ICOT Technical Memorandum: TM-1295

TM-1295

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

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

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

中島 浩 (京都大学工学部)

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において，データ構造の一部を指すようなポインタ（部分構造ポインタ）の取り扱いが問題となる。即ち，データ構造S_1が部分構造S_2を含むとき，S_1のコピー後にS_2へのポインタを発見した際には，S_2がコピーされることを防止しなければならない。また逆に，S_2のコピー後にS_1へのポインタを発見した際には，S_1とS_2のコピー先が等価になるように何らかの方法で関連づける必要がある。更に，互いに一部を共有するような二つの部分構造がある場合には，両者が反復してコピーされることを防止しなければならない。

本稿では，基本的なコピー型のゴミ集め方式と，二つの深さ優先順コピー型ゴミ集め方式であるリンク法と予約スタック法に関して，上記の部分構造に関する問題点の解決方法を示す。部分構造間の相互共有がない場合には，コピーされた二つのデータ構造が等価であることを示す「透明ポインタ」を追加すれば良い。また，論理型言語の処理系の変数参照ポインタのように，もともと透明ポインタを持つような場合には，ポインタ型の追加は不要である。

一方，部分構造間の相互共有がある場合には，二つのデータ構造を合体させる処理を行なえば良い。このためにはデータ構造の先頭とそれ以外を区別する必要があるが，リンク法以外ではこの区別に透明ポインタを利用することができる。

Copying Garbage Collection with Substructure Pointers

Hiroshi Nakashima (Kyoto University)

Abstract

In copying garbage collection, a substructure included in a larger structure causes troublesome problems. When a structure S_1 has a substructure S_2, and a pointer to S_2 is found after S_1 was copied, S_2 should not be copied to avoid duplication. Conversely, when a pointer to S_1 is found after S_2 was copied, S_1 and S_2 in copy destination area should be made equivalent. Furthermore, if two substructure shares a part of each other, it should be prevented that they are copied repeatedly.

In this paper, we show solutions to those problems for conventional copying garbage collection and novel depth first methods, *link* and *reservation stack* methods. If the mutual sharing by substructures is not allowed, *transparent* pointers to connect equivalent structures will solve the problem. If the system inherently has the transparent pointer, such as variable reference in logic programming language, it is not necessary to add new pointer type for transparent pointers.

On the other hand, if the mutual sharing is allowed, combining two substructures will solve the problem. Although this solution requires that the first word of a structure is distinguish from subsequent words, transparent pointer type will be utilized for the identification, except for the link method.

1 はじめに

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 [1, 2, 3]。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。

また著者らは従来型のコピー型ゴミ集め方式の欠点である幅優先順処理を改良し，スタック領域が不要な深さ優先順の方式を二種類提案した [4]。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。

これらのコピー型ゴミ集め方式では，いずれもデータ構造へのポインタはその先頭を指示し，かつその全体が有効なデータであることを前提としている。しかし，プログラミング言語やその処理系によっては，データ構造の一部を指すようなポインタが用いられることがある。例えば Prolog のような論理型言語の処理系では，変数参照のポインタがデータ構造の要素を直接指すようにすることが多い。またより一般的な例としては，データ構造の大きさを示す情報をポインタに持たせ，複数のデータ構造に領域を共有させるものもある [5]。

このような部分構造が存在すると，データ構造のコピーを注意深く行なわなければならない。即ち，データ構造 S_1 が部分構造 S_2 を含むとき，S_1 のコピー後に S_2 へのポインタを発見した際には，S_2 がコピーされることを防止しなければならない。また逆に，S_2 のコピー後に S_1 へのポインタを発見した際には，S_1 と S_2 のコピー先が等価になるように何らかの方法で関連づける必要がある。更に，互いに一部を共有するような二つの部分構造がある場合には，両者が反復してコピーされることを防止しなければならない。これらはコピー型に特有の問題であり，例えば mark and sweep のようにデータ構造のコピーを行なわない方式では部分構造の取り扱いは容易である。

そこで本稿では，基本的なコピー型のゴミ集め方式と，二つの深さ優先順コピー型ゴミ集め方式であるリンク法と予約スタック法に関して，上記の部分構造に関する問題点の解決方法を示す。

以下本稿では，まず 2 章において従来型と深さ優先順のコピー型ゴミ集め方式について述べる。次に 3 章では部分構造間の相互共有がない場合と，相互共有がある場合の各々について，問題の解決法を示す。

2 コピー型ゴミ集め方式

2.1 従来方式

従来のコピー型ゴミ集め方式では，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。

2.2 リンク法

深さ優先順にコピーを行なう場合，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，S の処理を中断して T を処理する。従って S の処理に戻るための処理文脈を保存する必要があるが，リンク法では以下のものを処理文脈として用いる。

(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

これらの文脈は図1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され，スタックを用いずに深さ優先順のコピーが行なわれる。即ち付録 A.2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.2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，コピー済あるいはコピー中の部分構造がなければ，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。

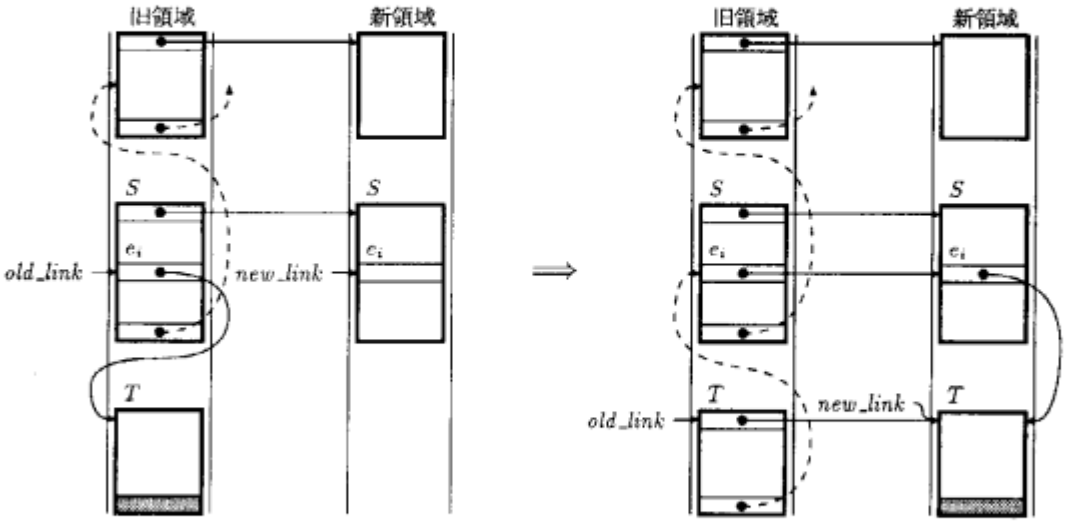

図 1: 非終端構造のリンク

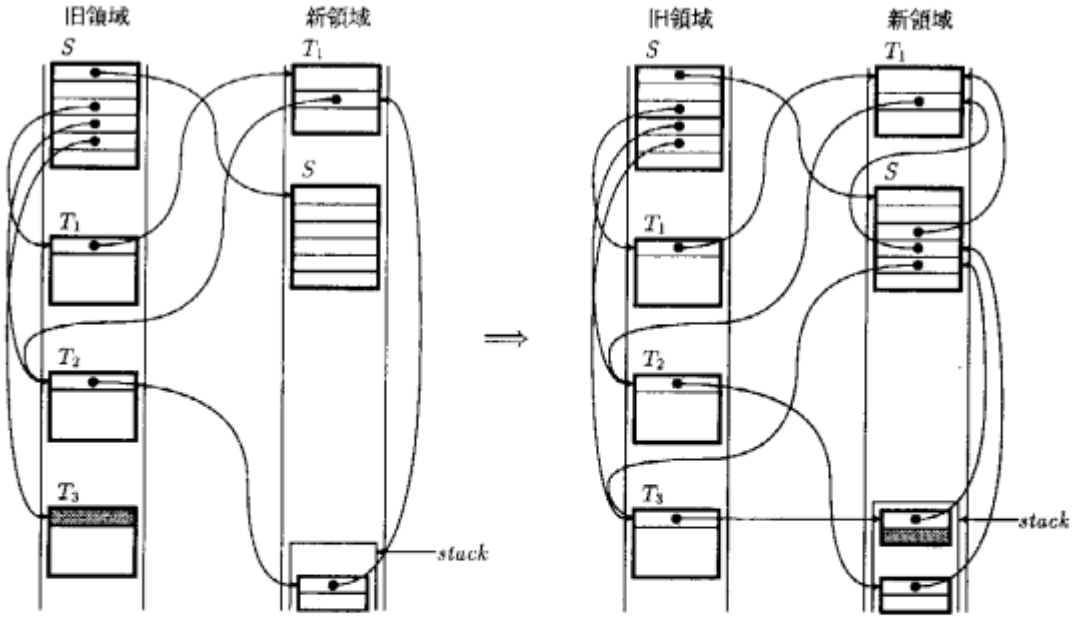

図 2: 予約スタック

2.3 予約スタック法

前述のリンク法では，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の処理を行なった。一方，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.3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。

さて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。そこで予約スタックに非終端構造Tを指すポインタのアドレスとT の先頭語を退避し，Tの先頭語は退避先のアドレスに変更する。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2)。

- 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2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。

3 部分構造の処理

3.1 基本的な問題とその解決法

データ構造の一部を指す部分構造ポインタが存在する場合，従来の方式と深さ優先順の二つの方式に共通して，以下のような処置が必要である。

(1) コピー済通知

図3(a)のように，データ構造S_1がコピーされた後でS_1に含まれる部分構造S_2を処理する場合，S_2はコピー済としなければならない。従ってコピー済の通知のためには，S_1の先頭語にコピー先へのポインタを記録するだけでは不十分であり，部分構造ポインタから指されうる全ての語に対応するコピー先へのポインタ記録しなければならない。またコピー済であることを知るには，そのような語が全て連続する新領域（コピー先）へのポインタであることを確認しなければならない。

(2) 部分構造のコピー

図3(b)のように，データ構造S_1に含まれる部分構造S_2がコピーされた後でS_1を処理する場合，S_1を新領域 (S_1') にコピーした上で，新領域でのS_2 (S_2') と新領域でのS_1'の中のS_2に対応する部分 (S_2'') とを，何らかの方法で関係付けなければならない。しかしS_2'を指しているポインタを見つけてそれらがS_2''を指すように変更するには，コピー済の全てのデータを調べる必要があり非効率である。そこで，論理型言語の処理系で用いられる変数参照ポインタや，関数型言語の処理系で用いられる不可視ポインタなど「透明な」ポインタを，S_2'とS_2''の関係付けに用いることができる。例えばS_2''の各語にS_2'の対応する語へのポインタを格納することによって，S_2'とS_2''を論理的に等価にすることができる（以下，順ポインタ法という)。逆にS_2'内容を全てS_2''にコピーし，S_2'の各語をS_2'の対応する語へのポインタに変更することも

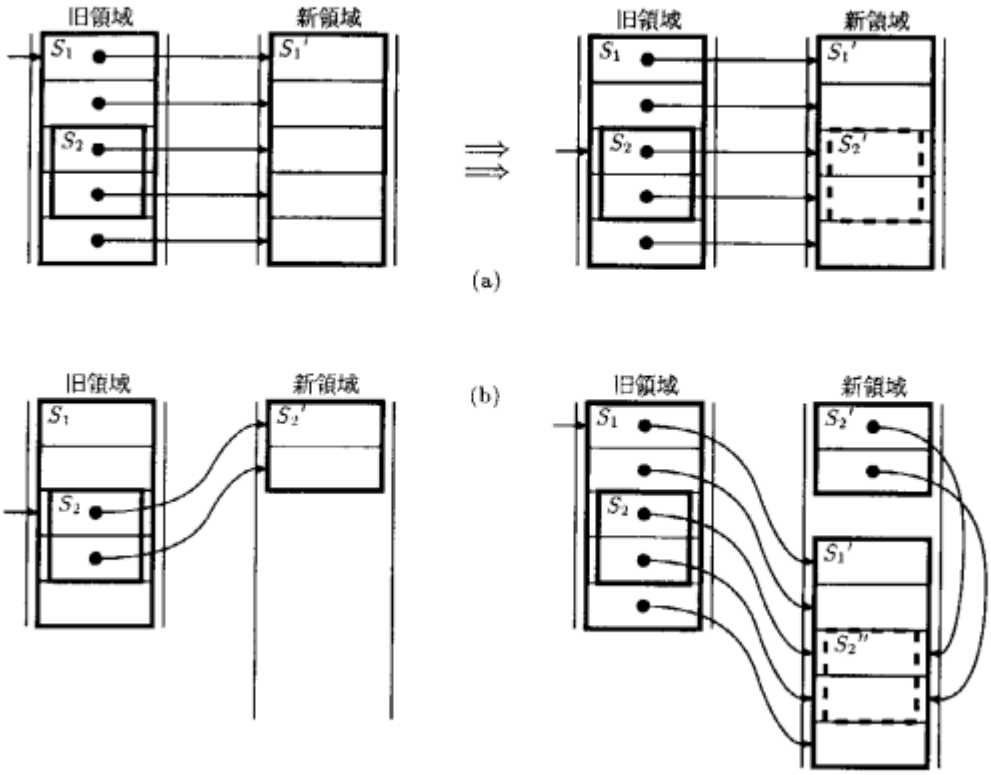

図 3: 部分構造の処理

できる（逆ポインタ法）。なおいずれの方法においても，新領域のスキャンの際に新領域へのポインタが出現するので，これを無視するようにしなければならない。

逆ポインタ法は透明なポインタを持たないような言語処理系にも比較的容易に適用でき，コピーが全て完了した後で S_2' を指すポインタを S_2'' を指すように変更することができる。即ち，S_2' から S_2'' へのポインタを通常のものと区別できるようにしておき，コピー完了後の後処理として一度だけ新領域をスキャンするか，あるいはルートからポインタの連鎖を再度たどることにより，S_2' を指すポインタを発見して変更することができる[1]。

そこで，ここでは逆ポインタ法を選択することとするが，順ポインタ法を選択しても以下の議論は本質的に変わらない。また，いずれの方法においても旧領域の S_1 の各語には，S_2 の部分も含めて S_1' へのポインタを格納する。なお S_2 を二重にコピーするため，新領域の大きさが旧領域の大きさよりも小さいことが保証できなくなるが，これを解決する効率的な方法はない。

以上をまとめると，付録 A.1における関数 $copied(w)$ と手続き $copy\_words(ao, n, an)$ を，図 A.4のように変更すれば良い。なお，説明を簡単にするために部分構造ポインタはデータ構造の任意の語を指しうるものとし，$new\_area\_p(w)$ は w が新領域へのポインタ語であれば真を返す関数，$trtag$ は透明ポインタを表すタグとする。

なお，部分構造 S_1 と S_2 が互いに相手を含まず，かつ一部の領域を共有することがあると，付録 A.4に示した方法では S_1 や S_2 が複数回コピーされることがある。例えば S_1 が S_1' にコピーされた後に S_2 が S_2' にコピーされ，更にその後で S_1 へのポインタが再び現れるとする。S_1 と S_2 の共有部分は S_2' にコピーされているので，S_1 の各語のポインタは連続領域を指示しておらず，コピー済とは判定されない。これが繰り返すと S_1 と S_2 のコピーが多数できてしまう。この問題の解決法については，3.4節で述べる。

[1] 順ポインタ法では一度スキャンしてポインタを逆転し，その後もう一度スキャンしてポインタ変更を行なう必要がある。

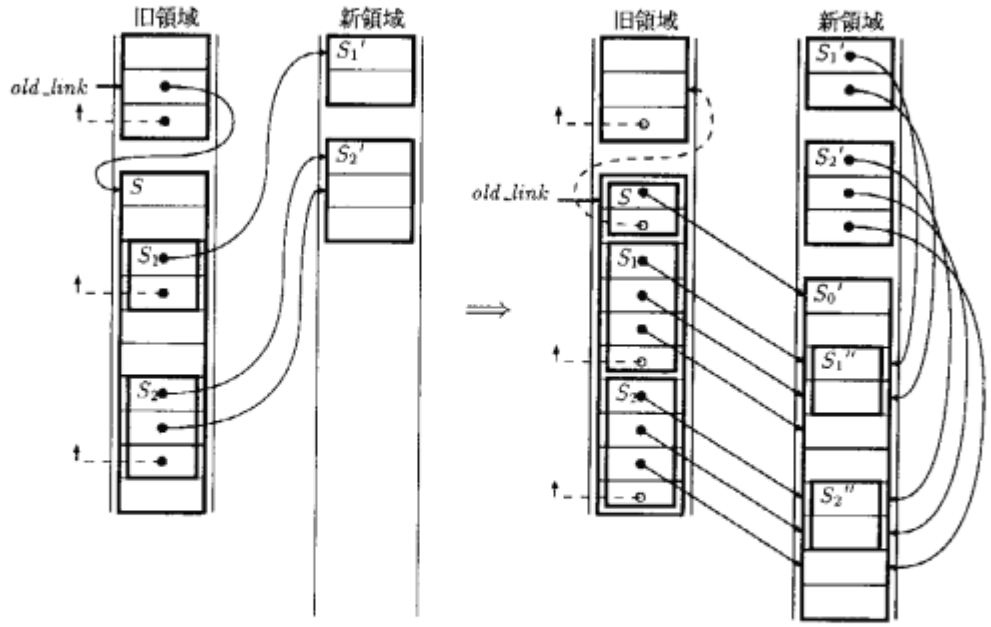

図 4: リンク法でのコピー中部分構造の処理

3.2 リンク法での問題と解決法

リンク法では未コピーとコピー済の他に「コピー中」という状態が存在するが，2.2節で述べた基本的な方式ではこのコピー中とコピー済とを同一視することができた。しかし，データ構造の一部を指すポインタが存在すると，コピーしようとするデータ構造の中にコピー中の部分構造が含まれていることがあり，これをコピー済のものと区別する必要が生じる。また非終端構造の末尾要素（リンク）が新領域へのポインタに見えることを利用して終了判定を行なったが，コピー中のデータ構造の中にコピー済の部分構造が含まれている場合，その要素であるコピー先へのポインタをリンクと区別する必要もある。

そこで，コピー先へのポインタとリンクとをタグなどを用いて区別するとともに，個々の要素を処理する前にあらかじめ全ての要素を3.1節で述べた方法で新領域にコピーし，末尾以外の要素には新領域へのポインタを格納するように変更すればよい。但し，図4に示すようにコピー中の非終端構造 S_1 と S_2 を含む非終端構造 S のコピー処理では，各々のコピー先 S_1' と S_2' の各語を S のコピー先 S' に移動するとともに，S_1 と S_2 のリンクを移動する。即ち，S_1 と S_2 の間の語が S_1 の一部に，S_2 よりも後の語は S_2 の一部となるようにし，これらの語が後に処理が S_1 や S_2 に戻った時に処理されるようにする。従って，直ちに処理されるのは S の先頭から S_1 の直前の語までの領域 (S_0) となる。

以上をまとめると，付録A.2に示した手続き *push_stack* を，付録A.5に示すように修正すればよい。なおリンクのタグとして透明ポインタのタグ *trtag* を用い，コピー先へのポインタのタグ *ptag* と区別できるようにしている。従って言語処理系が本来透明ポインタを持つか否かに関わらず，リンクのために特別なタグを用意する必要はない。また，付録A.2の手続き *link_gc* で用いている関数 *copied*(w) は，付録A.4に示したものとほぼ同じであるが，リンクをについては無条件に連続領域にコピー済と判断するようにする必要がある。更に，手続き *next_element* において，処理対象の要素を旧領域からではなく新領域からロードするように変更しなければならない。

3.3 予約スタック法での問題と解決法

予約スタック法では未コピーとコピー済の他に「コピー予約済」という状態が存在する。従ってデータ構造の一部を指すポインタが存在する場合には，コピーしようとするデータ構造がコピー予約済の部分構造を含む可能性があることを考慮しなければならない。また，コピー予約済の通知は（コピー済とは違って）先頭語に対してのみ可能であるので，先頭語を共有し大きさが異なるデータ構造のコピー予約にも若干の配慮が必要である。

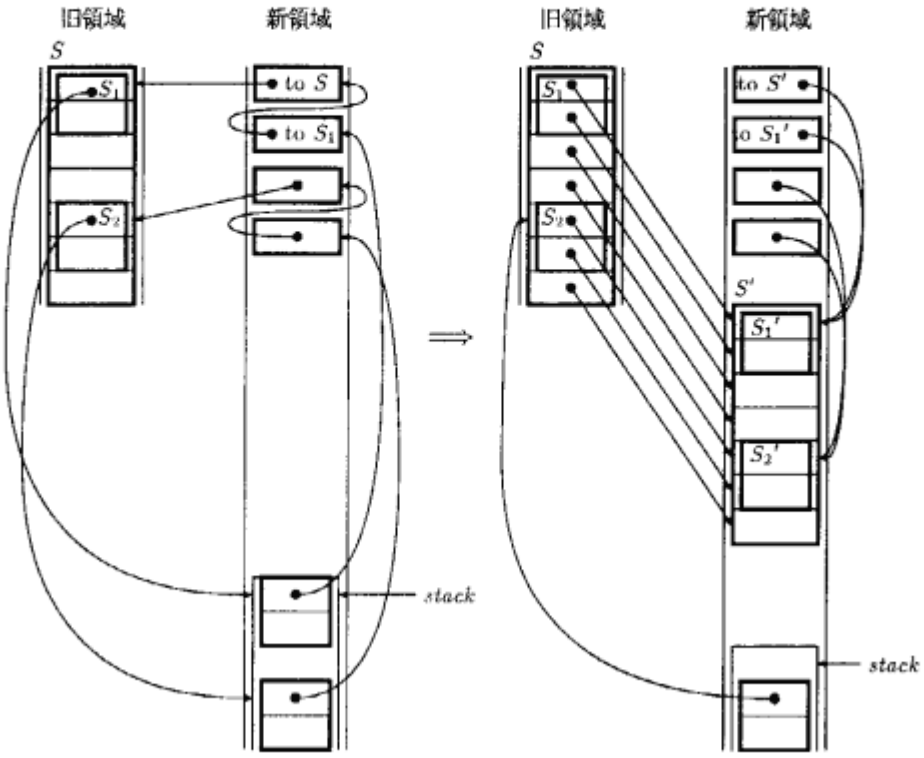

図 5: 予約スタック法でのコピー予約済部分構造の処理

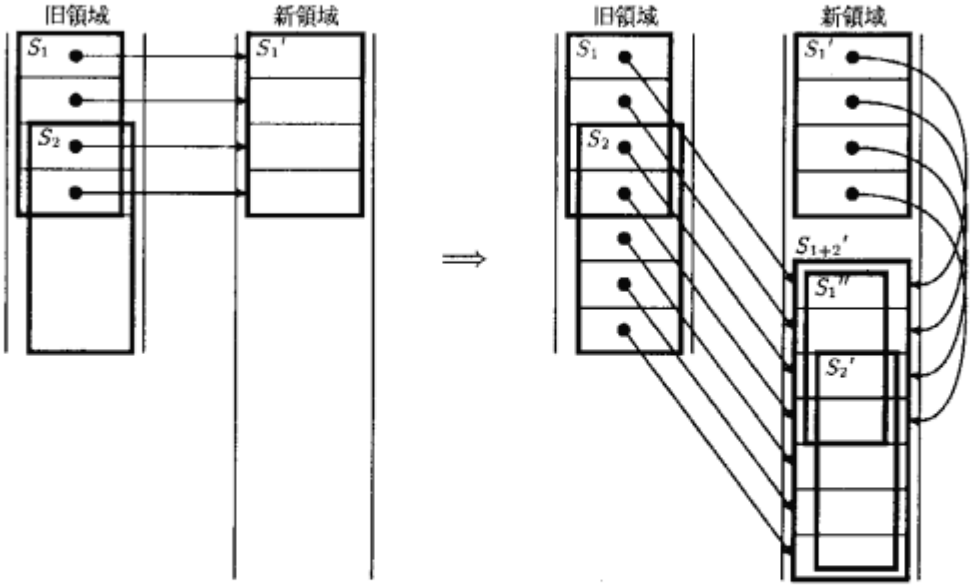

図 6: 互いに一部を共有する部分構造

ポインタ s が指すデータ構造 Sが全体としてコピー済か否かは，付録 A.4に示した関数 $copied(w)$ を一部修正し，予約スタックへのポインタを未コピー要素とみなすことにより判断できる。また予約済か否かの判断と，予約済であった時のポインタの連鎖への挿入は 2.3節で述べたものと同じである。

一方，予約スタックから Sの情報をポップして処理する際には，後述する方法で S がコピー済か否かを調べる。コピー済でなければ連鎖中のポインタの中で最大のデータ構造を指示するものを求め，それを S の大きさとする。次に，個々の要素を処理する前にあらかじめ全ての要素を 3.1節で述べた方法で新領域にコピーする。但し，予約スタックへのポインタ，即ちコピー予約済の部分構造があれば，予約スタックに退避された先頭語をコピーした上で，連鎖中のポインタが全て Sのコピー先を指すようにする。また予約スタックには任意の旧領域アドレス（例えば旧領域の部分構造の先頭アドレス）を入れ，部分構造がコピー済であることを通知する（図 5)。従って，予約スタックのポップの際には，ポインタの連鎖の先頭アドレスが新旧どちらの領域のものかにより，コピー済か否かを判断できる。

以上をまとめると，付録 A.3に示した手続き pop_stack を，付録 A.5に示すように修正すればよい。なお，付録 A.3の手続き $next\_element$ は，処理対象の要素を旧領域からではなく新領域からロードするように変更しなければならない。

3.4 互いに一部を共有する部分構造

3.1節で述べた方法では，部分構造 S_1 と S_2が互いに相手を含まず，かつ一部の領域を共有することがあると，S_1 や S_2が複数回コピーされることがある。これを防止するには，後から見つかった部分構造のコピーの際に，両者を合体した構造を新領域にコピーする必要がある。例えば，図 6に示すように，S_1がコピーされた後で S_2へのポインタが見つかった場合，何らかの方法で S_1の先頭語を見つけ，両者を合わせた構造 S_{1+2}を新領域にコピーする。逆に S_2が先にコピーされた場合は，S_1へのポインタを発見した時に S_2の末尾語を探す必要がある。

この S_1の先頭語あるいは S_2の末尾語の発見のために，S_1あるいは S_2を指すポインタを探索するのは非効率である。そこで，コピー済のデータ構造の先頭語とそれ以外の語をタグによって区別し，旧領域をスキャンしてデータ構造の拡大を行なうことが考えられる。例えば，先頭語のタグを $htag$，非先頭語のタグをこれまで通りの $ptag$ とする。

全体としては未コピーのデータ構造 Sをコピーする際，先頭語が $ptag$ をタグとする新領域ポインタであれば，Sの直前の語から逆方向に旧領域をスキャンし，最初に見つかった $htag$ を持つ新領域ポインタが拡大した構造の先頭語となる。また，Sの末尾語が $ptag$ をタグとする新領域ポインタであれば，S の直後の語から順方向に旧領域をスキャンし，最初に見つかった未コピー語あるいは $htag$ を持つ新領域ポインタの直前の語が拡大した構造の末尾語となる。

なおこの方法は旧領域をスキャンするので，重なりあった部分構造を持ちうるデータ構造の全ての語は，ポインタ語か否かがその語を調べるだけで明らかなる必要がある。また，リンク法の場合には $htag$, $ptag$, $trtag$ の 3 種類のポインタ型のタグが必要であり，スキャンの際には $trtag$ は $ptag$ と同一視する。

一方，予約スタック法の場合，データ構造の拡大は予約スタックから構造を取り出した時点で行なう。その際，データ構造に含まれる全てのコピー予約済構造を調べ，それらの中で最も末尾語のアドレスが大きいものを見つける必要がある。そのアドレスとデータ構造自身の末尾語のアドレスの大きい方が仮の末尾語となり，その語が $ptag$ をタグとする新領域ポインタであれば，さらに順方向にデータ構造を拡大する。

4 おわりに

本稿では，コピー型ゴミ集め方式における部分構造の処理方式を，従来方式と深さ優先順の二方式について述べた。

部分構造間の相互共有がない場合には，コピーされた二つのデータ構造が等価であることを示す「透明ポインタ」を追加すれば良い。また，論理型言語の処理系の変数参照ポインタのように，もともと透明ポインタを持つような場合には，ポインタの追加は不要である。

一方，部分構造間の相互共有がある場合には，二つのデータ構造を合体させる処理を行なえば良い。このためにはデータ構造の先頭とそれ以外を区別する必要があるが，リンク法以外ではこの区別に透明ポインタを利用することができる。

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: 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, Technical report, ICOT (1994). (to be published.).

[5] Chikayama, T., Yokota, M. and Hattori, T.: Fifth Generation Kernel Language, *Proc. Logic Programming Conf.* (1983).

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 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
    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 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 ;
```

A.4 部分構造がある時のコピー済判定と要素のコピー

```
function copied(w : mem_word) : boolean ;
   s :      integer ;      { データ構造の大きさ }
   h_old :  mem_addr ;     { データ構造の先頭（旧領域）}
   h_new :  mem_addr ;     { データ構造の先頭（新領域）}
   i :      integer ;      { 要素のオフセット }
begin
   s ← object_size(w) ;
   h_old ← address(w) ;
   w ← load(h_old) ;
   copied ← new_area_p(w) ;
   if copied then begin                                    { 先頭語がコピー済 }
      h_new ← address(w) ;
      i ← 1 ;
      while copied and i < s do begin                      { 全要素をスキャン }
         w ← load(h_old + i) ;
         copied ← new_area_p(w) and address(w) = h_new + i ; { 要素が連続した新領域へコピー済ならば真 }
      end ;
   end ;
end ;

procedure copy_words(ao : mem_addr, n : integer, an : mem_addr)
   i :  integer :     { 要素のオフセット }
   w :  mem_word ;    { 要素の値 }
   a :  mem_addr ;    { コピー済要素のコピー先 }
begin
   for i = 0 to n − 1 do begin
      w ← load(ao + i) ;
      if new_area_p(w) then begin                          { 要素はコピー済 }
         a ← address(w) ;
         store(an + i, load(a)) ;                          { 要素を部分構造から移動 }
         store(a, make_word(trtag, an + i)) ;              { 部分構造には透明ポインタをストア }
      end
      else                                                 { 要素は未コピー }
         store(an + i, w) ;                                { 単にコピー }
      store(ao + i, make_word(ptag, an + i)) ;             { 要素のコピー済を通知 }
   end ;
end ;
```

A.5 部分構造がある時のリンク法の *push_stack*

```
procedure push_stack ;                                     { スタックのプッシュ}
   last_nc :    integer ;    { 最後の未コピー要素のオフセット }
   last_tail :  integer ;    { 最後のリンクのオフセット }
   last_link :  integer ;    { 最後のリンクの値 }
   i :          integer ;    { 要素のオフセット }
   w :          mem_word ;   { 要素の値 }
   a :          mem_addr ;   { コピー済要素のコピー先 }
```

```
    procedure move_link ;                                  { リンクの移動 }
    begin
      if last_tail < offset then begin                     { リンクは未出 }
        if last_nc ≥ offset then begin                     { 未コピー要素あり }
          store(old + last_nc, make_word(trtag, old_link)) ; { 最後の未コピー要素をリンクにする }
          first_tail ← last_nc ;
      end ;
      else begin                                           { リンクは既出 }
        if last_nc ≥ offset then                           { 未コピー要素あり }
          store(old + last_nc, make_word(trtag, last_link)) ; { 最後の未コピー要素にリンクを移動 }
        else                                               { 未コピー要素なし }
          store(old + last_tail, make_word(trtag, last_link)) ; { リンクは移動しない }
      end ;
    end ;
  begin
    store(old + offset, word) ;                            { 先頭要素を復元 }
    last_nc ← offset − 1 ;                                 { 未コピー要素は未出 }
    last_tail ← offset − 1 ;                               { リンクは未出 }
    first_tail ← offset − 1 ;
    for i = offset to size − 1 do begin                    { 全要素を処理 }
      w ← load(old + i) ;                                  { 要素をロード }
      if new_area_p(w) then begin                          { 要素は新領域へのポインタ }
        if tag(w) = ptag then begin                        { 要素はコピー済 }
          a ← address(w) ;
          store(new + i, load(a)) ;                        { 要素を部分構造から移動 }
        end ;
        else begin                                         { 要素はリンク }
          a ← a + 1 ;                                      { 直前の要素の次にコピーされている }
          store(new + i, load(a)) ;                        { 要素を部分構造から移動 }
          move_link ;                                      { リンクを移動 }
          last_tail ← i ;                                  { 最後のリンクの情報を設定 }
          last_link ← w ;
  last_n ← ofsset − 1 ;                                    { 未コピー要素は未出 }
        end;
        store(a, make_word(trtag, new + i)) ;              { 部分構造に透明ポインタをストア }
      else begin                                           { 要素は未コピー }
        store(new + i, w) ;                                { 単にコピー }
        last_nc ← i ;                                      { 最後の未コピー要素のオフセットを設定 }
      end ;
      store(old + i, make_word(ptag, new + i)) ;           { 要素のコピー済を通知 }
    end ;
    move_link ;                                            { リンクを移動 }
    if first_tail < offset then                            { 処理領域に未コピー要素なし }
      next_element ;                                       { データ構造の処理を完了 }
    else if fist_tail = offset then                        { 先頭要素のみが未コピー }
      store(old + offset, make_word(ptag, new)) ;          { リンクせずに要素を処理 }
    else begin                                             { 先頭要素以外が未コピー }
      old_link ← old + offset ;                            { リンクを先頭要素に設定 }
      new_link ← new ;
    end ;
```

```
end ;
```

A.6 部分構造がある時の予約スタック法の *pop_stack*

```
procedure pop_stack ;
  s : integer ;      { データ構造の大きさ }
  a : mem_addr ;     { 要素のコピー先アドレス }
  b : mem_addr ;     { 部分構造へのポインタの連鎖のアドレス }
  i : integer ;      { 要素のオフセット }
begin
  old ← load(stack) ;
  while old_area(old) then begin                { コピー済のデータ構造を全てポップ }
    stack ← stack + 2 ;
    if stack = new_base + new_size then begin   { スタックが空ならば終了 }
      stack_empty ← true ;
      return ;
    end ;
    old ← load(stack) ;
  end ;
  repeat                                        { データ構造の旧領域アドレスを求める }
    old ← address(load(old)) ;
  until old_area(old)
  store(old, load(stack + 1)) ;                 { 先頭語を復元 }
  size ← 0 ;
  repeat begin                                  { ポインタの連鎖をたどる }
    word ← load(new) ;
    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 }
    s ← object_size(make_word(tag(word), old)) ;
    if s > size then                            { 最大のデータ構造の大きさを求める }
      size ← s ;
    new ← address(word) ;
  end ;
  until old_area(new) ;
  offset ← 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 }
  old ← old + offset ;                          { 先頭要素のアドレスを設定 }
  new ← new_tail + offset ;
  n ← size − offset ;                           { 未処理要素数を設定 }
  for i = 0 to n − 1 do begin                   { 全要素を処理 }
    word ← load(old + i) ;                      { 要素をロード }
    if new_area_p(word) then begin              { 要素は新領域へのポインタ }
      a ← address(word) ;
      if reserved(word) then begin              { 要素は予約スタックへのポインタ }
        store(new + i, load(a + 1)) ;           { 先頭語を予約スタックからコピー }
        b ← load(a) ;
        store(a, old + i) ;                     { 予約スタックに旧領域アドレスをストア }
        repeat begin                            { 連鎖中のポインタがコピー先を指すようにする }
          word ← load(b) ;
          store(b, make_word(tag(word), new + i)) ;
          a ← address(word) ;
```

```
            end ;
            until old_area(b) ;
        else begin ;                                  { 要素はコピー済 }
            store(new + i, load(a)) ;                 { 要素を部分構造から移動 }
            store(a, make_word(trtag, new + i)) ;     { 部分構造に透明ポインタをストア }
        end ;
    end ;
    else                                              { 要素は未コピー }
        store(new + i, word) ;                        { 単にコピー }
    store(old + i, make_word(ptag, new + i)) ;        { 要素のコピー済を通知 }
  end ;
  word ← load(old) ;                                  { 先頭要素をロード }
  new_tail ← new_tail + size ;                        { コピー先の確保 }
  stack ← stack + 2 ;                                 { スタックをポップ }
end ;
```