            new_link ← load(old_link) ;            { 新領域リンクの復元 }
        end ;
    end ;
  end ;
begin
  ⟨ depth_gc と同じ手続き ⟩
end ;
```

A.4 予約スタック法の手順

```
procedure stack_gc ;
  ⟨ breadth_gc と同じ変数宣言 ⟩
  old_e :       integer ;  { 処理対象要素の旧領域アドレス }
  new_e :       integer ;  { 処理対象要素の旧領域アドレス }
  hsize :       integer ;  { ヘッダの語数 }
  n :           integer ;  { 未処理語数 }
  stack_empty : boolean ;  { スタックが空ならば true}
  procedure pop_stack ;
  begin
    new ← address(load(stack)) ;
    repeat begin                                  { 連鎖中の全てのポインタがコピー先を指すようにする }
      word ← load(new) ;
      store(new, make_word(tag(word), new_tail)) ;
      new ← address(word) ;
    end ;
    until old_area(new) ;                         { 旧領域アドレスが見つかるまで繰り返し }
    old ← new ;
    store(old, load(stack + 1)) ;                 { 先頭語を復元 }
    size ← object_size(word) ;
    hsize ← copy_header(word, new_tail) ;         { ヘッダをコピーし先頭要素のオフセットを得る }
    new ← new_tail + hsize ; new_e ← new ;        { 先頭要素のコピー先を設定 }
    old_e ← old + hsize ;
    word ← load(old_e) ;                          { 先頭要素をロード }
    store(old, make_word(ptag, new_tail)) ;       { 先頭語にコピー済を通知 }
    new_tail ← new_tail + size ;                  { コピー先の確保 }
    n ← size − hsize ;                            { 未処理要素数を設定 }
    stack ← stack + 2 ;                           { スタックをポップ }
  end ;
  procedure next_element ;
  begin
    n ← n − 1 ;                                   { 未処理要素数を減らす }
    if n > 0 then begin                           { 未処理要素あり }
      old_e ← old_e + 1 ;                         { 次の要素を処理 }
      new_e ← new_e + 1 ; new ← new_e ;
      word ← load(old_e) ;
    end ;
```

```
        else if stack = new_base + new_size                    { スタックは空 }
            stack_empty ← true ;
        else
            pop_stack ;                                        { スタックをポップ }
    end ;
begin
    new_tail ← new_base ;
    stack ← new_base + new_size ;                              { スタックを初期化 }
    for ∀new ∈ set_of_roots do begin
        word ← load(new) ;                                     { ルートを処理 }
        n ← 1 ;
        repeat begin
            stack_empty ← false ;
            if is_pointer(word) then begin                     { ポインタ語の処理 }
                old ← address(word) ;
                if copied(word) then begin                     { オブジェクトはコピー済 }
                    store(new, make_word(tag(word), address(load(old)))) ;
                    next_element ;
                end ;
                else if reserved(word) then begin              { オブジェクトはコピー予約済 }
                    old ← address(load(old)) ;                 { スタックのアドレス }
                    store(new, make_word(tag(word), address(load(old)))) ;
                    store(old, make_word(ptag, new)) ;         { ポインタを連鎖の先頭に挿入 }
                    next_element ;
                end ;
                else if object_size(word) = 1 then begin       {1 語のオブジェクト }
                    store(new, make_word(tag(word), new_tail)) ; { ポインタがコピー先を指すようにする }
                    word ← load(old) ;                         { ポインタの先を続けて処理 }
                    store(old, make_word(ptag, new_tail)) ;
                    new ← new_tail ;
                    new_tail ← new_tail + 1 ;
                end ;
                else begin                                     {2 語以上のオブジェクト }
                    stack ← stack − 2 ;
                    store(stack, make_word(ptag, new)) ;       { ポインタのアドレスを退避 }
                    store(stack + 1, load(old)) ;              { 先頭語を退避 }
                    store(old, make_word(ptag, stack)) ;       { 先頭語にコピー予約済を通知 }
                    next_element ;
                end ;
            end ;
            else begin                                         { 非ポインタ語の処理 }
                store(new, word) ;
                next_element ;
            end ;
        end ;
        until stack_empty ;                                    { スタックが空になるまで繰り返し }
    end ;
end ;
```