オブジェクト指向スクリプト言語 Ruby への世代別ガベージコレクションの実装と評価

木　山　真　人†

迅速なソフトウェア開発の要求が高まるにつれて，オブジェクト指向スクリプト言語 Ruby は，より多くの場面で使用され始めている．Ruby は従来のスクリプト言語のように規模の小さなプログラムだけに使用されるのではなく，その保守性の高さから規模の大きなプログラムにも使用されている．しかし，プログラムの規模が大きくなると実行時間が遅くなるという問題がある．これは，Ruby のごみ集め処理（GC）がマークスイープ法であることが原因の 1 つである．一般に，オブジェクト指向言語ではプログラマの負担を軽くするため，メモリ管理を処理系側で行う実装となっている．メモリを有効に利用するため，不要になったメモリを回収し再利用可能にする機構が処理系に必要となる．実装の容易さなどから，Ruby はマークスイープ法で GC を実装している．マークスイープ法では，実行中に存在するすべてのオブジェクトに比例して GC 処理時間が長くなる．そのため，オブジェクトを多数生成するような規模の大きなプログラムを実行すると，GC 処理時間が長くなり，その影響で実行時間が長くなる．そこで，プログラムの実行時間を短縮するため，GC の高速化に着目し，Ruby に世代別 GC の導入を検討する．本論文では，Ruby に世代別 GC を実装する場合の方法および結果を示す．世代別 GC にすることで，従来の GC にくらべ GC 処理時間を最大 78.74%，プログラムの実行時間を最大 21.51%短縮することができた．

Implementation and Evaluations of Generational Garbage Collection in Object-Oriented Scripting Language Ruby

Masato Kiyama†

Ruby, one of object-oriented scripting languages, is becoming more and more important as a tool for software development, as it provides great flexibility for rapid application development. Ruby is used for developing small-scale programs and large-scale programs. However, when the scale of a program becomes large, there is a problem that execution time becomes late. Because Ruby's Garbage Collection is mark-sweep. In general, memory management is implemented in object-oriented language in order to reduce programmers burden. GC is necessary to collect and reuse the unnecessary memory to utilize the memory effectively. By the mark sweep method, GC time becomes long in proportion to all the objects that exist during execution. Therefore, the ratio of whole execution time to GC execution time will increase as program size increase. In order to reduce program execution time, we introduce generational garbage collection in Ruby. In this paper, we show the method of implementation of generational garbage collection in Ruby, and how efficient that. It can reduce 21.51% of total execution time and 78.74% of the cost of garbage collecting for our benchmark.

1. はじめに

スクリプト言語が幅広い用途で使用され始めている．これはスクリプト言語が生産性に優れているためである[1),2)]．また，Python，Ruby[3)] といったスクリプト言語はオブジェクト指向機能を有しているため，生産性だけではなく，保守性も高い．そのため，オブジェクト指向スクリプト言語は文字列処理など規模の小さなプログラムだけではなく，Web アプリケーションなど規模の大きなプログラムに使用され始めている．このことから，オブジェクト指向スクリプト言語は今後ますます普及し，様々な用途で使用されると考える．

しかし，オブジェクト指向スクリプト言語には幅広い用途での使用を妨げる問題がある．それは，オブジェクト指向スクリプト言語の処理速度である．オブジェクト指向スクリプト言語の多くはインタプリタ方式で実装されているため処理速度が遅く，高速な処理を必要とするアプリケーション開発には向いていない．処理速度の問題を解決し，オブジェクト指向スクリプ

† 広島市立大学情報科学研究科
Graduate School of Information Sciences, Hiroshima City University

ト言語の幅広い用途での使用を促進させるため，オブジェクト指向スクリプト言語の高速化が重要となる．

そこで，著者はオブジェクト指向スクリプト言語の 1 つである Ruby の高速化を行っている[4]．Ruby には規模の大きなプログラムを実行すると実行時間が遅くなるという問題がある．これは，Ruby のガベージコレクション（以下，GC）がマークスイープ法であることが 1 つの原因であると考えられる．

オブジェクト指向言語では，メモリの動的領域確保は必須であり，プログラマがプログラムの本質的な部分に集中して開発できるように，自動領域管理機構としての GC は必要である．特に，オブジェクト指向プログラミングでは多数のオブジェクトを生成したり，それらが複雑に関係付けられたりするので，GC によるプログラマの負担の軽減は著しいものがある．一般に，GC はスタックやレジスタ上にあるオブジェクトへの参照をルート（root）とし，ルートから直接的あるいは間接的に参照されているオブジェクトを今後も使用する可能性があると判断し，回収しない．ルートから直接的にも間接的にも参照されていないオブジェクトはごみと判断され，回収される．

Ruby の GC は実装の容易さなどからマークスイープ法で実装されている．マークスイープ法では，実行中に存在するすべてのオブジェクトに比例して GC 処理時間が長くなる．そのため，オブジェクトを多量に生成するような規模の大きなプログラム実行すると，GC 処理時間が長くなり，その影響で実行時間が長くなる．

そこで，プログラムの実行時間を短縮するため，GC の高速化に着目し，Ruby に世代別 GC の導入を検討する．本論文では，マークスイープ法で実装された Ruby（以下，従来手法）に世代別 GC を適用し，その実装方法と有効性を示す．

2. Ruby について

本章では，Ruby について簡単に述べる．また，Ruby におけるメモリ管理方法，GC の問題点について述べる．

2.1 概　　要

Ruby はオブジェクト指向スクリプト言語の 1 つである．変数や式に型がない，整数などの基本的なデータ型をはじめとしてすべての値がオブジェクトなどの特徴がある．また，文字列操作，正規表現，ファイル入出力，配列，連想配列，プロセス操作，例外処理機能，ネットワーク入出力，スレッド機能などの豊富なクラスライブラリがある．これらを組み合せることで，様々な用途のプログラムを容易に記述することができる．さらに，移植性が高く，多くの UNIX 上や Windows 上で動作している．

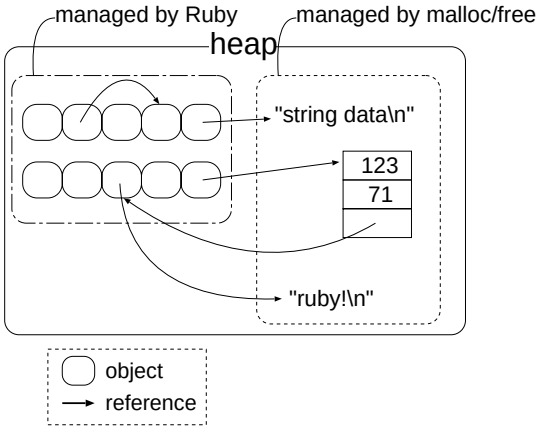

図 1　ヒープの構造
Fig. 1　The structure of the heap.

2.2 メモリ管理方法

Ruby では，生成されるオブジェクトや文字列などのデータはヒープ領域に確保される．そのため，GC はヒープ領域を対象として行われる．

Ruby のヒープ構造を**図 1** に示す．オブジェクトの割り当て/解放などは Ruby が管理する．管理の都合上，すべてのオブジェクトは同じ大きさである．文字列など可変部分のデータは OS あるいはライブラリの提供する malloc/free が管理する．

オブジェクトの割り当てと GC は以下の手順で実行される．

(1) オブジェクトの配列をヒープ領域から確保し，オブジェクトを単方向リストでつなげる．この未使用のオブジェクトのリストを freelist と呼ぶ．
(2) オブジェクトを割り当てるとき，freelist からオブジェクトを取り出し，クラスの設定/可変部分のデータの割り当てなどを行う．
(3) freelist につながるオブジェクトがなくなると GC が開始される．
(4) ルートから追跡可能なすべてのオブジェクトをマーキングする．
(5) オブジェクトの配列を走査し，マークの付いていないオブジェクトをごみと判断する．ごみとなったオブジェクトは可変部分のデータの解放などを行ない，freelist に移す．
(6) GC 終了後，freelist につながるオブジェクトが FREE_MIN 個以下ならば，オブジェクトの配列をヒープ領域から確保し，freelist につなげる．

```
typedef unsigned long VALUE;
struct RBasic {
 unsigned long flags;/* マークビットなど */
 VALUE klass;        /* クラス */
};
struct RArray {
 struct RBasic basic;
 long len, capa;     /* 配列，領域の長さ */
 VALUE *ptr;         /* 配列領域 */
};
```

図 2 オブジェクトの構造体
Fig. 2 Definition of object.

FREE_MIN は Ruby のソースコードで定義されている定数であり，4096 である．

Ruby におけるオブジェクトの構造体を**図 2** に示す．オブジェクトには，そのオブジェクトのクラス，マーク用のフラグ，文字列へのポインタ，配列の長さなどが保持される．文字列や配列など可変部分のデータはオブジェクトに保持されず，可変部分のデータへの参照が保持される．

例えば，配列オブジェクトは以下の手順で生成される．

(1) freelist からオブジェクトを取り出す．
(2) オブジェクトの flags に配列オブジェクトであることを示すビットを立て，klass に配列クラスへの参照を代入する．
(3) len に 0 を代入し，capa にあらかじめ確保する領域の大きさを代入する．
(4) capa 分の領域を malloc で確保し，確保した領域への参照を ptr に代入する．

2.3 GC の問題点

Ruby では，オブジェクトを多数生成するようなプログラムを実行する場合，プログラムの実行時間に占める GC 処理時間の割合が大きくなるという問題がある[4]．これは Ruby の GC がマークスイープ法であることが原因である．マークスイープ法は，ルートから追跡可能なすべてのオブジェクトにマークを付け，ヒープ領域全体を走査し，マークの付いていないオブジェクトをごみと判断し，回収する方法である．そのため，プログラム中に使用されるオブジェクトの数が多くなればなるほど追跡に時間がかかり，メモリ領域が大きくなればなるほど，ごみとなったオブジェクトの回収に時間がかかる．

3. Ruby への世代別 GC の実装

Ruby の GC はマークスイープ法である．そのため，オブジェクトを多数生成するようなプログラムを実行すると GC 処理時間が長くなるという問題がある．

この問題を解決する GC の実装方法の 1 つとして世代別 GC がある[5]．「生成されたオブジェクトのほとんどは寿命が短く，生成されてからすぐにごみとなってしまうが，ある程度生き続けたものは半永久的に生き残る」という性質がオブジェクトにあることが知られている[6]．世代別 GC は，この性質を利用した方法である．世代別 GC では，オブジェクトをその寿命に応じていくつかの世代に分け，通常は若い世代の領域のみを GC の対象とし，若い世代の領域が少なくなったとき，古い世代の領域を対象とする GC を行う．世代別 GC では通常，若い世代の領域のみを GC の対象とするため，追跡の時間がプログラム中に使用されるオブジェクトの数に依存することはなく，メモリ領域の大きさにごみの回収時間が依存することはない．

そこで，Ruby に世代別 GC の考えを導入することで，GC 処理時間が短縮されると考えられる．Ruby に世代別 GC を実装する場合に問題となる点とその解決方法を以降で説明する．

3.1 世代間の移動方法

オブジェクトの世代間移動方法を考えるとき，Ruby では 2 つの考慮すべき点がある．オブジェクトのアドレスと，プログラム実行中でのオブジェクトの再利用である．

世代別 GC では，GC を経験した回数をオブジェクトの寿命とするのが一般的である．オブジェクトを寿命に応じていくつかの世代に分類し，異なる領域に割り当てるのが世代別 GC の基本的な考え方である．通常の GC では，もっとも若い世代だけを GC の対象とする．そして，GC を経験した回数がある一定の回数（Advancement Threshold；AT）を越えた場合に，そのオブジェクトを 1 つ上の世代へ複写する．この複写を殿堂入り（tenuring）という．

このように，世代別 GC を実装するにはオブジェクトを複写して世代を移動することが一般的である．しかし，Ruby でオブジェクトを複写することは，以下の理由により困難である．

- Ruby の GC は半保守的[7] である．よって，世代間移動を複写によって行うのは可能[8] であるが，ソースコードに大幅な変更が必要となる．
- Ruby のソースコードには，オブジェクトのアドレスが変更されないことを前提としたインタプリ

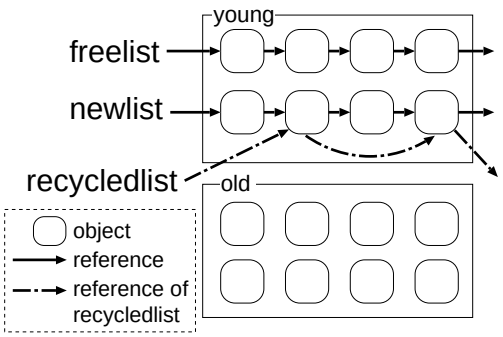

図 3 世代の構造
Fig. 3 Structure of generation.

タの実装部分がある．そのため，世代間移動を複写で行うには，それらすべての実装を変更する必要がある．

そこで，オブジェクトのアドレスを変更させず，世代間移動を行う方法を考える必要がある．

通常，オブジェクトがごみと判断されるのは GC のときである．しかし Ruby では，プログラム実行中に，オブジェクトをごみと判断する場合がある．それはインタプリタ内部でオブジェクトが明確にごみと判断された場合である．プログラム実行中に，ごみと判断されたオブジェクトは回収され，再利用される．そこで，プログラム実行中にオブジェクトを再利用する方法を考える必要がある．

以上の 2 点を考慮し，本手法ではオブジェクトを単方向リストでつなぎ，オブジェクト内部にあるフラグの書き換えによって世代間移動を行う．

図 3 に，本手法での世代の構造を示す．本手法では世代を若い世代（young）と古い世代（old）の 2 つに分ける．オブジェクトを割り当てるときは，オブジェクトを freelist から newlist へ移す．freelist につながるオブジェクトがなくなると GC が開始される．以下に GC の手順を示す．

(1) ルートから追跡可能な若い世代のオブジェクトをマーキングする．
(2) newlist のオブジェクトを走査し，マークが付いているオブジェクトにはなにもせず，古い世代のオブジェクトとみなす．マークが付いていないオブジェクトは可変部分のデータの解放などを行い，freelist に移す．

こうすることで，オブジェクトのアドレスを変更させることなく，世代間移動を行うことができる．

プログラム実行中にごみとなったオブジェクトが古い世代ならば，freelist につなげて再利用することができる．しかし，ごみとなったオブジェクトが若い世代のときは，freelist につなげることはできない．なぜなら，若い世代のオブジェクトは単方向リストでつながっているため，freelist につなげると，newlist のリンクが切れてしまう．newlist のリンクを切らないようにオブジェクトを freelist につなげようとすると，効率的に再利用できない．なぜなら，単方向リストでは，リストの途中にあるオブジェクトを取り出すのに時間がかかるからである．

そこで，本手法ではプログラム実行中にごみとなった若い世代のオブジェクトは recycledlist という新たに追加したリストにつながる（図 3）．ごみとなったオブジェクトの内部は空となるため，その部分を使って recycledlist にオブジェクトをつなげる．newlist につながる領域を使わず，オブジェクト内部の領域を使って recycledlist につなげるため，newlist のリンクは切れない．そのため，recycledlist につながっているオブジェクトは newlist にもつながっている．freelist につながっているオブジェクトがなくなると，recycledlist につながっているオブジェクトを新たに割り当てるオブジェクトとして使う．freelist からオブジェクトを割り当てるときは，newlist にオブジェクトをつなげる．recycledlist からオブジェクトを割り当てるときは，すでに newlist につながっているので，newlist のリンクが切れないように，オブジェクトを割り当てる．こうすることで，プログラム実行中にオブジェクトを効率的に再利用できる．

本手法では，オブジェクト毎に単方向リストのためのデータ領域が必要となるため，従来手法よりメモリ使用量が増加する．

3.2 世代間参照の検出方法

世代別 GC では若い世代の領域のみを対象とする GC のとき，ルートから参照されている若い世代のオブジェクトを追跡し，マーキングする．さらに，古い世代から参照されている若い世代のオブジェクトも追跡し，マーキングしなければならない．これは古い世代から参照されている若い世代のオブジェクトがごみと判断されないようにするためである．このため，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，その参照関係を管理する必要がある．

参照を検出するため，文献 9) では，OS の提供するシグナル機構を利用している．具体的には，まず古い世代のオブジェクトが存在するメモリ領域に対して，メモリの保護をかける．そして，その部分に書き込みを行なおうとしたときに，そのアドレスを調べる．そのアドレスに古い世代のオブジェクトが存在し，かつ書き込みを行なおうとするものが若い世代のオブジェ

クトへのアドレスならば，そのアドレスを次の GC のときのルートにするという方法である．この方法では，OS の発するシグナルを利用しており，コストがかかる方法である．

文献 10) では，古い世代からの参照の検出方法に仮想メモリのダーティビット情報を利用している．このダーティビット読み出し機能を利用することで，シグナルを利用せずにすみ，シグナルの処理にかかっていた時間を短縮することができる．さらに，ソースコード上でオブジェクトの参照の書き換え箇所に，検出のためのコードを挿入する必要がないので，ソースコードの変更箇所が少なくなる．しかし，この方法はページ毎に書き込みの有無を検出する方法であり，標準的なマシンでは 1 ページは 4096 バイトであるので，書き込みのあったオブジェクトを検出するには大きすぎ，若い世代を指しているオブジェクトを探すのに時間がかかってしまうという問題点がある．この問題点を解消するための研究として，文献 11) が提案されている．ダーティビット情報を利用する方法は，OS に強く依存する方法であるため，Ruby のように多くの OS で稼働するプログラムには向いていない．さらに，本手法では，オブジェクトの世代間移動を複写で行っていないため，ページの中に古い世代のオブジェクトと若い世代オブジェクトが混ざっている可能性があり，その場合無駄な検出をしてしまう．

本手法では，古い世代から若い世代への参照を検出するため，参照の書き換えが起こりうる全ての箇所で世代間参照検出用のコードを用いて検出を行っている．この方法は，OS の機能を使わないため，より少ないオーバヘッドで古い世代から若い世代への参照を検出することが可能である．

3.3 拡張ライブラリへの対応方法

高速化のため，プログラムの一部を C で記述する場合や，既存の C のライブラリを利用するため，Ruby は C による拡張機能を備えている．この C による拡張機能を用いて，Ruby から既存の C のライブラリなどを利用できるようにしたものを拡張ライブラリという．拡張ライブラリで生成されるオブジェクトは GC の対象となるため，世代間参照を検出しなければならない．

本手法は，拡張ライブラリで生成され，かつ，他のオブジェクトへの参照を持つようなオブジェクトを datalist という特別なリストで管理する．datalist につながったオブジェクトを毎回 GC の対象とするように管理する．こうすることで，拡張ライブラリで世代間参照を検出する必要がなくなる．

表 1 評価環境
Table 1 Evaluation environment.

CPU	PentiumII 350MHz	
キャッシュ	1 次キャッシュ	データ: 16 KB 命令: 16 KB
	2 次キャッシュ	512KB
主記憶	128 MB	
OS	Linux 2.2.16	
コンパイラ	gcc 2.95.2	

3.4 全領域 GC 開始とヒープ領域拡張条件

実装する世代別 GC には，若い世代の領域のみ対象とする GC と全領域を対象とする GC がある．そのため，それらの開始条件を決めなければならない．全領域を対象とする GC の開始条件が決まれば，それ以外の条件のとき，若い世代の GC が開始される．また，ヒープ領域が少なくなれば，ヒープ領域を拡張しなければならず，この条件も決める必要がある．

本手法では全領域 GC 開始条件とヒープ領域拡張条件を以下のようにする．

- 全領域 GC 開始条件
  若い世代の領域のみ対象とする GC が終了後，freelist につながっているオブジェクトが FREE_MIN_MINOR 個以下となるとき．
- ヒープ領域拡張条件
  全領域 GC 終了後，freelist につながるオブジェクトが FREE_MIN 個以下となるとき．

FREE_MIN_MINOR は本手法で新たに定義した定数であり，2000 である．FREE_MIN_MINOR を 2000 としたのは，「割り当てられるオブジェクト数が 2000 個以下になると，回収されるオブジェクトの割合が少なくなる」という観測結果[4] からである．従来手法の FREE_MIN と同様の理由である．また，FREE_MIN は従来手法と同じである．

4. 性 能 評 価

本手法を Ruby1.7.1（2001-07-26）版に実装し，その効果を測定した．評価環境を**表 1** に示す．従来手法（ruby）と本手法（gene）でテストプログラムを実行し，結果を比較することで評価する．

評価に用いたテストプログラムは以下の 4 つである．

- life: ライフゲーム
  150 世代まで計測．プログラム実行中に使用したオブジェクトの最大個数は 7 万個．拡張ライブラリを使用している．このプログラムは文献 12) にある．
- occur: 単語の出現頻度計測
  入力するファイルは Ruby のソースコード（約 7

表 2 GC 処理時間
Table 2 GC time.

	life	occur	occur2	compact
ruby	2093	1034	29049	804
gene	1383	607	6175	359

単位はすべて msec

表 3 実行時間
Table 3 Total execution time.

	life	occur	occur2	compact
ruby	39.78	7.73	92.49	5.34
gene	38.55	7.56	72.59	4.87

単位はすべて sec

万 6 千行）．プログラム実行中に使用したオブジェクトの最大個数は 3 万個．これは，Ruby 附属のサンプルプログラムである．

- occur2: 単語の出現頻度計測
  入力するファイルは GCC のソースコード（約 58 万行）．プログラム実行中に使用したオブジェクトの最大個数は 15 万個．
- compact: HTML ファイルを短くする
  入力するファイルは 125KB の HTML ファイル．プログラム実行中に使用したオブジェクトの最大個数は 7 万個．このプログラムは文献 13) で配布されている．

4.1 GC 処理時間

従来手法と本手法の GC 処理時間を**表 2** に示し，結果について考察する．

表 2 から，すべてのテストプログラムにおいて本手法の GC 処理速度は従来手法と比べて速いことが分かる．これは本手法が世代別 GC であるためだと考えられる．本手法は通常若い世代の領域のみを，従来手法ではすべての領域を GC の対象とする．そのため，従来手法と比べて本手法は 1 回の GC 処理時間が速くなり，それによって全体の GC 処理時間が速くなったと考えられる．

4.2 実 行 時 間

従来手法と本手法におけるテストプログラムの実行時間を**表 3** に示す．**図 4** は，従来手法の実行時間を 1 として，実行時間の相対値を示したグラフである．値が小さいほど実行時間が速いことを示している．グラフの斜線部分は実行時間内で GC の処理に要した時間を，白い部分は GC の処理以外に要した時間を示している．表 3，図 4 の結果について考察する．

図 4 から，すべてのテストプログラムにおいて本手法での実行時間は従来手法と比べて速い．また，図 4 から，GC 処理時間の減少が実行時間の短縮につながることが分かる．

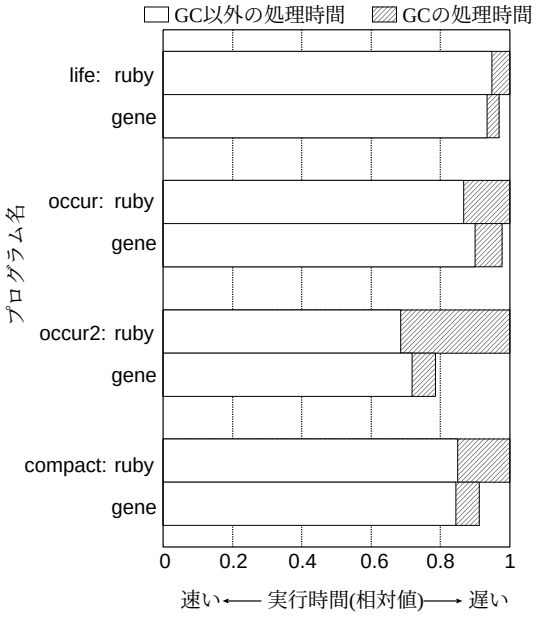

図 4 実行時間の比較
Fig. 4 Comparing execution time.

表 4 メモリ使用量
Table 4 Memory usage.

	life	occur	occur2	compact
ruby	6928	2656	8856	5276
gene	11692	2780	9476	5488

単位はすべて KB

図 4 から，occur と occur2 では従来手法と比べ，本手法の GC 以外の処理時間が遅いことが分かる．これは，従来手法にはない，世代間参照検出のコストと考えられる．本手法では，世代間参照検出をソフトウェアで検出しているため，参照の書き換えが起きると必ず世代間参照検出用の関数が呼ばれる．そのため，参照の書き換えが頻繁に行われる occur や occur2 などでは世代間参照検出のコストが大きくなり，GC 以外の処理時間に影響を与えると考えられる．

4.3 メモリ使用量

従来手法と本手法の最大メモリ使用量を**表 4** に示し，結果について考察する．

表 4 から，すべてのテストプログラムにおいて本手法でのメモリ使用量は従来手法と比べて多い．本手法でのメモリ使用量が増加する原因として，以下の理由が考えられる．

- 世代間移動を実現するために必要なオブジェクト毎の単方向リストのポインタ増加分．
- 古い世代でごみとなったオブジェクトは全領域 GC

のときのみ回収される．そのため，従来手法と比べ，無駄な領域が生じている．

5. 関連研究

オブジェクト指向スクリプト言語の高速化手法は，これまであまり研究されていない．これは従来のスクリプト言語が，短時間で終ってしまう小規模なプログラムにしか用いられなかったため，高速化の必要性がなかったからである．しかし，現在，オブジェクト指向スクリプト言語は規模の大きなプログラムにも用いられ，高速化の必要がある．規模の大きなプログラムでは，実行時間に占める GC 処理時間の割合が大きい．そのため，GC の高速化が必要となる．

GNU Emacs に世代別 GC[5] を実装する研究が，小林らによって行われている[10),11)]．仮想メモリのダーティビット情報を利用して，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の検出をする．仮想メモリのダーティビット情報を利用することで，GC の中断時間短縮，参照の検出のために必要なプログラムの修正コスト削減，参照の検出のオーバヘッド削減を目的としている．本手法は，世代間移動をオブジェクト内部にあるフラグの書き換えによって行うが，小林らの手法は，オブジェクトをコピーすることで世代間移動を行う点が異なる．小林らの手法は仮想メモリのダーティビット情報を利用しているため，参照の検出時間が長くなってしまうが，本手法は参照の検出をソフトウェアで実現しているため，参照の検出時間が短い．

6. 結論及び今後の展望

本論文では，オブジェクト指向スクリプト言語 Ruby に世代別 GC を実装する場合の方法および結果を示した．また，詳細な評価を行い，Ruby における世代別 GC の有効性を示した．計測結果から，世代別 GC にすることで GC 処理時間が最大 78.74%，プログラムの実行時間が最大 21.51%短縮した．

今後の展望として，以下が挙げられる．

- 古い世代のごみを減少:
  文献 14) の手法を適用することで，古い世代の領域でごみとなるオブジェクトを減少させる．こうすることで，メモリ使用量の減少，GC 処理時間の短縮の効果が期待できる．
- 古い世代の GC の処理時間短縮:
  古い世代の GC を時間的に分散させることで，古い世代の GC 処理時間を短縮する．こうすることで，インタラクティブなプログラムに対応することができる．

参考文献

1) Ousterhout, J.: Scripting: Higher level programming for the 21st century, *IEEE Computer*, Vol. 31, No. 3, pp. 23–30 (1998).
2) Prechelt, L.: An Empirical Comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a Search/String-Processing Program, Technical Report2000-5, Fakultät für Informatik, Universität Karlsruhe, Germany (2000).
3) まつもとゆきひろ, 石塚圭樹: オブジェクト指向スクリプト言語 Ruby, アスキー出版局 (1999).
4) 木山真人: オブジェクト指向スクリプト言語 Ruby への世代別ごみ集めの実装と評価, 情報処理学会論文誌, Vol. 42, No. SIG3(PRO10), pp. 40–48 (2001).
5) Jones, R. and Lins, R.: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley (1996).
6) Lieberman, H. and Hewitt, C. E.: A Real-Time Garbage Collector Based on the Lifetimes of Objects, *Communications of the ACM*, Vol.26(6), pp.419–429 (1983). Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.
7) 小野寺民也: 特集：＜ごみ集めの基礎と最近の動向＞保守的ごみ集め, 情報処理, Vol. 35, No. 11, pp. 1020–1026 (1994).
8) Bartlett, J. F.: Mostly-Copying Garbage Collection Picks Up Generations and C++, Technical Note TN-12, Digital Equipment Corporation, Western Research Lab (1989).
9) 萩原知章, 岩井輝男, 中西正和: オブジェクトの世代を考慮に入れた保守的ごみ集め, 情報処理学会プログラミング研究報告 97-PRO-16, pp. 31–36 (1997).
10) 小林広和, 寺田実: ダーティビット情報を用いた世代別ごみ集めの GNU Emacs への実装, 情報処理学会論文誌, Vol. 41, No. 7, pp. 1948–1955 (2000).
11) 小林広和, 寺田実: 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, 情報処理学会論文誌, Vol. 41, No. SIG4(PRO7), pp. 24–31 (2000).
12) http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/~poffice/mail/ruby-list/6611.
13) http://www.ruby-lang.org/en/raa.html.
14) 吉川隆英, 近山隆: 効率のモデルに基づきヒープサイズを自動調整する世代 GC 方式, 情報処理学会論文誌, Vol. 41, No. SIG9(PRO8), pp. 78–86 (2000).