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

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は，より多くの場面で使用されている．これまでのスクリプト言語は，主に小規模なプログラムに使用されていたが，オブジェクト指向スクリプト言語はその保守性の高さから大規模なプログラムにも使用されている．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．そのため，処理系はメモリを有効に利用するために，不要になったメモリを回収し再利用可能にするためのごみ集め処理（GC）が必要になる．多くのオブジェクト指向スクリプト言語は GC を有しているが，実装の容易さから，マークスイープ法，リファレンスカウント法が用いられている．しかし，これらの方法ではプログラムの規模が大きくなるにつれて，GC 処理時間の全実行時間に占める割合が大きくなる．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，世代別 GC の導入を検討する．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す．世代別 GC にすることで，従来の GC にくらべ GC 処理時間が最大 92%，プログラムの実行時間が最大 50%短縮することが分かった．

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

Masato Kiyama†

Object-oriented scripting languages are becoming more and more important as a tool for software development, as it provides great flexibility for rapid application development. Scripting languages have been used for developing small-scale programs, object-oriented scripting languages are also used for developing large-scale programs. In general, memory management is implemented in object-oriented language in order to reduce programmers burden. Garbage Collection is necessary to collect and reuse the unnecessary memory to utilize the memory effectively. Mark-sweep and reference counting are general use among most object-oriented scripting languages. But these method, 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 50% of total execution time and 92% of the cost of garbage collecting for our benchmark.

1. はじめに

スクリプト言語は現在幅広く普及しており，その重要性が指摘されている[1]．スクリプト言語は生産性に優れ，Python，Ruby といったスクリプト言語はオブジェクト指向機能を持つことで，生産性と保守性をさらに高めている．このことから，オブジェクト指向スクリプト言語は小規模から大規模なプログラムまで，幅広く使用されており，今後ますます普及すると考える．

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ガベージコレクション（以下，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大量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．そのため，多くのオブジェクト指向スクリプト言語は GC を有しているが，その実装の容易さから，リファレンスカウント法やマークスイープ法で実装されており，これが処理速度を遅くする原因となっている．

そこで本研究では，オブジェクト指向スクリプト言語の GC に世代別 GC[2] の手法を適用し，GC 処理時

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

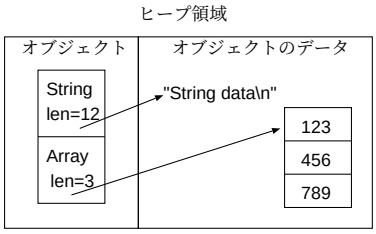


図 1 ヒープの構造
Fig. 1 The structure of the heap.

間の短縮を目的とする．

本論文では，広く使用されているオブジェクト指向スクリプト言語 Ruby[3] に世代別 GC の手法を適用し，その実装方法と有効性を示す．

2. Ruby について

Ruby について簡単に説明する．

2.1 概　　要

Ruby は，まつもとゆきひろ氏によって作成されたオブジェクト指向スクリプト言語であり，C 言語で書かれている．C による拡張機能を備えており，既存の C のライブラリを Ruby から扱えるようになっている．Ruby には，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である，といった特徴がある．また，Ruby には，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り，さまざまな用途のスクリプトを書くことができる．さらに，移植性が高く，多くの UNIX，DOS，Windows，Mac，BeOS 上で動作している．

2.2 メモリ管理方法

Ruby の処理対象となるオブジェクトはすべてヒープ領域に割り当てられる．Ruby のヒープ構造を**図 1** に示す．オブジェクトの割当ては Ruby が管理し，オブジェクトのデータ部分の割当ては，OS あるいはライブラリの提供する malloc/free が管理している．オブジェクトは，文字列の長さ，文字列へのポインタなどを含んでおり，オブジェクトのデータ部分には文字列や配列のデータが割り当てられている．オブジェクトは管理の都合上，その種類に関係なく，すべてポインタ 5 つ分に統一されている．Ruby のデータ構造を**図 2** に示す．RBasic はすべてのオブジェクトが持っている構造体であり，GC のときマーキングをするビットが flags にある．

2.3 GC の方法

Ruby では，最初にオブジェクト 1 万個分を配列と

```
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;
 int len;            /* 配列の長さ */
 int capa;           /* 領域の長さ */
 VALUE *ptr;         /* 配列領域 */
};
```

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

してメモリ領域から獲得し，それをリストとしてリンクしておく．これをフリーリストと呼ぶ．新しいオブジェクトはフリーリストから割り当てられ，フリーリストを使い切ると GC が起動する．GC を起動後，ごみとなったオブジェクトのデータ部分を解放するなどの後処理を行い，それをフリーリストにつなげ，再びオブジェクトの割当てを開始する．もし，GC を起動後，フリーリストにつながっているオブジェクトが 512 個以下なら，新たにオブジェクトを 1 万個分メモリ領域から獲得し，フリーリストにつなげる．

3. Ruby の GC の問題点

オブジェクト指向スクリプト言語の多くは，GC をリファレンスカウント法やマークスイープ法で実装しており，Ruby の GC はマークスイープ法である．マークスイープ法では，プログラム中に使用されるオブジェクトの数が多ければ多いほどマーキングに時間がかかり，メモリ領域が大きくなればなるほど回収に時間がかかる，という問題点がある．

これらの問題点を解決する GC の手法として世代別 GC がある．オブジェクトには，ある程度生き続けたものは半永久的に生き残り，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性質があることが知られている．そこで，オブジェクトをその寿命に応じていくつかの世代にわけ，通常は寿命の短い世代の領域のみを頻繁に GC を行い，領域が少なくなったときに長寿命領域の GC を行うという手法が，基本的な世代別 GC である．世代別 GC では通常，寿命の短い世代の領域のみ GC を行うので，マーキングの時間がプログラム中に使用されるオブジェクトに依存することはなく，メモリ領域の大きさに回収時間が依存することはない．

表 1 評価環境
Table 1 Evaluation environment.

CPU	PentiumII 350 MHz	
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	512 KB
主記憶	128 MB	
OS	Linux 2.2.12	
コンパイラ	egcs-2.91.66	

表 2 生存期間の長さが 0 の割合
Table 2 Rate of lifetime length 0.

プログラム	割合
ライフゲーム	95.45%
HTML パーサー	87.27%

世代別 GC を実装する前に，まず Ruby 上でのオブジェクトの振舞いが「生成されたオブジェクトのほとんどは寿命が短い」という仮定を満たしているか，また GC を改善した場合，どの程度の性能向上が得られるかを確認するため，第 1 次評価として，以降の評価を行った．

3.1 オブジェクトの生存期間

オブジェクトが割り当てられてから，GC によって回収されるまでの期間をオブジェクトの生存期間という．生存期間中に起動された GC の回数を生存期間の長さと定義し，Ruby のプログラムにおけるオブジェクトの生存期間を計測した．本論文の実験で用いた Ruby のバージョンは 1.4.4 である．評価環境を**表 1** に示す．ベンチマークプログラムには以下のものを使用した．

- ライフゲームのシミュレーション
  生物の生存競争をシミュレーションするプログラムである．150 世代まで計測した．生成されるオブジェクトは約 100 万個，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 万個と多数である．このプログラムは文献 3) のものを使用した．
- HTML のパーサー
  HTML ファイルからタグを除いたデータを返すプログラムである．生成されるオブジェクトは約 8 万個，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 1 万個と少数である．このプログラムは文献 4) のものを使用した．

オブジェクトの生存期間の長さが 0 の割合，すなわち，生成後すぐに回収されるオブジェクトの割合を**表 2** に示す．表 2 から，大多数のオブジェクトが生成後すぐに回収されていることが分かる．すなわち，Ruby 上でのオブジェクトの振舞いは「生成されたオブジェクトのほとんどは寿命が短い」ということがいえる．

表 3 GC 処理時間の占める割合
Table 3 Rate of GC time.

プログラム	割合
ライフゲーム	54%
HTML パーサー	9%

3.2 GC 処理時間の占める割合

GC 処理時間がプログラムの実行時間に占める割合を**表 3** に示す．

Ruby の GC はマークスイープ法であるため，GC 1 回にかかる時間は，そのときにプログラムで使用されていたオブジェクトの数とメモリ使用量に比例して大きくなる．ライフゲームでは，プログラムで使用するオブジェクトの数が多いため GC 1 回にかかる時間が長くなり，実行時間に占める GC の割合が大きくなる．一方，HTML パーサーの方は使用するオブジェクトの数が少ないため，実行時間に占める GC の割合が小さくなっている．GC の処理時間を短縮することで，ライフゲームのような使用されるオブジェクトの数が多いプログラムの実行時間を短縮することが可能となる．

以上の結果から，Ruby の GC に世代別の考えを適用することは有効であり，GC の処理時間を短縮することは，実行時間の短縮に大きな影響があると考えられる．そこで，本研究では Ruby の GC に世代別 GC を実装する．

4. Ruby への世代別ごみ集めの実装

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．

4.1 世代間の移動方法

世代別 GC は，オブジェクトの寿命の長さを判定し，いくつかの世代に分類して異なる領域に割り当てるのが基本的な考え方である．通常の GC では，最も若い世代だけを 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）を超えた場合に，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[5] である．よって，世代間の移動に複写を使用することは可能であるが，

Ruby のソースコードに大幅な変更が必要となる．

- メモリに割り当てられたオブジェクトのアドレスがハッシュのキーなどに使われており，オブジェクトを複写することでアドレスが変化すると，キーのハッシュ値を再計算しなければならない．

そこで，本論文では双方向リストを用いて世代別 GC を実装する．オブジェクトを双方向リストでつなぎ，世代間の移動を複写ではなく，双方向リストの付け換えと，オブジェクト内部のフラグの書き換えによって行う．これにより，オブジェクトが世代間を移動してもアドレスの変更が起きない．双方向リストを用いる手法は，Treadmill GC[6] と同様の手法である．

この手法を用いると，オブジェクトごとに双方向リストのためのポインタを必要とするため，従来の方法よりメモリ使用量が増加する．

4.2 古い世代からの参照

世代別 GC では通常の GC のとき，最も若い世代だけを GC の対象とするが，古い世代のオブジェクトが若い世代のオブジェクトを参照している場合，それを検出しなければならない．

この参照を検出するため，文献 7) 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，かつ書き込みを行おうとするものが新しい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 8) 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，この方法はページごと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 9) 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，本論文の世代別 GC の実装では，オブジェクトの移動を複写で行っていないため，ページの中に古いオブジェクトと新しい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．

本論文では，古い世代から新し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すべての箇所で検出を行うようにソースコードを変更した．Ruby ではオブジェクトの書き換えは必ずオブジェクトが実行することと，オブジェクト指向特有のコード共有を利用すれば，オブジェクトの書き換えを検出する箇所を限定することができる．この考えを利用して，ソースコードの変更を行った．オブジェクトの書き換えを検出する箇所は 69 カ所あり，世代別 GC を実装するためのコードを 200 行追加した．また，検出を行う rb_gc_check_ref という関数を新たに追加した．古い世代から新しい世代を指すような参照が検出された場合は，次の GC のとき，そのオブジェクトをルートとして GC を行う．この方法では，OS の機能を使っていないため，より少ないオーバヘッドで古い世代から新しい世代への参照を検出することが可能である．

しかし，これは参照の書き換えが起こる可能性があるすべての箇所で検出を行っているため，古い世代から新しい世代への参照だけでなく，古い世代から古い世代，新しい世代から新しい世代，新しい世代から古い世代への参照の書き換えが行われた場合にも，rb_gc_check_ref 関数が呼ばれてしまう．そのため，参照の書き換えが頻繁に起これば，それがオーバヘッドの要因となりうる．これについては，5.4 節で議論する．また，Ruby を C のライブラリで拡張しようとする場合にも rb_gc_check_ref 関数を使用しなければならず，拡張ライブラリの作成者への負担を増加させる．

4.3 殿堂入り問題

世代別 GC では AT の値をいくつに設定するかが GC の効率とメモリ使用量に大きな影響を与える．この値が大きすぎると，長寿命と見なしてもよいはずのオブジェクトが短寿命領域に存在し続け，なかなか GC の対象外とならず無駄なマーキングをしてしまう．また小さすぎると，実際には短寿命であるにもかかわらず殿堂入りされ，長寿命領域内でのごみ（Tenured Garbage；TG）となってしまい長寿命領域が無駄に消費されてしまう．長寿命領域がすべて使用されると長寿命領域も含めた GC を行う必要がある．長寿命領域では生きているオブジェクトが多く，この GC には長い時間がかかってしまうため，長寿命領域の GC は

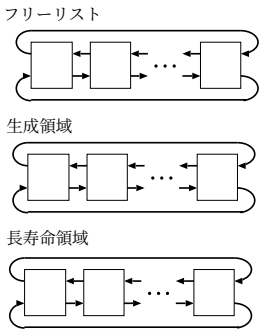


図 3 世代の構造
Fig. 3 Structure of generation.

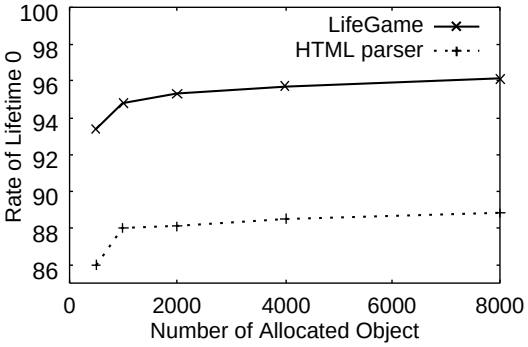


図 4 生存期間の変化
Fig. 4 Lifetime rate.

なるべく行わない方がよい．よって，長寿命領域の前の世代において，オブジェクトの大多数はごみとして回収し，長寿命領域でごみとなることを避ける必要がある．

本論文では，実装の容易さおよび 3.1 節の観測から，AT を 1 とし，生成領域と長寿命領域の 2 つの世代に分け，それぞれの領域を 1 つずつ用意し，フリーリストという領域を 1 つ用意した（**図 3**）．フリーリストには，まだ割り当てられていないオブジェクトがつながっており，GC によってごみになったオブジェクトはフリーリストにつながる．初期状態では，フリーリストに一定の個数（HEAP_SLOTS）のオブジェクトがつながっており，生成領域，長寿命領域にはなにもつながっていない．オブジェクトが割り当てられると，フリーリストから生成領域へつながり，フリーリストが空になると GC が起動する．GC を生き残ったオブジェクトは生成領域から長寿命領域へつながる．AT を 1 としたので，生成領域での GC を 1 回生き残ったオブジェクトはすべて長寿命領域へつながる．もし，フリーリストがある一定の個数（FREE_MIN）以下になったら，あらたに HEAP_SLOTS 個のオブジェクトをフリーリストにつなげ，オブジェクトの割当てを行う．

4.4 GC の起動

GC をいつ起動させるかは，重要な問題である．タイミングが早ければ TG は増え，タイミングが遅ければ，GC の処理に時間がかかる．

従来の方法（2.3 節）と同様に，HEAP_SLOTS を 10000，FREE_MIN を 512 としてしまうと，GC が起動してから，次の GC が起動するまでの間に割り当てられるオブジェクトの個数に，大きなばらつきが生じてしまう．このばらつきによって，オブジェクトの生存期間が変わってしまう可能性がある．そこで，GC が起動するまでに割り当てられるオブジェクトとオブジェクトの生存期間の関係を調べるため，GC が起動するまでに割り当てられるオブジェクト数を固定し，生成後すぐに回収されるオブジェクトの割合を計測した．**図 4** より，割り当てられるオブジェクトの個数が 2000 個ぐらいから，生成後すぐに回収されるオブジェクトの割合にあまり変化はない．

そこで本論文では，HEAP_SLOTS を 2500 個とし，FREE_MIN を 2000 個とした．こうすることで，オブジェクトの生存期間を一定に保ち，TG を抑えることができる．

しかし，この方法では，FREE_MIN が従来の Ruby の方法よりも大きいため，メモリ使用量が増加する可能性がある．

4.5 長寿命領域の GC

時間が経つにつれ，長寿命領域が使い尽くされてしまえば長寿命領域の GC を行わなければならない．長寿命領域は一般的に生成領域に比べて大きな領域が割り当てられており，長寿命領域の GC はめったに起こらないが，1 回起こると非常に時間がかかる．長寿命領域の GC をいつ行うかという問題が，メモリ使用量や GC の処理時間に影響を与える．

本論文では，以下の条件のとき長寿命領域の GC を行うこととした．

- オブジェクトの個数が 12500 個増加し，かつフリーリストの個数が 2000 個以下のとき
- 前回の長寿命領域の GC が起きて 20 回以上通常の GC が行われ，かつフリーリストの個数が 2000 個以下のとき

5. 評　　価

本論文の評価方法は，3.1 節と同様の評価環境で Ruby に本方式を実装し，3.1 節で用いたベンチマークプログラムを実行することで，従来方式との比較を行

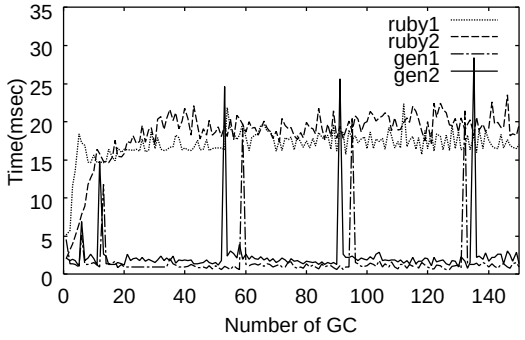


図 5 ライフゲームの 1 回の GC 時間
Fig. 5 GC time in life game.

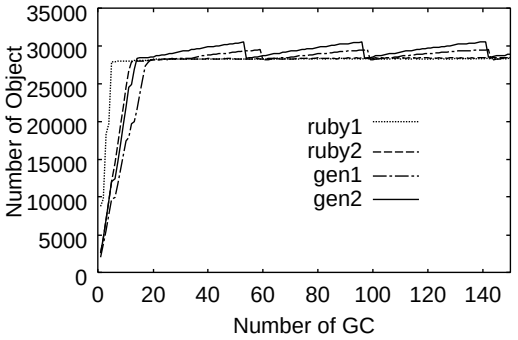


図 7 ライフゲームの生き残るオブジェクトの数
Fig. 7 Surviving objects of life game.

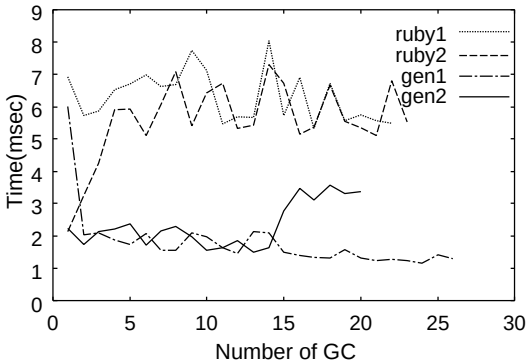


図 6 HTML パーサーの 1 回の GC 時間
Fig. 6 GC time in HTML parser.

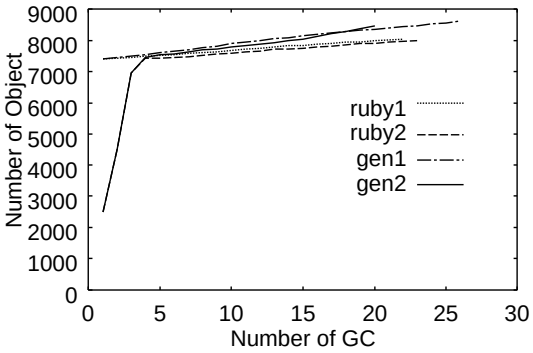


図 8 HTML パーサーの生き残るオブジェクトの数
Fig. 8 Surviving objects of HTML parser.

う．本方式では HEAP_SLOTS が 2500，FREE_MIN が 2000，従来方式では HEAP_SLOTS が 10000，FREE_MIN が 512 となっており，HEAP_SLOTS と FREE_MIN の値が異なっている．このことを考慮に入れ，GC は本方式だが，HEAP_SLOTS と FREE_MIN の値が従来方式と同じ場合，GC は従来方式だが，HEAP_SLOTS と FREE_MIN の値が本方式と同じ場合の評価も行う．以降，ruby1 は従来方式，gen2 は本方式を表し，ruby2 は GC が従来方式で，HEAP_SLOTS と FREE_MIN の値が本方式と同じ場合，gen1 は GC が本方式で，HEAP_SLOTS と FREE_MIN の値が従来方式と同じ場合を表している．

5.1 1 回の GC 時間の比較

図 5，**図 6** は横軸が GC の回数，縦軸はそのときの GC にかかった時間を表している．**図 7**，**図 8** は，横軸が GC の回数，縦軸はそのときの GC で生き残ったオブジェクトの数を表している．図 5，図 7 では，150 回以上から同様の傾向が見られるため，以降を省略した．

図 5 と図 7 から，従来方式の GC を実装している ruby1，ruby2 は，GC 1 回にかかる時間が生き残るオブジェクトの数に比例して大きくなっているのに対し，本方式の GC を実装している gen1，gen2 は生き残るオブジェクトの数にかかわらず，GC 1 回にかかる時間が短く，かつほぼ一定になっていることが分かる．これは本方式の GC が，通常，生成領域のみ GC の対象としているからである．同様のことが HTML パーサーを用いた評価（図 6 と図 8）でも確認できる．

図 5 において，本方式の 1 回の GC 時間が従来方式よりも長くなっている箇所があるが，これは長寿命領域の GC が行われた箇所である．長寿命領域の GC が行われる箇所では，本方式の方が従来方式よりも，1 回の GC の時間が長くなっている．これは本方式には，従来方式にはない双方向リストの付け換えの処理があるためだと考えられる．また gen1 に比べ，gen2 の方が長寿命領域の GC に時間がかかっている．これは，gen1 では長寿命領域の GC のとき処理するオブジェクトの数が 30000 個，gen2 では 32500 個となっており，gen2 の方が処理するオブジェクトの数が多いためだと考えられる．また図 5 と図 7 から，長寿

表 4 メモリ使用量
Table 4 Memory usage.

	ライフゲーム（KB）	HTML パーサー（KB）
ruby1	2720	292
ruby2	2792	344
gen1	3044	376
gen2	3432	516

表 5 GC の処理時間
Table 5 GC time.

	ライフゲーム（msec）	HTML パーサー（msec）
ruby1	8800	139
ruby2	3925	129
gen1	1212	51
gen2	719	47

表 6 メモリ保護の回数
Table 6 Number of memory protection.

プログラム	保護ポリシの変更	シグナルの発生
ライフゲーム	474902 (30500)	4992
HTML パーサー	8922 (8452)	301

括弧内はメモリ保護をかける回数を示す
単位はすべて回数

表 7 参照の書き換え回数
Table 7 Number of the write-barrier.

プログラム	参照の書き換え（回）
ライフゲーム	2704436 (1066)
HTML パーサー	290321 (132)

括弧内は長寿命領域から生成領域への
参照の書き換え回数を示す

命領域の GC が行われたときに，TG が回収されていることが確認できる．

図 6 において，ruby1 と gen1 の 1 回目の GC の時間が，ruby2 と gen2 に比べて長い．これは，ruby1 と gen1 の HEAP_SLOTS が大きいことが原因である．HEAP_SLOTS が大きいと，1 回目の GC が起きるまでに割り当てられるオブジェクトの数が多くなり，GC の時間が長くなる．また，gen2 では 15 回目の GC 以降，GC の時間がそれまでに比べ長くなっている．これは，15 回目の GC が起きた後，フリーリストへオブジェクトの追加が行われ，GC が起きるまでに割り当てられるオブジェクトの数が増えたことが原因と考えられる．

5.2 メモリ使用量の比較

表 4 は最大メモリ使用量を示しており，これには Ruby インタプリタ分は含まれていない．従来方式に比べ，本方式の最大メモリ使用量はライフゲームで 26%，HTML パーサーで 77%の増加となっている．原因としては以下のことが考えられる．

- 双方向リストのためのポインタ増加分
- FREE_MIN を 2000 にすることによって，プログラム実行中に使用するオブジェクトの数の増加
- TG が回収されないことで，一時的に無駄な領域が生じる

特にライフゲームに比べて，HTML パーサーにおける最大メモリ使用量の増加する割合が大きくなっている．これは，HTML パーサーの方がライフゲームに比べて，プログラム中に使用するオブジェクト数の増加する割合が大きいためだと考えられる．

5.3 GC 処理時間の比較

表 5 は GC の処理時間を比較したものである．ライフゲームで 92%，HTML パーサーで 67%，GC の処理時間を短縮している．これは本方式の GC が，通常，生成領域のみを GC の対象としているため，マークするオブジェクトの数が減ったことと，マークする領域が従来方式より少ないためページフォルトの起きる回数が減ったことで，GC の処理時間が短縮したものだと考えられる．

5.4 参照検出コストの比較

本論文で用いた古い世代からの参照を検出する方法と，OS の提供するシグナル機構を用いた方法の比較を行う．

シグナル機構を用いた方法では，1 つのオブジェクトで頻繁に参照の書き換えが行われると，それが大きなオーバヘッドとなりうる．そこで，保護をかけたオブジェクトに 1 回でも参照の書き換えが行われた場合には，その参照先にかかわらず保護を外し，次の GC のルートとする．また，生成領域から長寿命領域への移動，長寿命領域からフリーリストへの移動のときに，メモリ保護，もしくはメモリ保護ポリシの変更が行われるものとする．この方針をもとに，本方式を変更し，メモリ保護ポリシの変更回数，シグナルの発生回数の計測を行った．**表 6** にその計測結果を示す．また，本方式でのベンチマークプログラム実行中に起きた参照の書き換え回数を計測し，**表 7** に示す．

長寿命領域のオブジェクトへの書き込みを，シグナル機構を用いて検出すると，表 1 の環境ではマシンサイクルでおよそ 5000 サイクル必要である．一方，本論文で用いた方法では，オブジェクトが長寿命領域にあり，かつ参照の書き換え先が生成領域を指しているかどうかを調べるのに 7 サイクル，生成領域を指していた場合，次の GC のときそのオブジェクトをルートにするための準備におよそ 150 サイクル必要となる．また，オブジェクトにメモリ保護をかけるには，およ

表 8 参照検出にかかる時間
Table 8 Write-barrier time.

プログラム	シグナルを用いた方法	本論文の方法
ライフゲーム	1475	55
HTML パーサー	78	6

単位はすべて msec

表 9 実行時間
Table 9 Total execution time.

	ライフゲーム（sec）	HTML パーサー（sec）
ruby1	16.21	1.53
ruby2	11.08	1.51
gen1	8.56	1.47
gen2	8.03	1.45

そ 3000 サイクル，保護のポリシを変更するにはおよそ 900 サイクルかかる．これをもとに，参照検出にかかる時間を計算した．その結果を**表 8** に示す．

ライフゲーム，HTML パーサーとも，本方式のほうが参照検出にかかる時間が速いことが分かる．また，参照検出にかかる時間が大きなオーバヘッドにならないことが分かる．

5.5 実行時間の比較

表 9 はプログラムの実行時間を比較したものである．本方式の方が従来方式に比べて，ライフゲームで 50%，HTML パーサーで 5%減少している．これは，GC 処理時間の短縮分だけ，プログラムの実行時間が速くなったと考えられる．

6. 関 連 研 究

オブジェクト指向スクリプト言語における GC の効率化の手法は，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わるような小規模なプログラムにしか使われなかったため，最適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大規模なプログラム開発にも用いられ，実行時間に占める GC の割合が増えており，最適化の必要性がある．

Treadmill GC に世代別 GC の考えを取り入れた方法が，小池らによって提案されている[10]．オブジェクトの移動を，双方向リストの付け換えによって行う点は同じである．本方式は一括型 GC であるが，小池らの方式は実時間 GC であるという点が異なっている．また，このことからオブジェクトの管理方法が異なる．

小林らによって，GNU Emacs に世代別 GC を実装する研究が行われている[8),9)]．これは世代間の参照の検出に，仮想メモリのダーティビット情報を利用することで，GC の中断時間を短縮することを目的としている．オブジェクトの世代間の移動方法，世代間の参照の検出方法という点が本方式とは異なる．

7. 結論および今後の展望

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 92%，プログラムの実行時間が最大 50%短縮した．

本方式の特徴として，以下があげられる．

- Ruby ではオブジェクトの書き換えは必ずオブジェクトが実行することと，オブジェクト指向特有のコード共有を利用して，オブジェクトの書き換えを検出する箇所を限定していること．
- GC が起動するまでに割り当てられるオブジェクトの数を 2000 から 4500 までとし，生成後すぐに回収されるオブジェクトの割合を一定に保っていること．

今後の展望としては，以下があげられる．

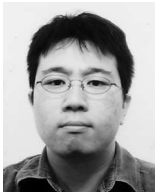
- 長寿命領域のごみの減少：
  オブジェクト指向スクリプト言語に最適な AT を設定する，もしくは，世代数を増やすことで，長寿命領域のごみを減少させる．こうすることで，メモリ使用量の減少，GC の処理時間の減少の効果が期待できる．
- 長寿命領域の GC 1 回の処理時間の短縮：
  長寿命領域の GC を時間的に分散させることで，長寿命領域の GC 1 回の時間を減少させる．こうすることで，インタラクティブなプログラムに対応することができる．
- 拡張ライブラリ作成者への負担軽減：
  拡張ライブラリで生成されるオブジェクトのみ長寿命領域に入れず，つねに GC の対象とすることで，拡張ライブラリで rb_gc_check_ref 関数を呼ぶ必要をなくす．こうすることで，拡張ライブラリ作成者の負担を軽減する．

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) Jones, R. and Lins, R.: *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley (1996).
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) http://www.ruby-lang.org/en/raa.html.
5) 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).
6) Baker, H.G.: The Treadmill, Real-time Garbage Collection without Motion Sickness, *ACM SIGPLAN Notices*, Vol.27, No.3, pp.66–70 (1992).
7) 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).
8) 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報告，97-PRO-16, pp.37–42 (1997).
9) 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：プログラミング，Vol.41, No.SIG4 (PRO 7), pp.24–31 (2000).
10) 小池龍信，岩井輝男，中西正和：オブジェクトの世代を考慮に入れたインクリメンタルなごみ集め処理，情報処理学会論文誌：プログラミング，Vol.40, No.SIG7 (PRO 4), pp.1–8 (1999).

(平成 12 年 7 月 14 日受付)
(平成 13 年 1 月 22 日採録)



木山　真人（学生会員）

昭和 51 年生．平成 11 年広島市立大学情報科学部情報工学科卒業．現在同大学院同研究科同専攻修士課程在学中．スクリプト言語，オブジェクト指向言語等に興味を持つ．