Vol. 35 No. 11　　情　報　処　理　　Nov. 1994

解　説

ごみ集めの基礎と最近の動向

5. 保守的ごみ集め†

小野寺　民也††

1. 序

記憶域におけるポインタの所在が正確に把握されていない状況で行われるごみ集めは，**保守的ごみ集め**ないし**保守回収** (conservative collection*) と呼ばれる．自動回収といえば主として記号処理言語や関数型言語の話題であったが，保守回収の登場により，C や C++ といった汎用言語においても本格的に追求されるようになってきた．本稿では保守回収についての入門的解説を試みるが，広く汎用言語の実装者にも興味をもっていただけるものと信ずる．

自動であると手動であるとにかかわらず，動的メモリの回収器は割当器と不可分の関係にある．したがってこれらを**ヒープ管理系**として把握することが重要である．一つのプロセスにおいて使用されるヒープ管理系が一つであるとは限らない．実際，後で詳述する例では，「自動回収を行うヒープ管理系」と「malloc と free の対からなる手動回収の系」が並存している．また，Lisp 処理系の場合も，固定長のセル用，可変長のシンボル用，文字列用，と別々のヒープ管理系を使用していることがある．通常**ヒープ**といえば念頭にあるプロセスの動的記憶域全体を指すが，混乱の生じない限り，本稿では念頭にあるヒープ管理系の制御下にあるヒープ領域のことを単にヒープということにする．また，ヒープ管理系によって割り当てられ回収されるものを**オブジェクト**ということにする．

さて，印掃式 (mark-and-sweep) や複写式 (copying) 回収において**ポインタ追跡**は重要不可欠な作業である．常識的に考えれば，追跡するためには回収器はオブジェクトへのポインタのある場所を正確に知っている必要がある．このために Lisp の処理系などではポインタやオブジェクトにタグを付けるのが常套手段となっている．ところが，記憶域におけるポインタの場所について必ずしも正確な知識をもっていなくても，ポインタ追跡は可能である．トリックは単純で，ポインタであるか否か曖昧な語でポインタと同じビットパターンをもつものはすべてポインタとみなすのである．このようにして追跡を行っていく回収を保守回収という．結果として本当はごみであるオブジェクトまで保持してしまうかもしれないが，いわゆる**懸吊ポインタ** (dangling pointer) を生み出さないという点で，回収器は安全に動作する．

ポインタは，一般に，レジスタ，スタック，大域部およびヒープの各域に存在する．これら全域が曖昧であるときの保守回収を特に**全保守** (fully conservative) といい，一部が曖昧であるときを**半保守** (partially conservative) という．半保守回収では，レジスタとスタックのみを曖昧だとするものが特に重要である．

追跡時の作業である「語 w のビットパターンがポインタのビットパターンと一致しているか否かを調べること」を**ポインタ判別**といい，「本当はごみであるオブジェクトまで追跡し保持してしまうこと」を**過剰追跡**という．前者は保守回収の設計上の要点であり，後者は実用上の論点となる．

本稿の構成は次のようになっている．まず，**2.** で保守回収と回収方式の関係についてまとめ，**3.** で保守回収の意義について述べる．次いで，**4.** でポインタ判別のアルゴリズムの一例を詳説し，**5.** で過剰追跡について論ずる．最後に **6.** で総括する．

† Introduction to Conservative Collection by Tamiya ONODERA (IBM Japan, Tokyo Research Laboratory).

†† 日本アイ・ビー・エム(株)東京基礎研究所

☆解説者は以前これを「保整回収」と訳出したが，必ずしも普及しているとはいえないため，本稿では保守的に「保守回収」の訳を用いることにする．

2. 回収方式との関係

全保守回収の場合，オブジェクトの移動をともなうような回収方式に適用することはできない．ポインタの値を更新するがゆえにポインタの場所を正確に知っている必要があるからである．したがって，非圧縮の印掃式が採用されることとなり，圧縮印掃式や複写式には使えない．ただし，いわゆるオブジェクト表を導入することにより，ポインタ追跡は保守的だがポインタ更新は正確に行うことができる．つまり，オブジェクト本体へのアクセスを必ずこのテーブルのエントリを介して間接的に行うのであるが，実行効率の低下が著しいためあまり有望な方法とはいえないであろう．

半保守回収の場合は，オブジェクトの移動をともなう方式にも適用可能である．たとえば，複写式と組み合わせることができる．そのトリックは，「複写できないものは複写しない[1]」ということである．つまり，「曖昧な領域から参照されているオブジェクトは動かさず，それ以外のオブジェクトを複写移動する」のである．この場合当然ながら，曖昧な領域をかなり限局できなければ複写式の利点が生きてこない．

参照計数式 (reference counting) はポインタ追跡をともなわないので，保守回収とは無関係である．しかしながら，遅延参照計数 (delayed reference counting) の場合は，スタック中のポインタを識別する必要が生じてくる．この識別を「保守的に」行うことも可能で[11]これは半保守回収の変形と考えることができる．

一方，世代回収と保守回収は直交する概念である．実際，印掃式全保守世代回収[8]や複写式半保守世代回収[2),10]なるものが提案されている．

表-1に主な保守回収器の実装例をまとめておく．圧縮印掃式といえば逆転ポインタ法による美しいアルゴリズムがあるが，現在の実用システムにおいては「忘れられた方式」といってよく，表-1にもこの辺の事情が反映されている．本稿でも圧縮印掃についてはこれ以上議論せず，単に印掃式といえば非圧縮のものを指すものとする．また，ここでは取り上げなかったが，並列化という点では，印掃式の場合の例[6]があることを補足しておく．

表-1 主な保守回収器

	(非圧縮) 印掃式	複写式	遅延参照計数式
全保守	Boehm et al.[5]	不可	不可
半保守	—	Bartlett[1]	Rovner[11]
全保守世代	Demers et al.[8]	不可	不可
半保守世代	—	Bartlett[2] Onodera[10]	—

3. 意義

保守回収の決定的な意義として二つ指摘することができる．第一に，保守回収は，自動回収のない既存の言語に自動回収を組み込む際に威力を発揮する．たとえばC言語の処理系のように，自動回収を意図せずに作成された言語処理系では，ポインタ追跡に必要な情報は通常生成されない．すなわち，ヒープに割り当てられるオブジェクト内のポインタ位置やスタックフレーム中のポインタ位置に関する情報は実行時に存在しないのである．これに対処する方法として，言語処理系を全面的に修正してポインタの在処をすべて明確にするのも一法であろうが，作業量が大きすぎる．全保守回収を採用することにすれば，多少の過剰追跡はあるものの，言語処理系や言語仕様をなんら修正することなく自動回収を組み込むことができる．実際，Boehm ら[5]は，全保守回収によりC言語に印掃式回収器を組み込んでいる．

さらに，言語処理系に対するほどほどな修正により，レジスタとスタックのみを曖昧とする半保守回収を用いることも合理的な選択である．なぜなら，比較的簡単な修正を言語処理系に加えることにより大域部とヒープにおけるポインタの所在を明確にすることができるからであり，また，半保守回収は全保守に比べて過剰追跡が少なくかつ回収方式として複写式を採用することも可能となるからである．

保守回収の第二の意義は，自動回収のある言語Gを自動回収のない高級言語Hで実装する場合に効率的な実装を可能とする点である．正確にいえば，言語Gがコンパイル方式の言語で高級言語Hに変換される場合，あるいは，言語Gがインタプリタ方式の言語でインタプリタが高級言語Hで書かれている場合である．例として，Lisp インタプリタのC言語による実装を考えてみる．保守回収を用いないとすれば，Lisp オブジェクトへの

ポインタを正確に識別するために，Cの実行時のスタックとは別にLispオブジェクト専用のスタックを用意することになるであろう．ポインタの把握は正確になるが，実行時スタックに比べ低速にならざるをえない（参考までにオブジェクト指向言語COBをC言語で実装したときの経験を述べると，COBオブジェクトを専用スタックに積んだ場合12〜31%の実行速度の低下が観測された）．しかし，保守回収を使用すればLispオブジェクトもまた，C言語のデータとの混在を恐れることなく，高速の実行時スタックに積むことが可能となるのである．

4. ポインタ判別

本章では，具体例として，Boehmらの印掃式全保守回収器[5]におけるポインタ判別のアルゴリズムについて述べる．

データ構造

Boehmらのヒープ管理系が採用しているデータ構造をまず説明する．ヒープ管理の単位は記憶塊で，必要に応じてOSから獲得され，大きさは4Kの整数倍で4Kバイトの境界に整合される．一つの記憶塊から割り当てられるオブジェクトの大きさは同一である．つまり，ヒープ管理系はオブジェクトの大きさごとに別々の自由リストをもっている．ただし，大きさが2K以上のオブジェクトはすべて一つの自由リストに繋がれる．

一つの記憶塊は，**図-1**に示すように4つの領域に分割されている．index部については後述する．size部にはこの記憶塊から割り当てられるオブジェクトの大きさが保存されている．マークビットはオブジェクト本体とは別の領域に格納されており，その大きさは，size部の値によって決定される．

管理系は，その制御下にあるすべての記憶塊の番地を配列marksに保存する．すなわち，記憶塊CをOSから獲得したとき，管理系は，配列marksの未使用の添字iを選び，marks[i]にCの番地を格納し，Cのindex部に添字iを格納するのである．

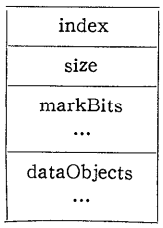


図-1 記憶塊のデータ構造

こうした初期化の意図について述べる．Boehmらは，このヒープ管理系がmalloc/freeの系と並存することを想定している．このため，ある記憶塊が自分の管理系に属していることを確認する必要が生じ，上述のようなデータ構造はこの確認のためなのである．つまり，先頭アドレスがaである記憶塊が自らの制御下にあることは，C言語ふうに書けば

```
marks[a->index]==a                    (1)
```

という条件式を評価することにより確認することができる．これは，**メモリ押印** (memory marking, たとえば参考文献7)) というよく知られた技法である．

アルゴリズム

語wに対するポインタ判別は，これをアドレスとみなして次の3条件の成立を順に確認することである．すべて成立する場合に限り，語wはポインタと同じビットパターンをもつと判断する．

1. 語wは管理系の制御下の記憶塊を指しているか？　具体的には次の三つの小条件の成立を調べることになる．

（a）　wがヒープの上限と下限の間にあるか？

（b）　語wの下位12ビットをクリアしたものをaで表したとき，a->indexは配列marksの正しい添字であるか？

（c）　条件式(1)は成立しているか？

2. wがオブジェクトの先頭へのポインタであるか？　具体的には，wと& a->dataObjectsの差が0またはa->sizeの正整数倍になっていることを調べる．

3. wが指すオブジェクトが自由リスト上に繋がれていないか？

第3段は，すでにごみとなっているオブジェクトを追跡しないために必要である．簡単に実装するには，自由リスト上にあるか否かを示すビットをオブジェクトごとに用意するのがよいであろう．ただし，回収方式として印掃式を使っている場合一という条件がつく．なぜなら，このビットの操作のために，ごみとなったオブジェクトの数に比例した処理が必要となるからである．回収方式として複写式を用いている場合は，生きたオブ

ジェクトの数に比例した処理に基づかなくてはならないため，もう少し複雑な方法を用いなければならない[1),10)].

議　論

条件式(1)が示唆するように，aの指すページは読出可能でなければならない．このチェックが条件1(a)であるわけだが，OSによってはこのチェックでは不十分であることに注意する必要がある．たとえば，Mach OSではプロセスのヒープは連続しているとは限らず，条件1(a)の成立は読出可能であるための十分条件とはならない．すなわち，条件1(c)のチェックで保護違反が起こる可能性がある．このような場合の対処法としては，たとえば，ポインタ判別のときだけ保護違反を無視するように割込ハンドラを設定しておけばよい．

第2段では，オブジェクトが生きている間は**基底**ポインタ，すなわちその先頭へのポインタが存在していることが仮定されている．しかし，言語処理系や最適化の度合によっては，これは必ずしも真とは限らない．基底ポインタからなんらかの計算の結果得られるポインタを**派生**ポインタというが，前者は消滅し後者のみが存在していることがあるのである．たとえば，基底ポインタと定数を加算して得られる「オブジェクトの内部へのポインタ」が，典型的な**派生**ポインタである．派生ポインタのみが残存する場合は，第2段の条件をそれに応じて変更する必要がある．先の内部ポインタの例では，「w が & a->dataObjects と“aの指す記憶塊の最終バイトのアドレス”の間にあるか？」と変更することになり，条件はかなり緩和される．

保守回収にとって不幸なことに，ある種のCPU上のある種の言語処理系によっては，オブジェクトが生きているにもかかわらず基底ポインタもなく派生ポインタもオブジェクトの内部を指していないことがあり得る[3)]．この問題を根本的に解決するには言語処理系から支援を得る以外になく，この方向での提案もなされている[3)]．簡単にいえば，派生ポインタが生きている間は基底ポインタも生かしておく—というものである．

5. 過剰追跡

過剰追跡の程度は最終的にはまさにアプリケーションに依存するとしかいいようがないのだが，本章では，まず一般的特性を列挙し，しかる後にBartlett[1)]およびWentworth[12)]による測定結果を紹介し，最後に過剰追跡の抑制策について述べる．

一般的特性

過剰追跡の根本原因は，ポインタ判別の結果が誤って真となってしまうことである．したがって，判別が真となりやすいほど，過剰追跡もそれだけはなはだしくなるのであり，そのような状況をいくつか列挙すると次のようになる．

- 使用中のヒープ領域が広い．
- 派生ポインタが存在する．
- 走査領域が広い．

第1項および第2項は，第2節の条件1(a)および条件2をそれぞれ成立しやすくしてしまう．第3項からは系をいくつか導くことができる．たとえば，過剰追跡は全保守回収のほうが半保守回収よりも激しくなるであろう．また，ポインタがバイト整合される計算機のほうが語整合される計算機よりも実質的に走査領域が広くなり過剰追跡ははなはだしくなるであろう．保守回収の起動に関して，「スタックが長いときは延期すべし[10)]．」というのも第3項の教訓である．

測定結果

Bartlett[1)]は，自ら提案した複写式半保守回収器をSchemeに組み込み，Boyerベンチマークや Scheme コンパイラを動かし，追跡の結果複写されるオブジェクトの総量を測定している．それによると，派生ポインタがあると考えてポインタ判別を行った場合，そのために過剰に追跡される量は全ヒープの1％以下となっている．

一方，Wentworth[12)]は，Boehmらの印掃式全保守回収をPascalで書いたLispに組み込み，5つのプログラムを動作させている．各プログラムの最終的なセルの使用量は6Kから8K個である．セル領域の大きさを16K個としたとき，過剰追跡されたセルの割合は最大のもので7％程度であり，セル領域の大きさを32K個に増やすと，最大のもので3％と改善する．

以上の例に関する限り，過剰追跡は大きな問題を引き起してはいないが，Wentworthは深刻な事

態の発生も指摘している．深刻過ぎてプログラムが最後まで走行できなかったために測定結果はないのだが，原因について考察しているので，ここで簡単な例を用いて説明してみよう．次のようなリスト構造でキューが実現されているとする．要素の追加のために Node 型のデータが動的に作成され tail において付け加えられ，要素の取出は head においてなされる．

```
struct Node {
  struct Node *next;
  void *item;
};
struct Node *head;
struct Node *tail;
```

初期化や境界条件の詳細は一切無視するとして，要素の追加および取出ルーチンは次のように書くことができる．

```
void put (void* item){
  struct Node *p=new Node;
  tail->next=p;
  tail=p;
}
void *get(){
  struct Node *p=head;
  head=head->next;
  return p->item; //*p gets garbage.
}
```

何の変哲もない単純極まるコードだが，このキューが激しく使用され大量のノードが割り当てられついに自動回収が起動されるに至ったとしよう．少しばかり観察力を働かせてみると，このとき，最初取り出されたノードから最近取り出されたノードまでポインタによる連鎖がきれいに続いていることが分かる．曖昧な領域に存在するある語が，十分昔のノードへのポインタと同じビットパターンであったとすれば，文字どおり芋蔓式に大量のノードが追跡されることになる．

Wentworth によると，関数型言語におけるグラフ縮約も同様の問題を内包しているという．グラフ縮約の場合は，単なる一つのデータ構造という次元を越え計算機構そのものに関係しているだけに，問題はより深刻であるといえる．

抑 制 策

過剰追跡は基本的にポインタ判別が誤って真となるから生ずるのであるが，その原因には2種類ある．**誤認**と**残留**である．Boehm[4]はそれぞれに対して，興味深い対処法を提案しているので，ここで要点を紹介する．

ポインタ誤認とは，ポインタ型ではないデータ，たとえば整数型のデータが，ポインタと判別されてしまうことである．ポインタ誤認を極力回避するために Boehm の案出した方法は，アドレスの**黒表** (blacklist) の作成である．すなわち，近い将来誤認を引き起こしそうなアドレスは黒表に登録するのである．そして，そのアドレスを含むヒープ領域の割当使用はしばらく禁止するのである．

第2節の事例に沿って具体的に述べる．黒表はポインタ判別の際に作成される．すなわち，条件1(a)の判定において，語wがヒープの上限下限の間になかった場合，“近傍”にあるか否かがさらに調べられ，近傍にあるならば黒表に登録されるのである．その後記憶塊の要求のあった場合，ヒープ管理系は，黒表に登録されたアドレスを努めて避けて記憶塊を割当てるのである．

大域部にあるデータの中には初期化ののち長く値を変えないものが少なからず存在する．そうしたデータがポインタ誤認を引き起こすと，その結果追跡されるごみであるオブジェクトたちは同様に長く保持され回収されないままの状態となってしまう．アドレス黒表は主にこうしたポインタ誤認を避けることを狙っている．Boehm は，初期実験として，簡単なプログラムを黒表つきの管理系および黒表なしの管理系の上でそれぞれ動かして，全ヒープの何%が過剰追跡のため回収されないかを測定比較している．それによると，黒表なしで 10〜30% だったものが黒表つきでは 0〜3% にまで低下している．

残留ポインタとは，曖昧な領域に残されたポインタですでにごみとなっているオブジェクトを指しているもので，確実にポインタ判別を真ならしめる．先のキューの例はこの典型である．また，スタック領域は残留ポインタの主要な発生源で，その理由は，コンパイラが生成するコードは通常スタックフレームを割当のときも解放のときもゼロクリアしないようになっているからである．

残留ポインタをなくす有効な方法は，使用済みオブジェクトへのポインタはゼロクリアするよう

に心がけることである．キューの例では復帰前にp->next をゼロクリアすべきだったのである．一方，スタックフレームの場合は，ポインタ型の局所変数をゼロクリアするコードを書いたとしても，コンパイラの最適化によって除去されてしまう公算が大である．Boehm は，ある程度定期的に呼ばれるメモリ割当関数を利用して，その中で隣接する未使用のスタック領域をゼロクリアすることを提案している．C言語ならばそのようなコードを容易に書くことができる．

ポインタ誤認を回避し残留ポインタを抹消しても，それでもポインタ判別は誤って真となる．これには被害を最小限にするようにデータを編成することによって対処することになる．Boehm[4]は，このような観点から，オブジェクト群をポインタで上下左右格子連結するためのデータ構造について論じている．

6. 結　語

C言語に自動回収を組み込むことが可能であることを Boehm[5]らが実際に示して以来，保守回収はにわかに斯界の関心を集めている．これは，Cプログラマのメモリ管理に対する苦悩がいかに深いものであるかの証左であろう．では，Cの世界でも自動回収がメモリ管理の主流となるのか？それは自動回収の実行性能いかんに懸かっている．この点について，Zorn[13]による実に興味深い研究があるので，ここにその要点を紹介したい．

Zorn は，動的メモリを多用する6つのプログラムを5つのヒープ管理系と組み合わせて走らせ，「CPU 性能」「メモリ使用量」「参照局所性」を測定評価している．使用されたヒープ管理系には，本稿で紹介した Boehm らによる保守回収器が含まれている．他は，すべて手動回収に基づくもので，4.2 BSD の UNIX で配布されているmalloc/free の系や SUN OS のライブラリにある系といった，ワークステーションの世界で人口に膾炙したものである．測定結果を要約すると，保守回収器は「CPU 性能」では手動回収に伍しているものの，「メモリ使用量」は30〜150%ほど多くなっている．悲惨なのは「局所性」の指標の一つである「ページフォルト率」で，BSD の系に比べ1桁から2桁も悪くなっている．

結論をいえば，この保守回収器は，Cの世界より手動回収を駆逐し得るほどの性能に現段階で達しているとはいえない．しかし，だからといって，この時点で，保守回収自体を放擲してしまうのは性急すぎるであろう．改良の余地は大いに残されており，なかんずく，30年を越える自動回収研究の中でも最大の所産である世代回収の大技法がまだ控えているのである．

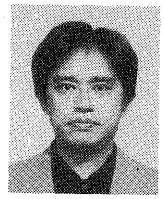
最後になるが，「保守的に追跡する」というBoehm の着想は，メモリ漏洩発見器に応用され，これが商用のプログラミング環境に組み込まれ[9]，大いに好評を博していることを付け加えておく．

参 考 文 献

1) Bartlett, J. F.: Compacting Garbage Collection with Ambiguous Roots, DEC WRL Research Report 88/2 (Feb. 1988).
2) Bartlett, J. F.: Mostly-Copying Garbage Collection Picks Up Generations and C++, DEC WRL Technical Note TN-12 (Oct. 1989).
3) Boehm, H. J.: Simple GC-Safe Compilation, Position Paper for OOPSLA '91 Workshop on Garbage Collection (Oct. 1991).
4) Boehm, H. J.: Space Efficient Conservative Garbage Collection, *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, 197-206 (1993).
5) Boehm, H. J. and Weiser, M.: Garbage Collection in an Uncooperative Environment, *Software Practice and Experience*, 18, 9, 807-820 (Sep. 1988).
6) Boehm, H. J. et al.: Mostly Parallel Garbage Collection, *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 157-164 (1991).
7) Comer, D.: *Operating, System Design, the Xinu Approach*, p. 231, Prentice Hall (1984).
8) Demers, A. et al.: Combining Generational and Conservative Garbage Collection: Framework and Implementations, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, 273-282 (1990).
9) Hastings, R. and Joyce, B.: Purify: Fast Detection of Memory Leaks and Access Errors, *Proceedings of the USENIX Winter '92 Conference*, 125-136 (1992).
10) Onodera, T.: A Generational and Conservative Copying Collector for Hybrid Object-oriented Languages, *Software Practice and Experience* 23, 10, 1077-1093 (Oct. 1993).
11) Rovner, P.: Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language, Xerox Palo

Alto Research Center Report CSL-84-7 (July 1985).
12) Wentworth, E. P.: Pitfalls of Conservative Garbage Collection, *Software Practice and Experience*, 20, 7, 719-727 (July 1990).
13) Zorn, B.: The Measured Cost of Conservative Garbage Collection, *Software Practice and Experience*, 23, 7, 733-756 (July 1993).

(平成6年3月29日受付)



小野寺民也（正会員）

1959年生．1983年東京大学理学部情報科学科卒業．1988年同大学院理学系研究科情報科学専門課程修了．理学博士．同年日本アイ・ビー・エム(株)入社，現在東京基礎研究所勤務．コンピュータ・グラフィクスの形式化，プログラム言語処理系，オブジェクト指向プログラミングなどの研究に従事．著書「A Formal Model of Visualization in Computer Graphics Systems」(共著，Springer-Verlag)．日本ソフトウェア科学会，ACM，IEEE，USENIX 各会員．