相補型ガーベジコレクタ

松井祥悟[†] 田中良夫[††]
前田敦司[††] 中西正和[†††]

本論文では，並列型（parallel）および漸次型（incremental）ガーベジコレクションの基本アルゴリズムである相補型ガーベジコレクタ（Complementary Garbage Collector）の提案およびその評価を行う．このアルゴリズムは，増分更新型（incremental update）とスナップショット型（snapshot-at-beginning）という2つの基本アルゴリズムを相補的に組み合わせたものである．ゴミの回収効率の良さと正当な（無矛盾な）実装の容易さという両者の長所を併せ持つ．このアルゴリズムは，現在広く使用されているスナップショット型アルゴリズムを代替する．この型を基本アルゴリズムとしている現存の並列型および漸次型ガーベジコレクションに直ちに応用できる．Complementary Garbage Collectorを並列型mark-and-sweep法および漸次型mark-and-sweep法に組み込み，評価を行った結果，ゴミセルの回収効率は一括型GCと同程度まで改善されることが確認された．これにより実行速度，実時間性（無停止性）が改善された．

Complementary Garbage Collector

SHOGO MATSUI,[†] YOSHIO TANAKA,[††] ATSUSHI MAEDA [††] and MASAKAZU NAKANISHI [†††]

This paper describes the design and evaluation of Complementary Garbage Collector that is a fundamental algorithm for the parallel and incremental garbage collector. Our algorithm is a combination of two types of fundamental algorithms, *incremental update* and *snapshot-at-beginning*, which are complementary to each other. The algorithm has both advantages of each types, those are a great efficiency of the garbage collection and ease of a consistent implementation. This algorithm can be substituted for *snapshot-at-beginning* algorithm widely used in several types of the parallel and incremental garbage collector. Measurements of this algorithm in a parallel and an incremental mark-and-sweep GC indicate that it improves the efficiency of the collection to be as well as the sequential garbage collector. Consequently, the execution time of the list processing is shortened and the range of real-time processing is extended.

1. まえがき

lispを代表とするリスト処理言語ではガーベジコレクション（GC）が必要である．従来の一括型GCは通常のリスト処理を中断して行う．この中断はリスト処理の高速化や実時間化（無停止化）の障害となる．このため，中断時間の短い世代管理型GC(generational GC）や中断が発生しない並列GC（parallel GC）や漸次GC（incremental GC）が研究されている．

並列GCおよび漸次GCはほとんどがスナップショット型（snapshot-at-beginning）と呼ばれる基本アルゴリズムを採用している[9]．このアルゴリズムは簡潔であるため，様々なタイプのGCに応用可能である．また，正当な（無矛盾な）実装も容易である．一方，単位時間あたりのゴミの回収量が少ないという欠点を持つ．GCのゴミセルの回収がリスト処理の自由セルの消費に追いつかない状態が発生しやすいため，自由セルの枯渇によるリスト処理の中断が生じる．これは同時に処理時間の延長を引き起こす．

この基本アルゴリズムの改良および効率化は，このアルゴリズムを採用する並列GCおよび漸次GC全般の高速化および実時間性の改善につながる．したがって，重要な研究課題である．

本論文ではこのスナップショット型に代わる基本アルゴリズムとして**相補型ガーベジコレクタ（Complementary Garbage Collector）**を提案する．これを応用した並列型および漸次型のマークスイープGC

† 神奈川大学理学部情報科学科
Department of Information Science, Faculty of Science, Kanagawa University
†† 慶應義塾大学大学院理工学研究科
Graduate School of Science and Technology, Keio University
††† 慶應義塾大学理工学部数理科学科
Department of Mathematics, Faculty of Science and Technology, Keio University

(mark-and-sweep GC) を汎用並列ワークステーション上の lisp 処理系に実装し，評価を行う．

2. 並列型および漸次型 GC の基本アルゴリズム

2.1 並列型および漸次型 GC の動作

並列 GC や漸次 GC は，マークスイープ GC (mark-and-sweep GC) やコピー GC (copying GC) のような一括型の GC を並列化したものである．一括型 GC が，自由セルの枯渇のたびに起動され，その間，リスト処理が完全に停止するのに対して，並列 GC や漸次 GC は，リスト処理プロセス (mutator) と GC プロセス (collector) が並列に動作する．並列 GC は，collector に独立した専用プロセッサを割り当てた完全な並列処理により GC を行う．一方，漸次 GC は 1 台のプロセッサの疑似並列処理により GC を行う．collector の動作を小さな部分に分け，mutator の一般の処理の中に埋め込む．実際には，たとえば，決められた個数のセルに対する印づけや回収作業を，cons 関数が呼ばれるたびに実行する．

並列および漸次型のマークスイープ GC では，1 つの GC サイクルは“ルート挿入 (root-insertion)”，“印づけ (marking)”，“回収 (sweeping)” の 3 つのフェーズからなる．ルート挿入フェーズでは，collector は mutator が保持する生きている (使用中の) セルを指すポインタをすべて収集する．これをルートポインタと呼ぶ．また，これを保持するものをルートと呼ぶ．実際には，mutator のレジスタやグローバル変数，スタック上の領域などがルートとなる．このようなルートの集まりをルートセットと呼ぶ．印づけフェーズでは，収集したルートポインタから到達可能なすべてのセルにマーク済みの印を付ける．回収フェーズでは，全セルを走査し，印のないセルを自由リストにつなげ，マーク済みの印を消す．コピー型 GC では，印づけおよび回収の代わりにコピー (copying) の動作を行う．

本論文では，並列および漸次型 GC アルゴリズムの基礎となる基本アルゴリズムに関して議論する．基本アルゴリズムの観点からみると，元となる GC がマークスイープ型であるかコピー型であるかは重要ではない．また，並列 GC と漸次 GC は，元になる一括型 GC が同じタイプであるならば (たとえば並列型マークスイープ GC と漸次型マークスイープ GC)，アルゴリズム上は同じものであるとみなせる．したがって，本論文では，以後，主に並列マークスイープ GC について説明する．また，並列 GC と漸次 GC を総称して**並列処理型 GC** と呼び，これらを一括して取り扱う場合に，この名称を用いることにする．

また，単位時間あたりに回収できるゴミセルの数(回収したゴミセルの数/collector の動作時間) を，その GC の**回収能力**と定義し，以後の議論に用いる．

2.2 基本アルゴリズム

一括型 GC を単純に並列化した GC を考えると，collector の印づけやコピー動作中に mutator がセルのポインタを書き換えた時，使用中のセルが誤って回収される場合がある．並列処理型 GC には，これを補償する処理が含まれている．この補償方法の違いにより，**スナップショット型** (snapshot-at-beginning)* と，**増分更新型** (incremental update) の 2 種類に分類できる[9]．

図 1 の例で考えると，collector がルート r1 につながる a～d のセルに印づけした時点で，*1 と*2 のポインタがこの順で破線で示すように書き換えられた場合，残りのルート (r2, r3) に対する印づけを行ってもf, g, h のセルには印はつかない．これらのセルは書き換え後もルートから到達可能であるにもかかわらず回収されることになる．これを防ぐためには，mutator のセルの書き換え時に，変更した部分のポインタ情報を collector へ通知する必要がある．通知されたポインタにつながるセルに対し，collector が新たに印づけを行うことで問題は解決される．

collector へのポインタの通知の方法には，1) 切断したポインタを通知する方法と，2) 新たに書き込むポインタを通知する方法の 2 種ある．図 1 の場合，1) の方法では，*1 でセル c へのポインタ，*2 ではセル f へのポインタを collector へ通知する．2) の方法では，*1 でセル f へのポインタ，*2 ではセル i へのポインタを

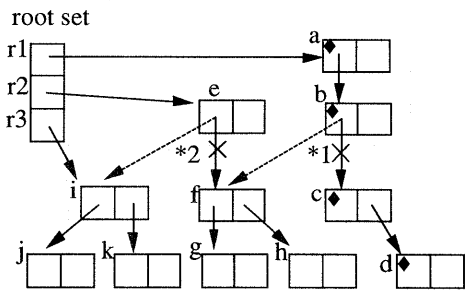


図 1 並列化の問題点
Fig. 1 Pointer rewriting problem.

* これは基本アルゴリズムの名称である．このアルゴリズムを採用する湯浅の漸次型 GC (snapshot GC)[8]にちなんで命名されたが，そのアルゴリズムそのものを指すものではない．湯浅の GC そのものを指す場合には，snapshot GC と表記することにする．

collector へ通知する．どちらの方法でも，問題となる f, g, h のセルは印づけされる．GC の回収能力や実装方法は，これらの方法に大きく依存している．

スナップショット型アルゴリズムは，1)の方法をもとに考案された方法である．増分更新型アルゴリズムは 2) の方法をもとに考案された方法である．

2.3 スナップショット型アルゴリズム

スナップショット型アルゴリズムではルート挿入は mutator を止めた状態で行う．印づけ時の mutator のセルの書き換えの際には，1) の方法で collector へ通知され，そのポインタにつながるセルはすべて印づけされる．また，ルート挿入以降に cons されるセルには，collector に回収されないようにあらかじめ印が付けられる．この結果，collector の印づけフェーズの間に，mutator がセルやルートを書き換えたりポインタを捨て去ったりしても，それらにつながるセルはその GC サイクルでは回収されない．基本的に，GC サイクル開始時のセルの状態をスナップショット写真を撮るように記録しておき，その記録にあるすべてのセルを生きているとみなして保存しておこうという考え方である．

この方法では，回収フェーズで回収されるゴミは，ルート挿入時にゴミであったセルだけである．印づけ中にゴミとなったセルは次の GC サイクルで回収される．したがって，ゴミの回収能力は一括型 GC と比べて小さくなる．このようなセルは，GC の 1 サイクルの処理時間が長いほど多く発生する．また，cons されたセルがゴミになるまでの時間は，mutator の動作の状態により変わるが，比較的短い場合が多いとされている[4]．したがって，セルを大量に消費し，印づけに時間がかかる（生きたセルが多い）ようなアプリケーションでは特に回収能力が低下する．

2.4 増分更新型アルゴリズム

増分更新型アルゴリズムでは，通常，ルート挿入は mutator を止めて行う必要はない．印づけ時の mutator のセルの書き換えの際には，2) の方法で collector に通知され，それにつながるセルはすべて印づけされる．また，印づけ時に cons され，それ以降も使用されるようなセルは，その時点でルートから到達可能なセル上のポインタを書き換えることにより，ルートから到達可能となる．その際，2)の方法では，その新しいセルへのポインタが通知されるため，そのように cons されたセルに対しても必ず印づけが行われる．したがって，cons 処理において，あらかじめセルに印をつけておく必要はない．

この方法では，切断されたポインタにつながるセルや cons された後に捨てられたセルは，ゴミとなる以前に collector に印づけされていないならば，同じ GC サイクルの回収フェーズで回収される．したがって，スナップショット型と比べて，ゴミの回収能力は大きくなる．しかし，この方法には実装上の大きな問題が存在する．

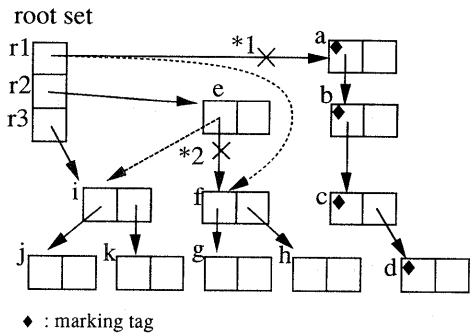


図2 ルートの書き換え
Fig. 2 Rewriting of the root.

このアルゴリズムは，ルートが 1 つであることを前提に考案された．しかし，一般のリスト処理の実装では，ルートは複数個存在するのが普通である．このアルゴリズムの正当性を維持するためには，これらのルートを通常のセルと同様に扱い，実質的にルートが 1 個となるように実装しなければならない．実際には，ルートを書き換えた時にも通常のセルの場合と同様の collector へのポインタ通知処理を行う．

図 2 に例を示す．a〜d のセルの印づけが終わった時点で，*1 のようにルート r1 を書き換え，つづいて*2 のようにセルを書き換えるとする．この場合，*1 の書き換えが起こった時に，セル f へのポインタを通知しなければならない．この通知を行わない場合には，残りのルート（r2, r3）に対する印づけを行っても f, g, h のセルには印はつかない*．同様の問題は，ルート間のポインタのコピーにおいても発生する．

しかし，このようなルートの書き換えはひんぱんに起こる．たとえば，図 2 の r1, r2 がグローバル変数であるとすると，*1 の書き換えは，

```
r1:=car(r2);
```

である．これは単なる car の処理であり，非常に頻度が高い．このように頻度が高い操作において通知処理を行うことは，mutator の非常に大きなオーバヘッドとなるため，現実的でない．レジスタやグローバル変数上で処理するポインタのコピーをセル上に残したり，cons 時のセルの印を操作するなどの実装時の工夫

* スナップショット型の場合は，*2 の書き換え時に，f へのポインタが通知されるため，このような問題は起こらない．

で，通知処理が必要なルートの数を少なくし，mutatorのオーバヘッドを小さくすることができる．しかし，この場合でも，オーバヘッドを完全になくすことはできず，また，実装が複雑になり，ゴミの回収能力が高いというこのアルゴリズムの利点を相殺する．結果的に，実装の容易さやmutatorのオーバヘッド，ゴミの回収能力というすべての面でスナップショット型アルゴリズムに及ばないものとなる．

以上のように増分更新型アルゴリズムには実装上の問題があるため，現在，実用化されているGCはほとんどが実装の容易なスナップショット型アルゴリズムを用いるGCである．Bakerのincremental copy GC[3]，松井らのSynapse GC[7]，湯浅のsnapshot GC[8]は，スナップショット型アルゴリズムを使用している．増分更新型アルゴリズムはDijkstraのGC[1]やKungらのGC[2]で使用されているが，前述の理由で実用的な実装は不可能である[11]．

3. スナップショット型アルゴリズムの効率

3.1 並列処理型GCの効率の評価法

本論文では，GC率（GC ratio）と改善率（Improvement ratio）を使用した評価法を用いる．

1つのlispプログラムを，一括型GCを持つlisp（以下seq-lisp）と並列処理型GCを持つlisp（以下para-lisp）において実行した場合の処理時間を次のように定める．

$T_{seq.gc}$：seq-lispのGC時間の合計．

$T_{seq.tp}$：seq-lispのリスト処理単体の時間の合計．

$T_{seq.total}$：seq-lispの全処理時間（$T_{seq.total}=T_{seq.gc}+T_{seq.tp}$）．

$T_{para.gc}$：para-lispのcollectorの処理時間の合計．

$T_{para.tp}$：para-lispのmutatorの処理時間の合計．

$T_{para.total}$：para-lispの全処理時間．

GC率G，改善率Iを次のように定める．

$$G=\frac{T_{seq.gc}}{T_{seq.total}},$$

$$I=\frac{T_{seq.total}-T_{para.total}}{T_{seq.total}}.$$

次の仮定のもとで，GとIの関係を求める．

モデル 動作させるlispプログラムが安定であり，定常的にゴミを出す．

並列GCの改善率

並列GCの場合，$T_{para.total}$は次のようになる．

$$T_{para.total}=\max(T_{para.tp},\ T_{para.gc}).$$

したがって，Iは，

$$I=\min\left(\frac{T_{seq.total}-T_{para.tp}}{T_{seq.total}},\ \frac{T_{seq.total}-T_{para.gc}}{T_{seq.total}}\right).$$

ここで，para-lispのmutatorの処理のオーバヘッド時間を$T_{para.oh}$とし，para-lispのGC処理はseq-lispのGC処理のr倍の処理時間が必要であるとすると，

$$T_{para.tp}=T_{seq.tp}+T_{para.oh},$$

$$T_{para.gc}=rT_{seq.gc}\quad(r>0).$$

オーバヘッド率（overhead ratio）を$O=T_{para.oh}/T_{seq.total}$とすると，並列GCの$I$は，次のようにまとめられる．

$$I=\min(G-O,\ 1-rG). \qquad (1)$$

漸次GCの改善率

漸次GCの場合，$T_{para.total}$は次のようになる．

$$T_{para.total}=T_{para.gc}+T_{para.tp}.$$

並列型と同様に，

$$T_{para.tp}=T_{seq.tp}+T_{para.oh},$$

$$T_{para.gc}=rT_{seq.gc}\quad(r>0).$$

以上より，漸次GCのIは，

$$I=(1-r)G-O \qquad (2)$$

となる．

動作効率

$1/r$はGC動作の効率を表す．並列処理型GCのゴミの回収能力が一括型GCの$1/r$倍であることを意味する．この一括型GCの回収能力に対する並列処理型GCの回収能力の比をGCの**動作効率**と定義する．Oはmutatorの処理のオーバヘッドを示す．理想的な状態，すなわち，並列処理型GCの回収能力が一括型GCの回収能力と全く同じでありmutatorのオーバヘッドが全くない場合には，$r=1$，$O=0$となる．

並列処理型GCの動作効率は，動作させるアプリケーションに依存する．式(1)および式(2)を用いると，そのGとIを実測することにより，それぞれのアプリケーションを実行した場合のGC動作効率を個別に求めることができる．実際には，GとIをプロットすることで，グラフの移動量や傾きからOとrが求まる．

3.2 スナップショット型アルゴリズムの効率

マルチプロセッサワークステーションLUNA 88k，mach OS上の並列GC lispで実験したスナップショット型GCのGとIのグラフを**図3**に示す．並列型はParaGC，漸次型はIncGCと記した．自由セルはそれぞれ250,000個ある．並列型はSynapse GCである．mutator，collectorにそれぞれ1台ずつプロセッサを割り当てた並列動作を行う．collectorはセルの枯渇

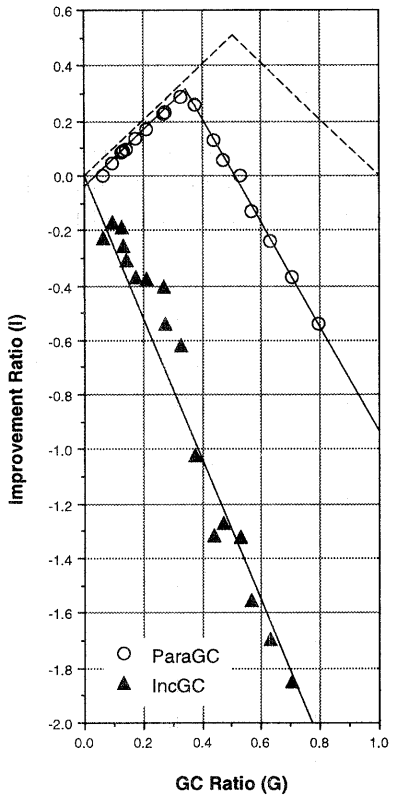


図3 GC率と改善率
Fig. 3 GC ratio and improvement ratio.

がないかぎり，常に並列に動作する．並列動作はC threadsパッケージで実現した．漸次型は，湯浅の方法と同様である．全セルの10%を切った時点でcollectorがGC処理を開始し，1セルをconsする間に，20個のセルに対し印づけまたは回収を行う．並列GCの理想曲線（$r=1$, $O=0$）を破線で付記した．漸次型の理想曲線は$I=0$の直線となる．実験に使用した関数は，結果を残さないconsを一定回数繰り返す関数eatcell*である．また，Gを変えるために，数種の長さの固定リストを作り，同じ関数を実行した．

並列型のグラフにおいて，単調増加部分（Gが0から極大部分まで）は，Hickeyらの論文[5]のstableの状態（セルの枯渇が発生しない状態），単調減少部分のうちIが正の部分はalternatingの状態（2回に1度のGCサイクルで，セルの枯渇が発生する状態），単調減少部分のうちの負の部分はcriticalの状態（毎回のGCサイクルでセルの枯渇が発生する状態）である．セルの枯渇が発生すると，mutatorの処理は，collectorが新たにセルを回収するまでの間，中断される．したがって，グラフの極大かつ最大を示す部分のGの値は，実時間処理の可能なGの上限値（**実時間処理限界点**）である．

並列型の場合，式(1)より，ゴミ回収能力の低下によりrが大きくなると，グラフのピークは原点方向に移動し，実時間処理限界点も小さくなる．また，グラフの改善率が負の部分は，一括型に比べて処理時間が長くなっている部分である．単調減少部分で改善率が負になるGの最小値（**効率改善限界点**）は，rが大きくなるほど小さな値となる．漸次型の場合，式(2)より，ゴミの回収能力の低下によりrが大きくなると，グラフの傾きが大きくなり，全般の改善率が下がる．

図3の場合のrは並列型で約2，漸次型で約3である．すなわち，ゴミの回収能力はそれぞれ一括型の約1/2および1/3に低下していることを示す．並列型の場合，実時間処理限界点は約0.35であり，理想的な曲線の0.5と比べると，低くなっていることがわかる．また，効率改善限界点は0.5となり，一括型の場合より処理時間が長くなる部分が現れる．漸次型の場合，グラフの$G=0.35$付近に不連続部分があり，Gがこれより大きい部分ではrはさらに大きい．実験でもこの部分ではセル枯渇が生じた．理論上，実時間処理限界点はグラフ上に現れないが，実際には，この不連続部分がこれを示すものと考えられる．

4. 関連研究

スナップショット型GCでは，GCサイクル中に生じたゴミは2回目の回収フェーズで回収される．しかし，印づけには毎回一括型マークスイープ法と同じ時間かかる．このため最悪の場合には，回収能力は一括型マークスイープ法の1/2となる．1回のGCサイクルの時間を短縮できれば，単位時間当たりのGCサイクルの回数が多くなり，総合的な回収能力は上がる．

Partial Marking GC[10),12)]は通常のGCサイクル（full markingサイクル）の直後に，部分的なセルに対してだけ印づけを行うGCサイクル（partial markingサイクル）を挿入したGCである．partial markingサイクルでは，直前のfull markingサイクルの間にconsされた生死の定かでないセルに対してだけ印づけを行う（この印付けをpartial markingという）．したがって，印づけが短時間で終了し，1回のサイクルの所要時間（サイクル時間）も短くなる．結果的に，直前のfull markingサイクルの間に生じたゴミを短い時間で回収することが可能となる．また，サイクル時間が短いのでこの間に生じるゴミが少なくなり，直後

```
* (defun eatcell(n)
    (cond((zerop n)nil)
         (t(cons nil nil)(eatcell(1-n)))))
```

のGCサイクルでの回収能力を上昇させることになる．

Partial Marking GCは，「full markingサイクルの回収フェーズにおいて，先の印づけフェーズでセルに付けられたマーク済みの印を消去しない」という単純な操作で実現できる．この操作によって残されたセル上の印により，その直後に行われるGCサイクルの印づけは，他のサイクルと全く同じ通常の印づけの動作を行うにもかかわらず，自動的にpartial markingとなる．

Partial Marking GCは，以上のような簡単な操作でスナップショット型マークスイープGCのゴミの回収能力を改善できる．しかし，コピー型GCへの応用が困難であるという欠点を持つ．なぜならば，マークスイープ型の「回収フェーズにおいて，マーク済みの印を消去しない」と等価な操作を，コピー型のGC上で実現することがむずかしいからである*．

5. Complementary Garbage Collector

5.1 基本的な考え方

すでに述べたようにスナップショット型アルゴリズムは，実装は容易であるが，ゴミの回収能力が低い．一方，増分更新型アルゴリズムは，ゴミの回収能力は高いが，ルートの書き換え問題があり，実装が困難である．この両者を相補的に組み合わせて，利点だけを生かすような方法があれば問題は解決する．

増分更新型アルゴリズムのルート書き換えの問題は，通常の印づけが終了した時点で補償処理を行うことで解消できる．これは，「ルートセットを調べ，未検査のセルへのポインタがあれば追加の印づけを行う」という動作である．しかし，この補償処理を同じ増分更新法で行う場合には，この補償処理中にも同様にルート書き換え問題が発生する可能性がある．このための補償処理がさらに必要となる．すなわち，ルートセットに未検査のセルへのポインタが残されているかぎり，補償処理を繰り返さなければならない．このようなセルは，mutatorがひんぱんに行うルートの書き換え処理やcons処理で発生する．したがって，補償処理の印づけの間もmutatorがリスト処理をつづけている場合には，回収フェーズへの移行が困難となり，補償処理の長時間化を招く．mutatorがすべての自由セルを消費するまで補償処理が終了しない場合も考えられ，現実的ではない．一方，補償処理中にmutatorを停止させておく場合は，上記問題は起こらず，補償処理は簡単になる．しかし，未検査のセルの量が多い場合には追加の印づけに時間がかかり，mutatorの長時間の停止を引き起こすことになる．これは，実時間性を損ない，ゴミ回収能力も低下させる．以上のように，増分更新型の動作だけで行う補償処理は実際的でない．

Complementary GCは，増分更新型アルゴリズムのルート書き換え問題の補償処理をスナップショット型アルゴリズムで行うものである．Complementary GCは，増分更新型アルゴリズムでルート挿入，印づけを行った後に，スナップショット型アルゴリズムでルート挿入，印づけ，回収を行う．この5つのフェーズが1つのGCサイクルを形成する．増分更新型の部分は通常の増分更新型アルゴリズムの1GCサイクルから回収フェーズを取り除いたものである．この間にmutatorがポインタの書き換えを行った場合は，2.2節の2)の方法，すなわち，上書きする新しいポインタをcollectorへ通知する．後半に行うスナップショット型アルゴリズムは，単独で動作する場合のスナップショット型アルゴリズムと同じである．この間のポインタの書き換えは，1)の方法，すなわち，切断される古いポインタを通知する．また，この間に行われるcons処理では，セルにマーク済みの印をつけておく．後半に行うスナップショット型アルゴリズムのルート挿入および印づけが前半の増分更新型アルゴリズムの補償処理として機能する．

前半の増分更新型アルゴリズムの印づけフェーズ後にルートセットに残される可能性がある印づけされていないセルへのポインタは，直後に行うスナップショット型アルゴリズムのルート挿入で必ず検査され，印づけが行われる．したがって，ルート書き換え問題は補償される．また，スナップショット型アルゴリズムでは基本的にルート書き換え問題は生じない．このため，補償処理である後半のスナップショット型の動作はmutatorの動作中に行っても問題は生じず，繰り返し実行する必要もない．つまり，増分更新型だけで行う補償処理のように停止性に問題を生じたりmutatorの処理に長時間の中断が生じることはない．このように2つのアルゴリズムを相補的に組み合わせることで，実時間性やゴミ回収能力を損なわない補償処理を容易に実現できる．

スナップショット型アルゴリズムが正当であるならば，前半の増分更新型の部分で，後のスナップショット型の動作を阻害するような問題が起こらないかぎ

* 旧空間（from-space）と新空間（to-space）の他に，GCのコピー動作中のconsのためのセルの空間を別にもう1つ設けたり，それらのセルに識別タグを付加したりすることで実現できる．しかし，どちらもマークスイープ型と比べるとかなり複雑なアルゴリズムとなってしまう．

り，Complementary GC は正当である．このような問題は，スナップショット型アルゴリズム部分のルート挿入の時点で印づけ済みのセルから印のついていない生きたセルへのポインタが存在する場合にだけ生じる．スナップショット型アルゴリズム部分の印づけでは，このようなセルには到達できないからである．このようなセルはセルのポインタの書き換えを行った場合に生じる．しかし，増分更新型アルゴリズムでは，セルのポインタの書き換えを行った場合，上書きする新しいポインタは collector へ通知される．上のような印のついていないセルへのポインタは collector へ通知され，すべて印づけが行われる．したがって，増分更新型部分の印づけが終わった時点では，印づけ済みのセルからつながる印のついていない生きたセルは存在しない．

効率について考えると，それぞれのアルゴリズムを単独で動作させる場合と同様に，前半の増分更新型の印づけフェーズの間にゴミになったセルは後の回収フェーズで回収される可能性があるが，後半のスナップショット型の印づけフェーズの間にゴミになったセルは直後の回収フェーズでは決して回収されない．したがって，後半のスナップショット型の印づけフェーズの時間が短いほど，回収フェーズで回収できるゴミの数が多くなる．総合的なゴミの回収能力は，それぞれの型の印づけフェーズの処理時間の比率によって決まることになる．最悪の場合はスナップショット型アルゴリズムの効率と等しくなり，最良の場合は増分更新型アルゴリズムの効率と等しくなる．一方，後半のスナップショット型の動作で印づけされるセルは，前半の増分更新型のルート書き換え問題で生じるセルだけである．これらは，前半の印づけフェーズの間に，ポインタを切断されたり，新たに cons されることにより印づけフェーズが終了した時点でルートセットだけから到達できるようになったセルである．このようなセルの数は実行するアプリケーションにより変化するが，一般に，生きているセル総数に比べると十分少ない．したがって，総合的なゴミ回収能力は単独の増分更新型アルゴリズムの回収能力に近くなると推定できる．

```
type t_pointer = 1..M ;
     t_root = 1..R ;
     t_color = ( black, white, offwhite ) ;
     t_mode = ( incremental, snapshot, idle ) ;
     t_cell = record
              car, cdr : t_pointer ;
              color :    t_color
              end ;

var  cell : array[ t_pointer ] of t_cell ;
     root : array[ t_root ] of t_pointer ;
     FREE_LEFT : t_pointer ;
     FREE_RIGHT : t_pointer ;
     REQUEST_PUSH : t_mode ;
     CONS_COLOR : t_color ;
```

図4 Complementary GC アルゴリズムのデータ構造
Fig. 4 Complementary GC algorithm (data structure).

5.2 Complementary GC のアルゴリズム

Complementary GC のアルゴリズムのデータ構造，collector のアルゴリズム，mutator のアルゴリズムを，図4，図5，図6に示す．pascal 風の言語で記述し

```
Procedure 1_GC_Cycle ;
var i : t_pointer ;
begin
  {Root-insertion and Marking phase
              of incremental-update}
  REQUEST_PUSH := incremental ;
  for i := 1 to R do
    mark(root[i]) ;
  while << the stack is not empty >> do
      begin
        i := pop ;
        mark(i) ;
      end ;

  {Root-insertion phase
       of snapshot-at-beginning}
  suspend_mutator ;
  REQUEST_PUSH := snapshot ;
  CONS_COLOR := offwhite ;
  for i := 1 to R do
    push( root[i] ) ;
  resume_mutator ;

  {Marking phase of snapshot-at-beginning}
  while << the stack is not empty >> do
      begin
        i := pop ;
        mark(i) ;
      end ;

  {Sweeping phase}
  REQUEST_PUSH := idle ;
  for i := 1 to M do
      if cell[i].color = white then
        begin
          cell[i].color := offwhite ;
          cell[i].car := f ;
          cell[FREE_RIGHT].cdr := i ;
          FREE_RIGHT := i ;
        end
      else if cell[i].car <> f then
        cell[i].color := white ;
  CONS_COLOR := white ;
end

procedure mark( j : t_pointer ) ;
  begin
    while ( j <> NIL) and
          (cell[j].car <> f) and
          (cell[j].color <> black) do
      begin
        cell[j].color := black ;
        mark(cell[j].car) ;
        j := cell[j].cdr ;
      end
  end
```

図5 Complementary GC アルゴリズム (collector)
Fig. 5 Complementary GC algorithm (collector).

```
{replacing car pointer of cell[root[m]]
      to root[n] m, n : t_root }
procedure LPa  ;
  begin
    if REQUEST_PUSH = incremental then
      push(root[n])
    else if REQUEST_PUSH = snapshot then
      push(cell[root[m]].car) ;
    cell[root[m]].car := root[n]
  end

{cons( root[m], root[n]) m, n : t_root}
procedure LPc  ;
  begin
    while FREE_LEFT = FREE_RIGHT do {waiting} ;
    root[R] := FREE_LEFT ;
    FREE_LEFT := cell[FREE_LEFT].cdr ;
    cell[root[R]].car := root[m] ;
    cell[root[R]].cdr := root[n] ;
    if   CONS_COLOR <> offwhite then
      cell[root[R]].color := white ;
    root[m] := root[R]
  end
```

図6 Complementary GC アルゴリズム (mutator)
Fig. 6 Complementary GC algorithm (mutator).

た．このアルゴリズムは，並列処理型マークスイープGCにComplementary GCを組み込んだものである．

このアルゴリズムは，次のデータ構造を持つ．セルは2つのポインタフィールド (car, cdr) とカラーフィールド (color) を持つ．セルの総数はMである．ルートポインタは総数Rであり，配列rootに格納されているものとする．自由リストは1本のリストである．FREE_LEFTは自由リストの先頭のセルを，FREE_RIGHTは最後のセルを指す．また，自由リストのセルは，colorはoffwhiteにセットされ，carにはfという特殊なポインタが格納されている*．2つのフラグ，REQUEST_PUSH, CONS_COLORには，それぞれidle, whiteの初期値が設定されているものとする．

mutatorとcollectorの通信はスタックを用いて行う．このスタックの操作(push(), popおよびstack emptyのチェック)は非可分処理であるとする．また，suspend_mutatorはmutatorにリスト処理を中断させる手続きである．resume_mutatorはmutatorにリスト処理を再開させる手続きである．

LPaはポインタの書き換え処理 (rplaca) を表す．rplacdも同様である．LPcはセルの生成 (cons) を表す．それぞれ，root[m], root[n]のポインタを処理し，root[m]に値を返すものとする．

* このfという特殊なポインタは，自由セルとcons直後のセルとを区別するために用いる．colorがoffwhiteであり，かつ，carがfであるセルは自由セルである．colorがoffwhiteであるが，carがfではないセルは直前にconsされたセルである．これにより，スナップショット型動作の際のLPcの処理を簡略化している．

このアルゴリズムをスナップショット型アルゴリズムと比較すると，図5では，Root-insertion and Marking phase of incremental-updateの部分と，REQUEST_PUSHおよびCONS_COLORの2つのフラグに関する操作が付加されている．また，図6の手続きLPaのポインタの報告部分，手続きLPcのconsセルのcolorをwhiteに変える部分が変更されている．これらの付加および変更部分において，実装上問題になる部分はない．

実装時におけるmutatorのオーバヘッドについて考えると，手続きLPaに比べて手続きLPcの実行頻度は非常に高いため，この部分にオーバヘッドが生じると，mutatorの総合的なオーバヘッドが大きくなり，問題である．手続きLPaのポインタの報告部分の変更は，スナップショット型アルゴリズムの同じ部分より，場合分けが増えただけであり，大きなオーバヘッドとはならない．一方，consセルのcolorを変える操作は，collectorが常に並列動作を行う(停止しない)並列型のスナップショット型アルゴリズムでは全く不要である．したがって，このタイプのGCではオーバヘッドが生じることになる．しかし，自由セルの残量が

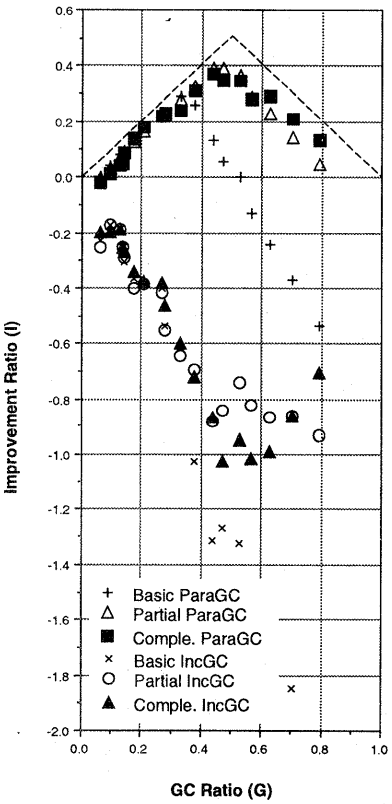


図7 Complementary GCの改善率
Fig. 7 Improvement ratio of complementary GC.

多いときにcollectorの動作を停止するタイプの並列GC（たとえばConditional Invoke GC[10]）や漸次GC（たとえばsnapshot GC[8]）の場合には，GCの回収能力を低下させないために，もともと同様のcolorの操作が必要である．このようなGCに応用する場合には，オーバヘッドは発生しない．

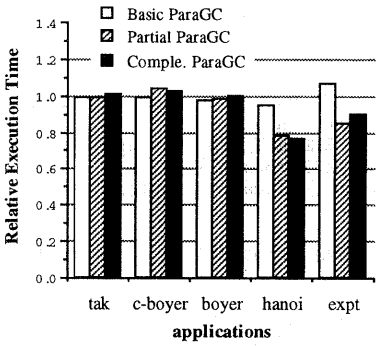
以上のように，このアルゴリズムはスナップショット型アルゴリズムを容易に代替できる．現在すでに実用化されている並列処理型GCに少ない変更で組み込むことができる．また，ここではマークスイープGCへの応用を示したが，同様に，増分更新型のコピー動作とスナップショット型のコピー動作*を相補的に組み合わせることでコピー型のComplementary GCを実現できる．

6. 評　価

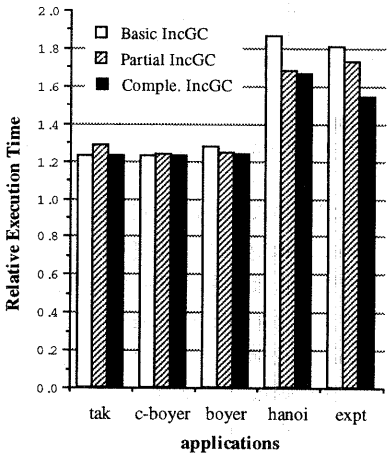
図3と同じ並列GC lispに同じ条件で実装し実測したGとIのグラフを**図7**に示す．従来型（スナップショット型）アルゴリズムの並列GCをBasic ParaGC，漸次GCをBasic IncGCと示した．同様にPartial Marking GCの並列GCをPartial ParaGC，漸次GCをPartial IncGCと示し，また，Complementary GCの並列GCをComple. ParaGC，漸次GCをComple. IncGCと示した．実装上，問題は生じなかった．また，アプリケーションの実行でも問題は生じていない．

並列型の場合，Partial Marking GCとほとんど同じ回収能力を示した．rの値は約1となり，ゴミの回収能力は一括型と同程度であることがわかる．これは，スナップショット型の2倍の回収能力を持つことを示す．この結果，実時間処理限界点は0.5付近まで上昇し，実時間性が向上していることがわかる．これは，全処理時間の半分がGCに費やされるようなリスト処理でも実時間処理が可能であることを示す．また，効率改善限界点は0.8付近まで上昇しており，一括型GCより処理時間が長くなってしまうような場合が少なくなった．理想の曲線（破線）に近づいていることがわかる．また，オーバヘッド率Oは，3つの方法ともに約0.1であり，大きな差異は認められない．

また，漸次型でも，Gが0.4を越える部分から大きく改善されている．また，グラフには現れないが，実際の計測では実時間処理限界点が並列型と同様，0.35付近から0.5付近まで上昇したことが確認できた．



(a) Parallel GC



(b) Incremental GC

図8 アプリケーション別の実行時間
Fig. 8 Relative execution time.

mutatorのオーバヘッドを観察するために，アプリケーション別の実行時間を**図8**に示した．一括型マークスイープGCを持つ処理系の実行時間を1とした場合の相対時間で示した．tak, boyerはGabriel[6]のベンチマークにあるものを用いた．c-boyerはコンパイルしたboyerである．hanoiはハノイの塔の計算を行うものであり，15の円盤の移動を求める計算を行った*．exptは累乗の計算を行う組み込み関数であり，(expt 10 4500)を実行した．多倍長の乗算が繰り返され，多くのゴミセルを生成する．tak, c-boyer, boyer, hanoi, exptのGC率は，順に9.1%, 10.9%, 17.5%, 43.3%, 48.8%となっている．

* 増分更新型のコピー動作では，mutatorがその間に行うconsは，旧空間（from-space）のセルを使用する．スナップショット型のコピー動作は，consは新空間（to-space）のセルを使用する．ポインタ書き換え時の報告方法はマークスイープ型と同じである．

```
* (defun hanoi(n p1 p2 p3)
   (cond((zerop n)nil)
        (t(append(hanoi(1-n)p1 p3 p2)
                 (cons(list 'enban n 'wo p1
                 'kara p3 'he 'utsusu)
                      (hanoi(1-n)p2 p1 p3))))))
```

並列型では，GC率が低い場合には原理的に処理時間は改善されない．GC率が低い3つのアプリケーションでは，Complementary型，Partial Marking型とも改善がみられない．逆に，これらの型は従来型と比べて実行時間が長くなっているのがわかる．これは，先に述べたポインタ書き換え時の報告処理やcons時のcolor操作の追加処理のオーバヘッドであると考えられる．ポインタ書き換え処理がほとんどないtakでは，Complementary型だけが悪い値を示している．これはcons時の追加処理のオーバヘッドであると考えられる．ポインタ書き換え処理が頻繁に行われるboyerやc-boyerでは，Complementary型，Partial Marking型とも悪くなっている．これは主にポインタ書き換え時の報告処理によるものであると考えられる．GC率が高いhanoiおよびexptでは，Complementary型，Partial Marking型とも大きく改善されている．特にexptでは，従来型が一括型より処理時間が延長されているが，逆に2つの型では短縮されている．グラフには現れないが，実際の計測時にはhanoiではすべての並列GCでセル枯渇が生じていた．exptでは従来型でセル枯渇が生じていたが，Partial Marking型，Complementary型ではセル枯渇は生じていなかった．

漸次型では，GC率が大きくなるにしたがって効果が大きくなる．tak，c-boyerは従来型とComplementary型がほぼ等しく，それ以外では，Complementary型がもっとも処理時間が短いことを示している．漸次型では，すでに述べたように，cons時の追加処理のオーバヘッドがないため，アルゴリズム本来の回収能力がそのまま結果となって現われたものと考えられる．GC率の大きなアプリケーションでは，非常に大きく改善されているのがわかる．exptは，測定時の観測でもセル枯渇による中断は生じなかった．すなわち，従来型の約80%の時間で処理が終了し，かつ，実時間処理が可能であるという特に良い結果を示している．

7. まとめ

本論文では，並列処理型GCの基本アルゴリズムであるスナップショット型アルゴリズムおよび増分更新型アルゴリズムの問題点を明らかにし，その改良法としてComplementary GCを提案した．Complementary GCは，従来の2つのアルゴリズムを相補的に組み合わせたものである．ゴミの回収能力の高さと実装の容易さという両アルゴリズムの利点を合わせ持つことを示した．

並列型および漸次型マークスイープGCへ組み込み，評価を行った結果，Complementary GCのゴミの回収能力はPartial Marking GCとほぼ同等であることがわかった．特に並列型GCでは，従来型の2倍の回収能力を示し，一括型GCとほぼ同等まで改善されることがわかった．最も重要なポイントは，これにより処理時間が短縮されるだけでなく，実時間処理の限界点が上昇する点にある．

Complementary GCは，多くの並列処理型GCの基本アルゴリズムとして採用されているスナップショット型アルゴリズムを代替することができる．これにより，これら処理系の処理時間および実時間性を直ちに改善できる．

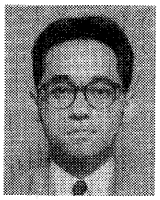
本論文では主に並列処理型マークスイープGCへの応用について報告したが，並列処理型コピーGCへの組み込み，実証についても計画している．

参考文献

1) Dijkstra, E. W. et al.: On-the-fly Garbage Collection, An Exercise in Cooperation, *Lecture Notes in Computer Science*, No. 46, pp. 43-56, Springer-Verlag, New York (1976).
2) Kung, H. T. and Song, S. W.: An Efficient Parallel Garbage Collection System and Its Correctness Proof, *IEEE Symp. Foundations of Computer Science*, pp. 120-131 (1977).
3) Baker, H. G.: List-Processing in Real Time on a Serial Computer, *Comm. ACM*, Vol. 21, No. 4, pp. 280-294 (1978).
4) Lieberman, H. et al.: A Real-Time Garbage Collector Based on the Lifetimes of Objects, *Comm. ACM*, Vol. 26, No. 6, pp. 419-429 (1983).
5) Hickey, T. and Cohen, J.: Performance Analysis of On-the-fly Garbage Collection, *Comm. ACM*, Vol. 27, No. 11, pp. 1143-1154 (1984).
6) Gabriel, R. P.: *Performance and Evaluation of Lisp Systems*, pp. 81-92, 116-135, The MIT Press, Cambridge (1985)
7) Matsui, S. et al.: SYNAPSE: A Multi-microprocessor Lisp Machine with Parallel Garbage Collector, *Lecture Notes in Computer Science*, No. 269, pp. 131-137, Springer-Verlag (1987).
8) Yuasa, T.: Real-Time Garbage Collection on General-Purpose Machines, *The Journal of Systems and Software*, Vol. 11, No. 3, pp. 181-198 (1990).
9) Wilson, P. R.: Uniprocessor Garbage Collection Techniques, *Lecture Notes in Computer Science*, No. 637, pp. 1-42, Springer-Verlag (1992).
10) Tanaka, Y. et al.: Parallel Garbage Collection by Partial Marking and Conditionally In-

voked GC, *Proceedings of the International Conference on Parallel Computing Technologies* (*Obninsk, RUSSIA*), Vol. 2, pp. 397-408 (1993).
11) 湯浅太一：実時間ゴミ集め，情報処理，Vol. 35, No. 11, pp. 1006-1013 (1994).
12) Tanaka, Y. et al.: Partial Marking GC, Parallel Processing: CONPAR 94-VAPP VI, *Lecture Notes in Computer Science*, No. 854, pp. 337-348, Springer-Verlag (1994).

(平成6年4月21日受付)
(平成7年5月12日採録)



田中　良夫（正会員）

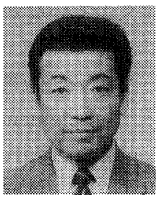
昭和40年生．昭和62年慶應義塾大学理工学部数理科学科卒業．平成元年3月同大学大学院前期博士課程修了．平成7年3月同大学院後期博士課程単位取得退学．同年4月より中西研究室研究員．Lisp処理系（主にガーベッジコレクション），並列処理，分散処理の研究を行っている．ACM会員．

前田　敦司（正会員）

昭和37年生．昭和61年慶應義塾大学理工学部数理科学科卒業．昭和63年3月同大学大学院前期博士課程修了．平成6年3月同大学院後期博士課程単位取得退学．慶應義塾大学研究生．Lisp処理系，コンパイラ，プログラミング言語理論，情報理論に関心を持つ．ACM会員．

松井　祥悟（正会員）

昭和34年生．昭和57年慶應義塾大学工学部数理工学科卒業．平成元年同大学大学院博士課程単位取得退学．平成元年神奈川大学理学部助手，平成7年専任講師．工学博士．Lisp処理系（Lispマシン，並列GC）の研究に従事．電子情報通信学会，ACM各会員．



中西　正和（正会員）

昭和41年慶應義塾大学工学部卒業．昭和44年慶應義塾大学工学部助手．平成元年慶應義塾大学理工学部教授．工学博士．昭和42年，日本初の実用Lisp処理系を作成．以後，記号処理言語，人工知能言語等の研究に従事．昭和50年プログラムの性質の自動証明系，昭和57年Lispマシン SYNAPSE の開発など．電子情報通信学会会員．情報処理学会プログラミングシンポジウム委員会幹事長．