世代別ごみ集めでのプログラムの文脈に基づくシンボルの配置法

小林広和[†] 寺田実[†]

世代別ごみ集めを実装する場合には，古い世代でのオブジェクトの書き換えの局所性が問題となる．本論文では，オブジェクトの書き換えの局所性を向上させるための方法として，プログラムの文脈に基づいて GNU Emacs のシンボルオブジェクトを配置する手法を提案する．

GNU Emacs は動的スコープをもち，シンボルは浅い束縛によって実装されているのでシンボルの値セルの書き換えが多量に発生する．プログラムの静的解析を行なうことによって，書き換えられる可能性があると判断されたシンボルを特に考慮して配置する．この方法を用いることより，世代別ごみ集めの古い世代での書き換えの局所性が向上することを実測によって示す．

A Method for Symbol Layout Based on Static Code Analysis for More Efficient Generational Garbage Collection

HIROKAZU KOBAYASHI[†] and MINORU TERADA[†]

We propose a method for symbol layout based on probability of rewrite which increases the efficiency of generational garbage collection. Our testbed, GNU Emacs, is a Lisp dialect with dynamic scoping and shallow binding, which tends to make frequent updates of symbols' value cells. We present a method for compile time detection of mutable symbols and treat mutable symbols separately. This technique can improve symbols' rewrite locality in the old generation.

1. はじめに

筆者は，論文 3) において世代別ごみ集め [4),8)] を GNU Emacs に実装した．その実装では，古い世代から若い世代への参照を検出するために，ダーティビット情報 [6)] を用いた．ダーティビット情報を利用した世代間参照の検出法は，ページマーキングの一種で，ハードウェアによって検出を行なう．したがって，オブジェクトの書き換えを検出するために必要となるソースコードの変更が少ないという利点がある．Emacs Lisp のように動的スコープを持ち，シンボルの値の実装に浅い束縛 [1)] を用いている処理系では，シンボルの値スロットの書き換えが多くなると考えられる．そのような状況でソフトウェアによって世代間参照の検出を行なった場合は，オブジェクトに対する書き換えがあるたびに書き換えを検出するコードが実行されるので，ハードウェアを用いた場合に比べ，通常処理の実行速度の低下が大きい．したがって，ダーティビット情報を用いた，ハードウェアによる世代間参照の検出が有利であると考えられる．

ダーティビット情報を用いた方法では，書き換えの検出を領域単位で行なう．この方法では，検出の単位となる領域の大きさがマシンのメモリページの大きさ☆となり，オブジェクトの大きさに比べて大きい．したがって，ある領域への書き換えがあった時に，多くのオブジェクトを次回の若い世代のごみ集めのルート検査の対象としなければならない欠点がある．ルート検査のコストを少なくするためには，書き換えられるページを少なくするとよい．そのためには，書き換えの局所性を向上させる必要があり，オブジェクトの配置が重要となる．

GNU Emacs に実装した世代別ごみ集めでは，シンボルページについては書き換えの局所性が低い．つまり，書き換えが行なわれたシンボルセルが多くのページに分散している．このために，ルート検査の対象となる領域が大きくなり，ごみ集めにかかる時間の増加

† 東京大学大学院工学系研究科
School of Engineering, University of Tokyo

☆ Intel の CPU を用いたマシンや Sun の Super SPARC などの一般的なマシンで 4kByte である．

をまねいている．

したがって，シンボルの書き換えの局所性を向上させれば，書き換えの起こるページが減少するので，世代別ごみ集めでのルート検査のコストを削減することができる．そのために，プログラムの静的解析の結果に基づいて，書き換えの可能性の高いシンボルを同じページに配置する方法を提案し，その手法の有効性を実測によって示す．

本論文の構成は次のとおりである．第2章においてGNU Emacsの特徴と筆者がGNU Emacsに実装した世代別ごみ集めについて説明する．第3章においてオブジェクトの配置と世代別ごみ集めのライトバリアの関係について述べる．第4章において本論文の手法を実現するためにEmacs Lispのバイトコンパイラと Emacs Lispの処理系に対して拡張した機能を述べ，それらの実装法を示す．第5章において本論文の手法を用いることで，GNU Emacsでのシンボルに対する書き換えの局所性がどのように改善されたかをいくつかの計測結果によって示し，その評価を行なう．第6章において本論文に関連する研究について述べる．第7章において本論文で行なったオブジェクトの配置法に関する議論を行なう．第8章において，本論文の手法で得られた結果についての結論をまとめる．

2. 世代別ごみ集めを実装したGNU Emacs

GNU Emacsバージョン19.28ベースのmule2.3に対して，筆者は世代別ごみ集めの実装を行なった．

GNU EmacsはEmacs Lispという Lispの方言で記述されている．Emacs Lispを他の Lisp処理系と比較した時の特徴として次のような点が挙げられる．

- バイトコードインタプリタとインタプリタを備えた Lisp処理系である．
- シンボルの値の実装に浅い束縛を用いている．
- 動的スコープを持つ．

筆者がGNU Emacsに対して行なった世代別ごみ集めの実装は，次のようなものであった．

- 世代数は2世代である．
- 世代別ごみ集めを実装する時に必要な，古い世代から若い世代への参照を検出する方法としてダーティビット情報を用いた．
- 若い世代のごみ集めを1回生き残ったオブジェクトは，すべて古い世代へと殿堂入り（tenuring）する．この結果，若い世代のごみ集めが行なわれた直後には，若い世代には生きているオブジェクトがない．
- 古い世代では，シンボル，コンス，ストリングなどのオブジェクトについては，ブロックを確保し，1つのブロックの中には同じ種類のオブジェクトだけになるように配置する．ブロックの大きさは一般的なマシンでのページサイズである4kバイトとし，ブロックの開始アドレスは4kバイトの倍数のアドレスにする．この方式はブロックサイズが異なること以外は，もともとGNU Emacsに実装されていたごみ集めと基本的には同じ方式である．
- 古い世代のごみ集めは基本的にはマークスイープ方式を用いる．

GNU Emacsの大きな特徴として，シンボルの値の実装に浅い束縛を用いていることがあげられる．浅い束縛では，あるシンボルの値の束縛が変化した時に，そのシンボルの値スロットを現在の値の束縛に書き換えるという方法を用いる．その結果，各シンボルに対して常に同じ場所を参照することによって，そのシンボルの値を得ることができるので，値の参照を速くできるという利点がある．つまり，浅い束縛では束縛が起こるたびにシンボルの値スロットを書き換えるので，シンボルの値スロットの書き換えが頻繁に起こると考えられる．実際にGNU Emacsで計測を行なうと，若い世代のごみ集めが実行されてから次に若い世代のごみ集めが実行されるまでの間に，書き換えられた古い世代のシンボルブロックの割合は約45〜50%程度であった．これは，一般のオブジェクトブロックが書き換えられる割合の約25%程度に比べて大きい．このことによって，浅い束縛を用いているためにシンボルの値スロットの書き換えが頻繁に起こることを確認できる．

3. 世代別ごみ集めとオブジェクトの配置

世代別ごみ集めは若い世代だけをごみ集めすることで，1回のごみ集めにかかる時間を短縮している．若い世代だけをごみ集めするためには，古い世代のオブジェクトから若い世代のオブジェクトへの参照を検出し，検出された参照を若い世代のごみ集めを行なう時のルートとして扱う必要がある．古い世代から若い世代への参照は，古い世代を書き換えた時に作成される可能性があり，また古い世代を書き換えた時にしか作成されない☆ので，古い世代の書き換えを検出することによって，そのような参照を検出することができる．このように古い世代に対する書き換えを検出すること

☆ 若い世代への参照を持ったオブジェクトの古い世代への殿堂入りによっても作成されるが，これは容易に検出することができる．

で，古い世代から若い世代への参照を検出する方法をライトバリアと呼ぶ．

ライトバリアには次のような方法がある．

- リメンバードセット [8)]
- カードマーキング [7)]
- ページマーキング [2),6)]

これらの方法の中で，ページマーキングとカードマーキングはヒープを同じ大きさのいくつかの領域に分割し，その分割した領域に対しての書き換えを検出するという，領域を単位とした検出方法である．ページマーキングやカードマーキングでは，古い世代のある領域の書き換えがあれば，その領域は若い世代への参照を含む可能性があるので，若い世代のごみ集めを実行する時にルート検査を行なう対象とする．

カードマーキングでは書き換えの検出をソフトウェアによって行なうので，検出の単位となる領域☆の大きさは，オブジェクトの大きさを考慮して最適な値を実装者が任意に選択することができる．しかし，ページマーキングでは，検出の単位となる領域の大きさは，実行するマシンのメモリページ単位であり，自由に選択することができない．メモリページの大きさは，一般的なマシンで 4096 バイトであることが多く，そのために最悪の場合は 1 ワードの書き換えが存在しただけで 1024 ワードの検査を行なう必要が生じる．したがって，同一のページに書き換えの対象となるオブジェクトを多く配置することにより，書き換えの局所性を向上させることが，ページマーキングを用いた世代別ごみ集めを行なう上で都合がよい．同様に，カードマーキングを行なう場合においても，書き換えの対象となるオブジェクトを同じカードの上に配置した方がルート検査のコストを減らすことができる．また，リメンバードセットを用いている場合にも，ごみ集めの実行時に検査するページが少ない方が，ページングの動作上有利となる．さらに，書き換えの局所性を向上させることにより，ごみ集めだけでなく，通常処理を行なっている時のページフォルトの減少，キャッシュミスの減少などの効果もあると考えられる．したがって，古い世代のオブジェクトの配置を工夫してオブジェクトの書き換えの局所性を向上し，書き換えの起こるオブジェクトを同一のページに配置するのは重要なことである．

本論文では，書き換えの局所性を考慮したオブジェクトの配置法の例として，プログラムの文脈の静的解析によるシンボルの配置法について取り上げる．プロ

☆ カードと呼ばれる．

```
(defun silly-loop (n)
  (let ((t1 (current-time-string)))
    (while (> (setq n (1- n))
              0))
    (list t1 (current-time-string))))
```

図 1 Emacs Lisp のプログラム例
Fig. 1 An example program of Emacs Lisp.

グラムの文脈を静的に解析することにより，書き換えの対象となりうることが分かるシンボルを，mutable シンボルと名付けることにする．mutable シンボルに対してプログラムの実行時に書き換えが行なわれるかどうかは，その書き換えを行なうコードをプログラムが実行するかどうかによる．したがって，プログラムの実行時にそのシンボルに対する書き換えが起こることを，必ずしも保証できるわけではない．

ここで，図 1 の Emacs Lisp のソースコードを例として取り上げる．このコードを実行すると，シンボル `n` とシンボル `t1` の値スロットが書き換えられる．この `n` と `t1` が書き換えられることは，静的にプログラムを解析することによっても分かる．つまりこれらは，mutable シンボルである．mutable シンボルを普通のシンボルとは特別に扱うことにする．そして，世代別ごみ集めを実装する場合に，古い世代において，mutable シンボルは mutable シンボルだけを持つページに配置することとする．このようなシンボルの配置法によって書き換えの局所性が向上するので，書き換えが行なわれた総ページ数が減少する可能性が高い．

以上述べた書き換えの局所性を考慮したシンボルの配置によって，世代別ごみ集めの性能の向上が可能であることを示すために，GNU Emacs において実装を行なった．

4. 実　　装

実装には次の二つのことが必要であった．

- Emacs Lisp のバイトコンパイラの変更
- Emacs Lisp 処理系の変更

4.1 バイトコンパイラの変更

GNU Emacs は Lisp のコードをバイトコンパイラによりバイトコードにコンパイルしてファイルに保存しておき，必要になった時にバイトコードを持っているファイルを動的に読み込む機能を備えている．また，ユーザが任意のバイトコードファイルを GNU Emacs に読み込ませ，実行させる機能も備えている．バイトコードファイルは一般の Lisp 式を含んでいても良いが，バイトコード関数オブジェクトを含む点において

一般の Lisp のソースファイルとは異なる．

このバイトコンパイラで Lisp のソースコードからバイトコードファイルを作成する時に，特別なシンボル領域に配置するような Lisp の関数呼び出しを，バイトコードファイルの先頭に付け加えることにした．特別なシンボル領域にシンボルを配置する関数は，**intern-mutable** と命名した．**intern-mutable** は，組み込み関数である **intern** と使い方は完全に同じである．しかし，intern されたシンボルが特別な領域に配置されるという点だけが，組み込み関数の **intern** と異なっている．

例えば，図 1 のプログラムをバイトコードにコンパイルしたものを逆アセンブルすると，**図 2** のバイトコードになる．このバイトコードの中で **varset** がシンボルの値を変更するコードである．さらに **varbind** がシンボルの値を束縛するコードである．これらのコードの引数となっているシンボルは，書き換えの対象となっている．

第 4.2 節において説明するとおり，一度 intern されたシンボルは intern された領域から移動されない．そのために，バイトコードを読み込む順序によっては，普通のシンボル領域に作成されたシンボルが，他のバイトコードファイルでは mutable シンボルであるとされている場合がある．**varref** はシンボルの値を参照するコードであり，値が参照されるシンボルは値を持つので，値スロットの書き換えが起こる可能性が高いと考えられる．実際に，**varref** の引数のシンボルを mutable シンボルとするものと，普通のシンボルとするものの 2 種類のコンパイラを作成し性能の比較を行なった．その場合に，**varref** の引数を mutable シンボルとしたものの方が性能が良いので，それも mutable シンボルとすることにした．

これら **varset**，**varbind**，**varref** バイトコードの引数となっているシンボルが mutable シンボルなので，**intern-mutable** で intern することにする．

4.2 Emacs Lisp 処理系の変更

Emacs Lisp 処理系に，**intern-mutable** という組み込み関数を新たに付け加えた．**intern-mutable** で intern されたシンボルは，普通のシンボルとは違い特別な領域に配置される．その特別な領域には，**intern-mutable** で intern されたシンボルしか配置されず，mutable シンボル領域と呼ばれる．**intern-mutable** で intern されたシンボルがすでに通常の **intern** によって intern されていた場合には，そのシンボルを mutable シンボル領域に新たに移動することはせずに普通のシンボルの領域に配置したまとする．

```
byte code for silly-loop:
  args: (n)
0 constant  current-time-string
1 call   0
2 varbind   t1
3:1 varref   n
4 sub1
5 dup
6 varset   n
7 constant  0
8 gtr
9 goto-if-not-nil 1
12 varref   t1
13 constant  current-time-string
14 call   0
15 unbind   1
16 list2
17 return
```

図 2 図 1 のプログラムをバイトコンパイルした時のアセンブラコード
Fig. 2 Assembly code of the byte-compiled **Fig.** 1 program.

従来の世代別ごみ集めの実装では，新たに作成されたシンボルは，若い世代にアロケートされ，そして，若い世代のごみ集めが行なわれた時に生き残ったシンボルだけが古い世代に殿堂入りする．しかし，**intern-mutable** を実装するにあたって，**intern-mutable** が新しく作成したシンボルは，若い世代にアロケートせずに，直接古い世代にアロケートするようにした．その理由は次のとおりである．

- 筆者による Emacs Lisp に対するオブジェクトの寿命計測の結果から，シンボルオブジェクトの場合，一度作られたものがごみとなることが少ないということが分かっている．
- 特に **intern-mutable** の対象となるシンボルは，関数の定義を変更しない限りごみとならないバイトコードから参照されているので，ごみとなる可能性が少ない．

したがって，このように長寿命であると判明しているシンボルを，直接古い世代にアロケートすることにより，ごみ集めに対する負荷を軽減することができる．

また，仮に mutable シンボルも若い世代にアロケートすることにすると，通常のシンボルとの判別のために 1bit が必要になるが，その場所を確保するのが Emacs Lisp では困難であるという理由も存在する．

このように改造したGNU Emacsでは，前述のmutableシンボルを特別に扱うバイトコンパイラによるバイトコードを読み込むことができる．そのようなバイトコードを読み込んだ場合には，`intern-mutable`でinternされているシンボルをmutableシンボル領域に作成する．しかし，`intern-mutable`以外の方法で新たに作成されたシンボルは，従来の世代別ごみ集めのGNU Emacsと同じく，若い世代にアロケートされ，若い世代のごみ集めを生き残ったシンボルだけ古い世代に殿堂入りする．この時，シンボルが殿堂入りする領域は普通のシンボル領域である．

また，このように改造したGNU Emacsでも，普通のバイトコンパイラで作成されたバイトコードも実行することが可能である．しかし，その場合にはそのバイトコードファイルから読み込まれたシンボルを特別に扱うことはなく，新たに作られたシンボルは，この場合にも，従来の世代別ごみ集めのGNU Emacsと同じように扱われる．

5. 性能評価

実装した機能の性能を計測し，評価するために，Sun Ultra1☆において，いくつかのテストプログラムを実行した．本論文で実装した世代別ごみ集めでは，ページサイズが4kバイトであることを仮定している．また，各オブジェクトのブロックのサイズを4kバイトにしてオブジェクトの配置を行なっている．それに対し，Sun Ultra1はページサイズが8kバイトであるので，1ページには2つのブロックが配置されることになる．その場合，同一ページに違う種類のブロックが配置される可能性もあるが，そのことが特に問題となるような結果とはならなかった．

テストに使用したプログラムは，バイトコンパイラとCのプログラムのインデントである．

5.1 バイトコンパイラ

バイトコンパイラを用いてEmacs Lispのソースファイルをバイトコンパイルした時の，書き換えの局所性とごみ集めの実行時間を計測した結果を示す．このバイトコンパイラには本論文で改造したバイトコンパイラを用いた．コンパイルするファイルは，標準でGNU Emacsに附属するLispファイルのうちでサイズの大きいものから10ファイルである．これらのファイルは合計約47000行あった．

表1は，バイトコンパイラを実行した時に，書き換えのあったブロック数とそれの総ブロック数に対する割合を示す．この表のブロック数は，各々の若い世代のごみ集め開始時に使用されていたブロック数の合計である．ごみ集めが実行された回数はどちらも82回であり同じであった．この表から，本論文の手法を用いることで，オリジナルに比べて約16%のシンボルブロックの書き換え率の改善がはかられていることがわかる．また**表2**から，mutableシンボルブロックの書き換え率が約88%と高くなり，その代わりに，一般のシンボルブロックでは書き換え率が25%と低くなっていることが分かる．このことから，プログラムの静的解析で書き換えられる可能性があると分かるシンボルは，普通のシンボルに比べて書き換えを受ける可能性が高いことが分かる．参照されたブロックの割合については特に違いはみられない．

表1 バイトコンパイルで書き換えのあったシンボルブロックの割合
Table 1 Ratio of dirty symbol blocks on byte-compilation.

	総ブロック数	書き換えブロック	参照ブロック
未配置	2770	1539(56%)	2740(99%)
配置済	2826	1128(40%)	2800(99%)

ブロック数:アロケートされていたブロックの総数
書き換えブロック:書き換えのあったブロック数
参照ブロック:参照されたシンボルブロック数
括弧内は総数に占める割合

さらに，バイトコンパイラの実行時の，ごみ集めの実行時間と処理系の実行時間について計測を行なったものを**表3**に示す．この表から分かるように，ごみ集めにかかる時間は約20msec短縮できている．ごみ集めの実行時のシステム時間はダーティビット情報の取得にかかった時間であるので，それを除いて考えるとユーザ時間では約50msecの時間の短縮が行なえている．これは，ごみ集めにかかるユーザ時間の約9%が削減できていることになる．また，処理系全体の実行時間も約400msec短縮できていることが分かる．この処理系の実行時間の短縮は，キャッシュミスの減少による処理速度の向上と考えられる．このことから，ごみ集めの時間が短縮されるだけでなく，書き込みの局所性の向上から処理系全体の性能の改善も行なわれることが分かる．

5.2 Cプログラムのインデント

C言語で書かれたGNU Emacsのソースファイルの一つであるemacs.cをGNU Emacsで読み込み，Cのプログラムの編集を行なうためのモードであるcc-modeにおいてインデントし直した時の，書き換えの局所性とごみ集めの時間を計測を行なった結果を示す．emacs.cは約1230行ある．

☆ CPUはUltraSPARC 200MHz，OSはSolaris2.5.1である．

表 2 シンボルの配置をする場合にバイトコンパイルで書き換えのあったシンボルブロックの割合
Table 2 Ratio of dirty symbol blocks on byte-compilation with symbol layout.

	mutable ブロック	一般のブロック
配置済	590/671(88%)	538/2155(25%)

分子/分母 :書き換えのあったブロック/ブロックの総数
mutable ブロック :mutable シンボルを配置したブロック
一般のブロック :一般のシンボルを配置したブロック
括弧内は分子が分母に占める割合を示す

表 3 バイトコンパイルの時間の計測結果
Table 3 Measured time of byte-compilation.

	utime	stime	etime	e-utime	e-stime
未配置	566	856	1410	24400	1190
配置済	516	863	1390	24000	1140

utime :ごみ集めで使用したユーザ時間の合計
stime :ごみ集めで使用したシステム時間の合計
etime :ごみ集めの実行にかかった総時間
e-utime :GNU Emacs の実行にかかったユーザ時間
e-stime :GNU Emacs の実行にかかったシステム時間
単位はすべて msec である

表 4 C のインデントで書き換えのあったシンボルブロックの割合
Table 4 Ratio of dirty symbol blocks on C program indent.

	総ブロック数	書き換えブロック	参照ブロック
未配置	1494	625(42%)	1010(68%)
配置済	1495	477(32%)	1062(71%)

ブロック数 :アロケートされていたブロックの総数
書き換えブロック :書き換えのあったブロック数
参照ブロック :参照されたシンボルブロック数
括弧内は総数に占める割合

表 5 シンボルの配置をする場合に C のインデントで書き換えのあったシンボルブロックの割合
Table 5 Ratio of dirty symbol blocks on C program indent with symbol layout.

	mutable ブロック	一般のブロック
配置済	327/390(84%)	150/1105(14%)

分子/分母 :書き換えのあったブロック/ブロックの総数
mutable ブロック :mutable シンボルを配置したブロック
一般のブロック :一般のシンボルを配置したブロック
括弧内は分子が分母に占める割合を示す

表 6 C のインデントの時間の計測結果
Table 6 Measured time of C program indent.

	utime	stime	etime	e-utime	e-stime
未配置	227	500	768	91060	580
配置済	203	480	682	90880	540

utime :ごみ集めで使用したユーザ時間の合計
stime :ごみ集めで使用したシステム時間の合計
etime :ごみ集めの実行にかかった総時間
e-utime :GNU Emacs の実行にかかったユーザ時間
e-stime :GNU Emacs の実行にかかったシステム時間
単位はすべて msec である

表 4 は，C のプログラムのインデントを行なった時に書き換えのあったシンボルブロックの割合を示す．ごみ集めが実行された回数はどちらも 65 回であった．この表から，本論文の手法を用いることで，オリジナルに比べて書き換えのあったシンボルブロックの割合が約 10%少なくなったことが分かる．また**表 5** から，バイトコンパイルの時にみられたように，mutable シンボルブロックへのシンボルの書き換えが集中していることも分かる．

表 6 は，C のプログラムのインデントを行なった時に，ごみ集めの時間を計測した結果である．C のプログラムのインデントでは，処理系の全体の実行時間に対してごみ集めの実行に要した時間の割合が，バイトコンパイラに比べて少なくなっている．しかし，この場合でも，ごみ集めの実行にかかった時間と，処理系の実行にかかった時間のどちらも短縮できていることが分かる．

処理系全体の実行時間の変化が少ないのは，GNU Emacs ではバイトコードを実行するので，オブジェクトの書き換えや参照が全実行時間に占める割合が低いためであると考えられる．したがって，コンパイラでマシンコードに変換してから実行するような処理系の場合は，違った結果になると考えられる．

6. 関連研究

Shaw は文献 6) において，ダーティビット情報を用いた世代別ごみ集めを実装した．Shaw は，シンボルの値スロットについては，書き換えが他のオブジェクトに比べて多く起こることに注目し，シンボルのすべてのフィールドを連続して同じページに配置するのではなく，各フィールド毎に専用のページに配置することによって書き換えの起こるページ数を減少させることが可能であると提案している．

また，Moon は文献 5) において，世代別ごみ集めの古い世代でのごみ集めに深さ優先方式のコピーを行なうことにより，広さ優先方式のコピーを行なう場合よりも参照の局所性が向上し，仮想メモリの性能が改善されるとしている．

本研究は，フィールド毎の配置ではなくオブジェクトの配置についての研究である点で Shaw の研究とは異なっていて，オブジェクトの書き換えの局所性に着目した点で Moon の研究とは異なる．

7. 議　論

本論文の手法では，ソースコード上でオブジェクト

の書き換えが行なわれることが検出された場合，その書き換えの対象となるオブジェクトを，書き換えの行なわれる可能性が高いオブジェクトだけが配置されている特別な領域に配置する．本論文の手法では，書き換えの行なわれるオブジェクトを静的な解析によって決定していたので，静的な解析では分からないオブジェクトの書き換えの検出を行なうことが不可能である．しかし，実行時の状況をプロファイリングすることにより，そのようなオブジェクトについてもうまく扱える可能性がある．

通常処理の実行時に書き換えのあった古い世代に存在するオブジェクトを，古い世代のごみ集めを実行する時に，mutable オブジェクト領域に移動するという方法も考えられる．現在の GNU Emacs の実装では，古い世代のごみ集めはコピー方式になっていないので，オブジェクトの移動は容易には実現できない．しかし，古い世代のごみ集めにコピー方式のごみ集めを採用している場合には，古い世代のごみ集めを行なう時に，書き換えの行なわれたオブジェクトを mutable なオブジェクト領域に移動するという方法を容易に実装することができる．古い世代のごみ集めを行なう時に書き換えの行なわれたオブジェクトを mutable なオブジェクト領域に移動するために必要となる操作は，

- プログラムの通常処理を行なっている時に，書き換えのあったオブジェクトの記録を作成する．
- 古い世代のごみ集めを行なう時に，その書き換えの記録にしたがってオブジェクトを配置する．

という二つの操作である．

プログラムが通常処理を行なっている時に書き換えのあったオブジェクトを記録するという処理は，世代別ごみ集めを実装する時に必要となる処理である．つまり，世代別ごみ集めが実装されている場合は，オブジェクトの書き換えの記録のために通常処理を行なっている時の性能低下が起こることはない．また，古い世代のごみ集めを実行する時に，書き換えの記録にしたがってオブジェクトの配置場所を決定するというのも特にコストのかかる操作ではない．したがって，書き換えのあったオブジェクトを mutable なオブジェクト領域に移動したとしても，プログラムの実行時の性能の低下は特にないと考えられる．

本論文で提案したオブジェクトの書き換えの局所性に基づいたオブジェクトの配置法と，Moon の手法に基づいた静的なオブジェクトの参照関係に基づくオブジェクトの配置法の 2 つを組み合わせることも可能であり，この 2 つの手法をともに用いることでプログラムの実行時の性能のさらなる改善が可能である．

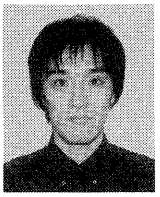
8. 結　　論

コンパイラによる静的解析の結果を用いてシンボルの配置を工夫することにより，書き換えの局所性を向上させることができ，書き換えのあったシンボルブロックの割合を約 10–15%減らすことができた．その結果，ごみ集めにかかる時間の短縮が行なえた．さらに処理系の実行時間も短縮された．今後はシンボルオブジェクト以外の一般的なオブジェクトへの適用と，実行時の動的な局所性の向上の手法，コンパイラを用いたシステムでの性能評価などを行なう必要がある．

参 考 文 献

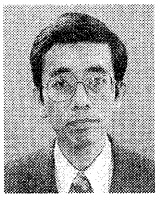
1) Baker, H. G.: Shallow Binding in Lisp 1.5, *Communications of the ACM*, Vol. 21, No. 7, pp. 565–569 (1978).
2) Boehm, H.-J., Demers, A. J. and Shenker, S.: Mostly Parallel Garbage Collection, *ACM SIGPLAN Notices*, Vol. 26, No. 6, pp. 157–164 (1991).
3) 小林広和, 寺田実: GNU Emacs への世代別ごみ集めの実装, 情報処理学会プログラミング研究会研究報告 97-PRO-16, pp. 37–42 (1997).
4) Lieberman, H. and Hewitt, C.E.: A Real-Time Garbage Collector Based on the Lifetimes of Objects, *Communications of the ACM*, Vol. 26, No. 6, pp. 419–429 (1983).
5) Moon, D. A.: Garbage Collection in a Large LISP System, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (Steele, G. L.(ed.)), Austin, TX, ACM Press, pp. 235–246 (1984).
6) Shaw, R. A.: *Empirical Analysis of a Lisp System*, Ph.D Thesis, Stanford University, Palo Alto, California (1988). Technical Report CSL-TR-88-351, Stanford University Computer Systems Laboratory.
7) Sobalvarro, P. G.: A Lifetime-Based Garbage Collector for LISP Systems on General-Purpose Computers, B.S. thesis, Massachusetts Institute of Technology EECS Department, Cambridge, Massachusetts (1988).
8) Ungar, D. M.: Generation Scavenging: A Non-disruptive High-Performance Storage Reclamation Algorithm, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM Press, pp. 157–167 (1984).

(平成 11 年 10 月 15 日受付)
(平成 12 年 2 月 22 日採録)



小林　広和（学生会員）

昭和 47 年生．平成 7 年東京大学工学部機械情報工学科卒業．平成 9 年同大学大学院工学系研究科情報工学専攻修士課程修了．現在同大学院同研究科同専攻博士課程在学中．記号処理言語での世代別ごみ集めの研究に従事．記号処理言語，オブジェクト指向言語などに興味を持つ．



寺田　実（正会員）

1959 年生．1981 年東京大学工学部計数工学科卒業．1983 年同大学院工学系研究科情報工学修士課程修了．同大学計数工学科助手，電気通信大学電子情報学科助手をへて 1991 年東京大学工学部機械情報工学科講師，1992 年同大学助教授，現在に至る．工学博士．主な研究分野はプログラム言語処理系，プログラミング環境など．日本ソフトウェア科学会，ACM 会員．