15. シーケンシャル・ガーベジ・コレクション†

黒　川　利　明††

概　観

ガーベジ・コレクション（ゴミ回収法，ゴミ集め，屑集め，屑拾いなどの訳語がある．以後は簡単に GC と略す．）について，どのような分類や，側面があるかを考えると**表-1**のようにまとめることができる．

このような GC に関する豊富な話題を要領よくまとめたものに **Cohen**[1]がある．

表-1(a)

- 対象データによる分類
  - 単一長 (single-sized)
  - 可変長 (varisized)
- プロセッサによる分類
  - 単一 CPU (停止型)
  - 単一 CPU (実時間型)
  - 複数 CPU (並列型)
- ゴミの見分け方による分類
  - セル内タグ・ビット
  - ビット・テーブル
  - レファレンス・カウンタ
  - 領域分離

表-1(b)

- システムの記憶管理による分類
  - 実記憶
  - 仮想記憶
- GC の段階
  - 印づけ (marking)
  - 回　収 (collection)
- 印づけの作業領域による分類
  - 固定域
  - 可変域（ゴミの量によって可変）
- 回収方法による分類
  - リストのチェイニング (chaining)
  - 圧縮型 (Compactify)
    - copying——非破壊
    - moving ——破壊
    - sliding——位置関係保存
    - condensing——できるだけ小さくする

表-1(c)

- 諸側面
  - アルゴリズムの提案
  - アルゴリズムの解析
  - アルゴリズムの検証
  - プログラミング言語への組み込み
    - Lisp
    - Prolog
    - ⋮
  - システムへの組み込み

1. 本稿の範囲と目的

同じ特集の中に日比野の「並列形ガーベジコレクション」，長谷川の「リストのコピー法」，佐々の「動的記憶領域割付け法」がある．本稿の範囲は，したがって，プロセッサの分類でいう「単一 CPU による GC」であり，そのうち回収に関する「リストのコピー法」についての詳細は長谷川に譲る．とはいっても「単一 CPU による GC」とは，とりもなおさず，現在の汎用電子計算機で実現される GC のすべてを含んだことになる．一応簡単にこれらに触れ，詳細については **Cohen** のまとめに譲り，本稿のポイントとしては，狭い意味のアルゴリズム論からは逸脱するかもしれないが，主として実用の観点から二，三の話題を取り上げて論じることにしたい．その話題とは，1) 実時間 GC, 2) GC の作業域の固定化，3) 記憶領域の巨大化，である．なお，参考文献については，**Cohen** が参考文献として取り上げていないものについて注意を払う．これからさらに検討しようとする読者は **Cohen**[1]を参考にされたい．

2. ガーベジ・コレクションの一般論

2.1 歴史と必然性

GC の歴史はリスト処理の歴史である．リストは**図-1**のようにポインタでつながったデータである．このデータは配列（アレイ）の場合と異なり，論理的には**図-2**のように，どう配置されていても構わない．

この事実は，不要になったリストの要素（図中の箱）をシステムが自動的に回収し，ユーザに供給することを可能にする．GC は見かけ上，「無限の論理的記憶を与える」ために考案された．

もちろん，この便宜は GC システムの犠牲の上に立

† Sequential Garbage Collection by Toshiaki KUROKAWA (Institute for New Generation Computer Technology).
†† (財)新世代コンピュータ技術開発機構

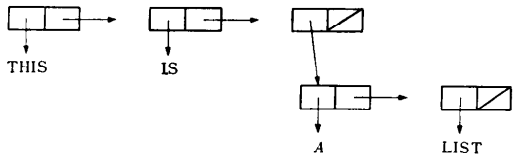

図-1　リストの例

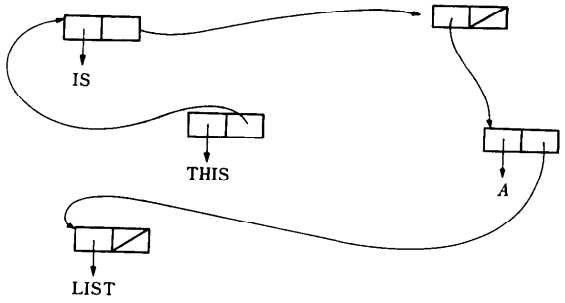

図-2　リストの適当な配置例

っている．オーバヘッドを避けるにはユーザが自分で記憶管理をすればよい．リスト処理システムは，この便宜を優先し，オーバヘッドはシステムに（現在はハードウェア）に任せようとする．それは，プログラムを物理的なハードウェアから離し，論理的な抽象機械で考えようとする試みと同じ軌道上にある．

2.2　印づけと回収

表-1(b)の GC の段階で述べたように，GC は印づけと回収の2段階をもつ．印づけは，ゴミとゴミ以外のものの区別をする．家庭のゴミとは反対に，「現在使用中」という印をつける．（ゴミが真っ先に印を付けられるものには，レファレンス・カウント法がある．もっとも，実態は使用中という印がついていないという意味の印である．他は使用中という印が最初につく．）

回収には，表-1のように，チェイニングと圧縮がある．チェイニングはリストの持つポインタをそのまま使う方式で，最も簡単である．圧縮型を取るのは，データが可変表であったり，仮想記憶でページ・フォールトの問題があったりする場合で，どちらかというとシステムの全体性能を考えた技法である．

回収についての一般論は「動的記憶領域割付け法」に譲る．

実時間 GC では，この回収を実行途中で順次行う．本節で説明する GC のアルゴリズムは，リスト要素の供給を全面的に停止し，印づけ後に一括回収する方式である．

回収した結果，仕事が無事修了するかどうかは，必ずしも明白ではない．仮想記憶の **Lisp** システムなどでは，全体の 10% 以下の要素しか回収できないときには，領域のサイズ自体を 10% 程度増やすものもある．このようなことは，システムに組み込んだ場合の運用論になる．

2.3　基本アルゴリズム

以下紹介する基本アルゴリズムには，多くの変種がある．基本形とその特徴，および筆者の評価のみを加えるに留める．詳細は **Cohen** などに譲る．

基本アルゴリズムとしては，次の4つに絞る．

1. **スタックとマーキング・ビット**を用いる法．
2. **レファレンス・カウント**を用いる法．
3. **コピー**する法．
4. **ポインタ反転法**

2.4　スタックとマーキング・ビットを用いる法

リストの印づけは，有向グラフの結合性判定の問題に帰着できる．使っているリストの根があれば，そこから順次たどっていけるリスト要素を印すればよい．単純には，すべてのリスト要素を順次調べては，子供を見つけ，追加する．

再帰的な考え方では，リストの根から出発して，各子供について，再帰的に（各子供を根だと思って）印づければよい．

$$\mathrm{mark}(x)=\mathrm{mark}(\mathrm{car}(x))\ \&\ \mathrm{mark}(\mathrm{cdr}(x))$$

この再帰プログラムをバラして，スタックを陽に表わすと，子供をスタックに積んで，空になるまでスタックの要素をマークするという方式になる．

本アルゴリズムの特徴は，次の通り．

- 巡回リスト（cyclic structure）も回収可能．
- 印づけにリスト要素を1回チェックするだけ．
- データ構造の大きさに比例したスタックがいる．

欠点は，スタックのサイズであり，これを回避するために，幾つもの変種が考案されている．

筆者の **Lisp** システム上での経験では，本当にデータ構造（リスト構造）の大きさに比例するような，特殊な構造が出てくるのは稀である．通常の場合，GC 用のスタックのサイズは有限個（例えば 10 個分）で十分であり，制御スタックと共有していればほとんど心配ない．（ただし，スタックの使用量を最小にする努力は必要．先読みの技法を徹底的に用いる[4]．）

実用的には役に立つと評価できる．

2.5 レファレンス・カウントを用いる法

リスト要素に対して，どれだけ他のリスト要素から参照されているかというカウンタを持っておく．もちろん，リスト操作で，ポインタをつけたり，外したりするときには必ず，カウンタの面倒をみる．

GC の時は，カウンタが零のものを回収する．

この方法の特徴は，以下の通り．

- 印づけの段階が不要．
- カウンタのサイズ分，データ領域が心要．
- 巡回リストのようにお互いに呼び合っているリスト要素は回収できない．
- リスト処理にカウンタ操作が入って遅くなる．

レファレンス・カウントの利点は印づけ不要にある．データ領域の問題については，**Cohen**[1]にあるように**Deutsch** と **Bobrow** の提案* もあるが，最悪の場合ポインタ幅のサイズがいる．リスト処理の速度は，いわゆる **Lisp** マシンなら無視できる．

欠点として，巡回リストや，有限のカウンタでのカスのようなものの処理に手間がかかるのをどう評価するかということになる．

筆者の私見では，特殊ハードウェアの支援がこの方法には必要である．

2.6 コピーする法

リスト領域に A と B の2面を用意し，現在使用中の A から，使用中のリスト要素だけを B にコピーする．A の使用中のリスト要素をすべて B にコピーすると，A と B の役割を交換すると A はすべて使用可能になる．

この方式の特徴は，次のようになる．

- 印づけ不要（コピーがその役割を果す．）
- コピー時にデータ領域の圧縮ができる．
- リスト領域として半分しか使えない．

仮想記憶システムでは，ページのスワップ操作とこの操作を重ね合わせて，さらに不可視ポインタ (invisible pointer) を導入して，実時間 GC にできる．

レファレンス・カウンタを用いるのに比べ，落ちこぼれのないのがよいが，半分しか領域が使えないという点を考えると，二の足を踏むというのが，筆者の気持である．

2.7 ポインタ反転法

Schorr と **Waite*** により開発された技法でアルゴリズムとしては非常に面白い．リストをたどってゆく時，帰り先を図-3のように，リスト要素の中へ埋め込んでおく．こうすると次には帰り先のリストをたどって元へ戻ることができる．

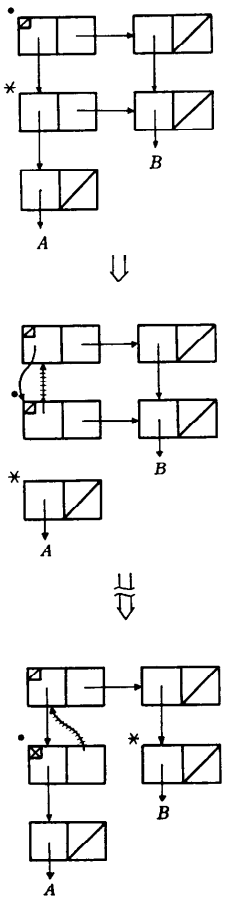

図-3 ポインタ反転

*は現在着目しているリスト要素
・はその一つ前のリスト要素

この方法の特徴は次の通り．

- 印づけの段階でスタックなどの作業域が不要．
- データ自身を途中で変更する．
- リスト要素を3度チェックする．

データの中に途中の作業情報を埋め込むところに，この手法の見事さがあり，また使用時の欠点がある．

実時間 GC などでは特に，このようなデータ自身の構造変更は不可能である．筆者の経験では，実際に使ってみると印づけにかかる時間が問題で，面白いが使う気になれなかった．

3. 実時間ガーベジ・コレクション

ここ数年，GC が注目を浴びてきた理由の1つは実時間 GC の実現にある．**Dijkstra*** をはじめとして，

Steele*, Baker* らが並列ないし実時間 GC のアルゴリズムを提案して注目を浴びた．(並列 GC についてはアーキテクチャ上の興味も大きい，日比野[2]参照.)

実時間 GC が注目を浴びたのは，それが GC のアルゴリズムに新しい分野を拓いたというだけではない．第一に GC を使ったシステム (Lisp など) において実時間応用が要求され出したこと，具体的にはロボットへの応用．第二に Lisp マシンが実用化されて実時間 GC がインプリメントされるようになったことがある．

MIT がプロトタイプを開発した Symbolics 社, LMI 社の **Lisp マシン**の場合，現時点ではコピー・アルゴリズムを用いているという．2 枚の記憶空間を用いて使用中のリストをコピーするものである．Baker* の**コピー・アルゴリズム**は仮想記憶方式におけるページの入れ替えを巧みに利用している．その代償として記憶容量が 2 倍必要となる．Lieberman と Hewitt* はこれを 2 枚から n 枚に拡張するという案を示している**.

Interlisp をベースにした XEROX 社の **Lisp** マシンの場合には，レファレンス・カウント法を使っている公算が大きい．富士通研の **Lisp** マシン[3]もページ単位のレファレンス・カウンタを持っている．

実時間 GC の問題点は，1) 使用記憶量の増大 (**レファレンス・カウント**でも 1/4 位はいくのではないか)，2) 実記憶システムでは明らかに，GC のために処理を停止させる古典的な手法に比べて実行時間が劣ること，があげられる．

仮想記憶システムの場合には，1) ディスク上でポインタ操作すると極端に処理速度が遅くなること，2) データをページ内に圧縮すると処理効率が大幅に向上すること，3) 仮想記憶の単価が大幅に低下していること，の三つの理由から実時間 GC が有効となっている．将来方向としては，仮想記憶システムにロジックを組み込むことにより一層効率化を進めることになるのではなかろうか．

実用的には，汎用大型の仮想記憶方式計算機における実時間 GC をどう実現するかという問題がある．仮想記憶方式の実現法や **Lisp** などの応用プログラムから多少なりとも情報を送れるかどうか，によっても異なるが，試してみる価値があろう．

* **Cohen** の論文に紹介があるので文献表にのせていないもの．

** Symbolics 3600 では GC は未実装．Baker のコピー方式でも問題が残るらしい．

4. 作業領域の固定化

GC の問題として，処理速度と GC 用の作業域がいつも問題点となっていた．

基本的アルゴリズムで紹介したように，実用上は一時中断してもよいなら，処理効率のよいスタックを使った方式も固定サイズで十分である．

実時間方式となると，各リスト要素に対して **Dijkstra** の方式のように，色分けするなら(並列ガーベジ・コレクション参照) 1 ビットずつ，レファレンス・カウントなら例えば 1/2 バイトずつ，コピーなら 1 ワードずつ作業域が必要となるわけである．

Lisp のようなシステムでは，常に使用するデータなどは，GC の対象から外してしまう方が印づけの手間が省ける．

そもそも，ゴミでない物を区別するために (回収できないデータのために)，作業域が必要だというのは残念なことである．作業域が必要だという場合には，それだけ回収されるゴミの期待値が小さいのだから．

かといって，作業域を少なくしようとポインタ反転法を用いるのは，得策とはいえない．効率的にスタックを用いた方法の方がよい．

筆者の私見では，ゴミの判別より，当分使用するデータを見分けることが，これから重要となる．そうすれば自動的に作業域の固定化が可能となろう．

5. 記憶領域の巨大化

GC の問題がこれから重要になるのは，記憶領域が巨大化して GC 時間が大幅に増加するようになるだろうからである．

巨大化の一時期においては，MIT の **Lisp** マシンが一時そうであったように，GC 自体必要ないという状態がある．ただし，このような時期が長続きはせず，いくらメモリを増やしても記憶領域が足りないと嘆くことになる．

発想として "Rent-a-memory"[5] が注目を浴びたのも，いずれはそのような解決策しかないかも知れないという感じを誰もが抱くからであろう．

当面考えられている策は，プロセッサの並列化と，メモリのバンク分けである．基本的には divide and conquer の線に沿っている．

この分野については，もう少し実用上の検討が必要なのかもしれない．

6. その他の話題

GC に関与する言語として従来は **Lisp** が中心的だったが，例えば **Prolog** だとか FP だとかの場合にはどうなのかは，最近検討が始まったばかりである[6),7)].

GC の応用分野としては，グラフの処理をはじめとして，ファイル・ディレクトリの管理やストリング列の情報管理（エディタなど）がある．個々の応用事例については，GC の一般的手法とは独立に問題が処理されているように見受けられる．

これらの個別事例についても，一般的手法と比較検討することによって，より効率のよい記憶管理が実現できるのではないかと，筆者は思っている．

参 考 文 献

1) Cohen, J.: Garbage Collection of Linked Data Structures, ACM Computing Surveys, Vol. 13, No. 3, pp. 341-367 (Sep. 1981).
寺島訳，つなぎのあるデータ構造のくず集め，bit 別冊 コンピュータ・サイエンス '81.
2) 日比野：ガーベジコレクションとそのハードウェア化，情報処理，Vol. 23, No. 8, pp. 730-741 (Aug, 1982).
3) 服部，篠木，品川，林：高速リスト処理に適したアーキテクチャについて，情報処理学会第18回記号処理研究会資料，pp. 95-101 (Mar. 1982).
4) Kurokawa, T.: A New Fast and Safe Marking Algorithm, SOFTWARE-Practice and Experience, Vol. 11, pp. 671-682 (Ang. 1981).
5) White, J. L.: Address Memory Management For A Gigantic LISP Environment or, GC Considered Harmful, Conf. Rec. of 1980 LISP Conference, pp. 119-127 (Aug. 1980).
6) Bruynoogh, M.: A note on garbage-collection in Prolog interpreters, Proc. First International Conf. on Logic Programming, pp. 52-55 (Sep. 1982).
7) Hudak, P. and Kellen R. M.: Garbage Collection and Task, Deletion in Distributed Applicative Processing Systems, Conf. Rec. of 1982 ACM Symposium on Lisp and Functional Programming, pp. 168-178 (Aug. 1982).

（昭和 57 年 12 月 2 日受付）