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実装手法の改良とその評価

木山真人[†] 佐原大輔[†] 津田孝夫[††]

オブジェクト指向スクリプト言語 Ruby を高速化するため，Ruby に世代別ガベージコレクション（GC）を実装する手法が提案されている．しかし，この方法では，(1) オブジェクトを双方向リストでつなげているため，メモリ使用量が多い，(2) 拡張ライブラリ作成者が参照検出をしなければならない，(3) ルートとなるオブジェクトが古い世代のオブジェクトを多数参照しているとき，無駄な追跡を多く行ってしまい，GC 処理時間が長くなるという問題があった．本論文で，この 3 つを解決する．(1)，(2) の問題を解決するため，新たな実装を行う．また，(3) の問題を解決するため，タイプ別ルート選択を提案する．タイプ別ルート選択とは，オブジェクトのタイプによってルートの場所を変更する方法である．これらの解決方法を実装し，性能評価を行った結果，本手法が有効であることが分かった．また，本手法はオリジナルの Ruby と比べ，GC 処理時間が最大 88.7%，プログラムの実行時間が最大 39.3%短縮することが分かった．

Improvement of the Implementation of Generational Garbage Collection in Object-oriented Scripting Language Ruby and Its Evaluation

Masato Kiyama,[†] Daisuke Sahara[†] and Takao Tsuda[††]

For high-speed processing of Ruby, we introduced generational Garbage Collection (GC) in object-oriented scripting language, Ruby. However, there are three problems with this implementation. (1) In order to move object, we use doubly linked list. To adopt this method increases memory usage. (2) To make extension libraries for Ruby, user must find the change of reference to objects. This makes the user's burden. (3) If most objects referred by root are in old generation, GC time becomes long because unnecessary trace occurs frequently. This paper shows how to solve the problem of (1) and (2). In order to solve the problem of (3), we propose type-based selection. Type-based selection is the method of selecting a place of root by the type of objects. We implemented generational GC with proposed method in Ruby. Compared with the former method, new proposed approach in this paper is effective. Compared with the original, the proposed approach can reduce 39.3% of total execution time and 88.7% of GC time on 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はじめている．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はじめている．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の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，著者ら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．高速化

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

のため，プログラム実行時間に占める割合が大きいガベージコレクション（以下，GC）に着目した．Ruby に世代別 GC を実装することで GC 処理時間を短縮し，実行時間の高速化を行った[4]．

しかし，文献 4) で実装した手法（以下，旧手法）では，いくつかの問題点がある．本論文では，これらの問題を解決する実装手法について述べ，その有効性を示す．また，オブジェクトの性質によってルートを選択する，タイプ別ルート選択を提案し，Ruby への実装手法について述べる．Ruby におけるタイプ別ルート選択の性能評価を行い，その有効性を示す．

2. Ruby の概要と GC の問題点

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であるなどの特徴がある．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．これらを組み合わせることで，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．

Ruby は GC 機能を有する言語である．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大量のオブジェクトを生成し，それらが複雑に関係付けられるため，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．

2.1 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．

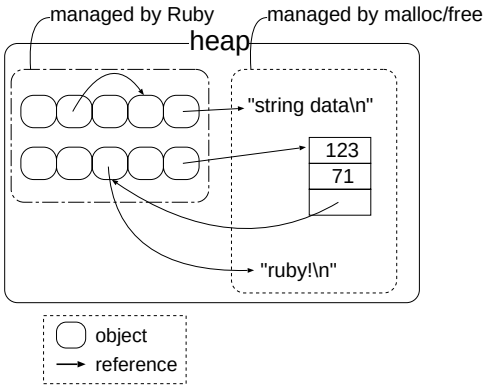

図 1 ヒープの構造
Fig. 1 The structure of the heap.

オブジェクトの割当てと GC は以下の手順で実行される．

(1) HEAP_SLOTS 個の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，HEAP_SLOTS 個の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．

HEAP_SLOTS と FREE_MIN は Ruby のソースコードで定義されている定数であり，それぞれ，10000 と 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．

たとえば，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．

```
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.

(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．

2.2 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．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. 世代別 GC を導入した旧手法の問題点

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．

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．そこで，著者らは Ruby に世代別 GC を実装し，GC 処理時間を短縮した[4]．

しかし，世代別 GC を導入した旧手法には問題点がある．以下で旧手法の問題点について述べる．

3.1 世代間移動方法の問題

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．

一般的に，オブジェクトの世代間移動は複写によって行われる．しかし Ruby では，オブジェクトの世代間移動を複写によって行うのは困難である．以下に理由をあげる．

- 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．

旧手法では，オブジェクトを双方向リストでつなぎ，双方向リストの付け換えと，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．双方向リストを用いてオブジェクトを移動させる手法は，Treadmill GC[9] と同様の手法である．

図 3 に旧手法での世代の構造を示す．旧手法での世代数は若い世代（young）と古い世代（old）の 2 つで

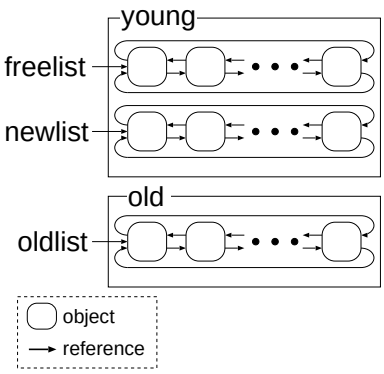

図 3 旧手法での世代の構造
Fig. 3 Structure of generation in the former method.

ある．freelist 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いる．オブジェクトを割り当てるときは，オブジェクトを freelist から若い世代のオブジェクトがつながる 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は oldlist に移す．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．また，プログラム実行中にごみとなったオブジェクトは freelist に移すことで，容易に再利用できる．

双方向リストを用いてオブジェクトを移動させる手法は，オブジェクトごとに双方向リストのデータ領域が必要となるため，メモリ使用量が多くなるという問題がある．

3.2 拡張ライブラリへの対応方法の問題

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．

旧手法では，拡張ライブラリ作成者が世代間参照を検出しなければならない．拡張ライブラリ作成者は世代間参照が行われる箇所を拡張ライブラリのソースコード中から探し，その箇所に世代間参照検出用のコードを挿入する必要がある．そのため，以下の問題がある．

- 拡張ライブラリ作成者への負担が増加する．
- 既存の拡張ライブラリを変更する必要がある．

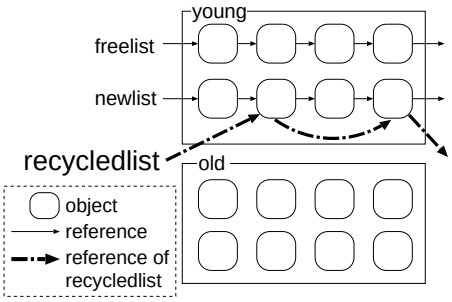

図 4 新手法での世代の構造
Fig. 4 Structure of generation in the new method.

4. 新 手 法

本章では，旧手法での問題を解決する方法について述べる．また，新手法で変更するヒープ領域拡張条件について述べる．

4.1 新たな世代間移動方法

3.1 節で述べたとおり，Ruby でオブジェクトの世代間移動を行うには，以下の 2 点を考慮する必要がある．

- 世代間移動によってオブジェクトのアドレスが変更されないようにする．
- プログラム実行中にオブジェクトを再利用可能にする．

旧手法では，この 2 点を考慮し，双方向リストでオブジェクトをつなぎ，世代間移動を行っている．しかし，この方法はオブジェクトごとに双方向リストのデータ領域が必要となるため，メモリ使用量が多くなるという問題がある．

この問題を軽減するため，新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．単方向リストでオブジェクトをつなぐため，旧手法と比べ，メモリ使用量が少なくなる．

図 4 に，新手法での世代の構造を示す．旧手法と同様に，新手法では世代を若い世代と古い世代の 2 つに分けている．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しない．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，新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 4）．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，オブジェクトを単方向リストでつなぐ方法でも，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．

4.2 新たな拡張ライブラリへの対応方法

3.2 節で述べたとおり，旧手法では拡張ライブラリ作成者が世代間参照検出用のコードを用いて世代間参照を検出している．そのため，以下の問題がある．

- 拡張ライブラリを作るユーザの負担が増加する．
- 既存の拡張ライブラリを変更する必要がある．

これらの問題を解決するため，新手法で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 の対象とすることで，世代間参照検出をする必要をなくす．世代間参照検出をする必要がなくなれば，拡張ライブラリで世代間参照検出のコードを挿入する必要がなくなり，旧手法での問題を解決することができる．

しかし，新手法では旧手法と比べ，1 回の GC 処理時間が長くなる可能性がある．これは，新手法では datalist につながったオブジェクトを毎回 GC の対象とするため，本来なら GC の対象外となるオブジェクトまで GC を行う可能性があるためである．これについては，6.1 節で議論する．

4.3 全領域 GC 開始とヒープ領域拡張条件の変更

Ruby に実装した世代別 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．

旧手法を実装したときの Ruby のバージョン 1.4.4 では，HEAP_SLOTS と FREE_MIN はそれぞれ 10000 と 512 である．そのため，GC が開始されるまでに割り当てられるオブジェクト数が大きく変化し，回収されるオブジェクトの割合に影響を与えている．そこで，旧手法では「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，HEAP_SLOTS と FREE_MIN をそれぞれ 2500 と 2000 とし，割り当てられるオブジェクト数を大きく変化しないようにし，回収されるオブジェクトの割合を一定に保つようにしている．

新手法を実装する Ruby のバージョン 1.7.0 では，HEAP_SLOTS はバージョン 1.4.4 と同じ 10000 だが，FREE_MIN は 4096 に変更されている．そのため，FREE_MIN を変更しなくても，回収されるオブジェクトの割合に影響がない．そこで，新手法では全領域 GC 開始条件とヒープ領域拡張条件を以下のように変更する．

- 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 は新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，旧手法の FREE_MIN と同様の理由である．また，HEAP_SLOTS と FREE_MIN はオリジナルの Ruby と同じである．

5. 提　　案

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．

そこで，旧手法では文献 10) と同様の方法であるリメンバードセットを用いて参照関係を管理している．リメンバードセットとは，若い世代のオブジェクトを参照している古い世代のオブジェクトを記憶するテーブルである．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，リメンバードセットをルートの一部として若い世代の GC を行う．リメンバードセットにオブジェクトを登録するときは，重複して登録されないようにチェックを行っている．

リメンバードセットを用いる旧手法では，リメンバードセットに登録されたオブジェクトが古い世代のオブジェクトを多数参照しているとき，無駄な追跡を多く行ってしまい，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ため，タイプ別ルート選択を提案し，新手法に実装する．

5.1 タイプ別ルート選択

リメンバードセットに登録されているオブジェクトから追跡を開始するとき，参照されているオブジェクトが若い世代のオブジェクトならマーキングする必要がある．しかし，参照されているオブジェクトが古い世代のオブジェクトなら追跡もマーキングもする必要はない．リメンバードセットに登録されているオブジェクトが古い世代のオブジェクトを多数参照しているならば，無駄な追跡が多く行われ，GC 処理時間が長くなってしまう．

この無駄な追跡をなくすため，タイプ別ルート選択を提案する．タイプ別ルート選択とは，オブジェクトのタイプによって，ルートの場所を変える方法である．

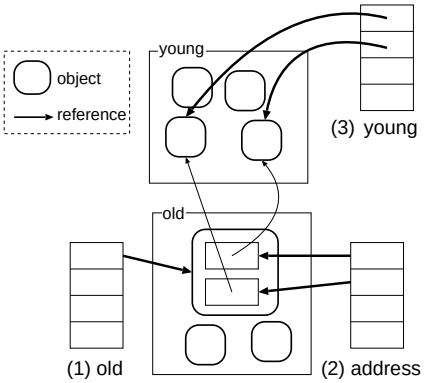

図 5 ルートの場所
Fig. 5 Three places of root.

オブジェクトには，他のオブジェクトへの参照を多数持つ性質と持たない性質，オブジェクトへの参照が頻繁に変化する性質と変化しない性質がある．他のオブジェクトへの参照を多数持つ性質があるオブジェクトがリメンバードセットに登録されると，GC のとき無駄な追跡が行われる可能性がある．リメンバードセットでは，ルートの場所として古い世代のオブジェクトを登録するため，このような無駄な追跡が行われてしまう．しかし，登録されるルートの場所を変えれば，無駄な追跡は行われない．

そこで，ルートの場所について考えてみる．ルートとして選択可能な場所は 3 つ（**図 5**）ある．それぞれの場所について利点と欠点を以下に述べる．

(1) old：古い世代のオブジェクト
ルートを格納するための領域が他の場所と比べて小さいという利点があり，無駄な追跡が行われる可能性があるという欠点がある．

(2) address：アドレス部分
無駄な追跡が行われないという利点がある．(1) と比べ，ルートを格納するための領域が大きくなるという欠点がある．

(3) young：若い世代のオブジェクト
(2) と同様の利点・欠点を持つ．若い世代のオブジェクトを追跡するとき，(2) では 2 回データをロードする必要があるが，この場所では 1 回という利点がある．しかし，ルートとなる若い世代のオブジェクトが古い世代から参照されなくなってごみとなる可能性があり，ごみを追跡してしまうという欠点がある．

このように，ルートの場所によって利点と欠点が異なる．そこで，タイプ別ルート選択では，この利点と欠点を利用し，オブジェクトの性質に合わせて，ルー

表 1 タイプの分類
Table 1 Classification of types.

	参照を多数持つ	持たない
参照が頻繁に変化する	アドレスタイプ	オールドタイプ
変化しない	ヤングタイプ	オールドタイプ

トの場所を選択する．

タイプ別ルート選択では，古い世代のオブジェクトをルートとするオブジェクトをオールドタイプ，若い世代を参照しているアドレス部分をルートとするオブジェクトをアドレスタイプ，若い世代のオブジェクトをルートとするオブジェクトをヤングタイプという．**表 1** に，オブジェクトの性質によって，どのタイプとして扱うかを示す．たとえば，オブジェクトへの参照が頻繁に変化せず，他のオブジェクトへの参照を多く持つようなオブジェクトはヤングタイプとして扱う．

分類の理由を以下に述べる．

- オールドタイプ
  他のオブジェクトへの参照を多く持たなければ，無駄な追跡が行われても，GC 処理時間に与える影響は少ない．そこで，ルートを格納するための領域を少なくするように，古い世代のオブジェクトをルートとする．
- アドレスタイプ
  他のオブジェクトへの参照を多く持つならば，無駄な追跡が行われないように若い世代を参照しているアドレス部分をルートとする．また，アドレス部分をルートとすれば，オブジェクトへの参照が頻繁に変化しても，ヤングタイプのように，ごみを追跡する可能性はない．
- ヤングタイプ
  オブジェクトへの参照が頻繁に変化しなければ，若い世代のオブジェクトをルートとしても，ごみを追跡する可能性がほとんどない．そこで，追跡のためのデータロードを少なくするため，若い世代のオブジェクトをルートとする．

このように，オブジェクトの性質によってルートの場所を変えることで，より高速な処理が可能になると考える．

5.2 Ruby へのタイプ別ルート選択実装

Ruby の配列・ハッシュオブジェクトは，他のオブジェクトへの参照を任意の数持つことが可能である．よって，配列・ハッシュオブジェクトは，アドレスタイプかヤングタイプのどちらかである．どちらのタイプか調べるため，6 章のテストプログラムを旧手法で実行し，古い世代にある配列・ハッシュオブジェクトから 1 回でも参照された若い世代のオブジェクトがすぐにごみとなる割合を求めた．結果を**表 2** に示す．表 2 の結果から，古い世代にある配列・ハッシュオブジェクトから若い世代のオブジェクトへの参照は頻繁に変化せず，ごみを追跡する可能性がほとんどないことが分かる．よって，Ruby の配列・ハッシュオブジェクトは，ヤングタイプだと考えられる．また，配列・ハッシュオブジェクト以外は他のオブジェクトへの参照を多数持たないため，オールドタイプである．

表 2 ごみの割合
Table 2 Rate of garbage.

テストプログラム	割合
life	0.4%
occur	0.1%
occur2	0.1%
compact	0%

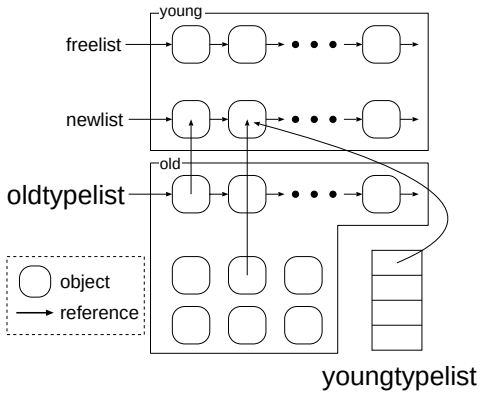

図 6 タイプ別ルート選択の構造
Fig. 6 The structure of the type-based selection.

Ruby にタイプ別ルート選択を実装するため，oldtypelist と youngtypelist という新しいリストを追加する（**図 6**）．oldtypelist はオールドタイプを管理するリストであり，youngtypelist はヤングタイプを管理するリストである．oldtypelist は，オブジェクトが持つ単方向リストのための領域を使ってつながっている．youngtypelist は，オブジェクトが newlist につながっているため，単方向リストのための領域が使えない．そのため，若い世代のオブジェクトへの参照を保持する領域を動的に確保している．また，oldtypelist も youngtypelist も重複してオブジェクトが登録されないようにチェックを行っている．

5.3 議　　論

Ruby におけるタイプ別ルート選択の実装では，配列・ハッシュオブジェクトをヤングタイプとする．本節では，この妥当性について議論する．

5.1 節で述べたとおり，ヤングタイプはアドレスタ

イプと比べ，追跡のためのデータロードが少なくなり，GC 処理時間が速くなると考えられる．しかしヤングタイプには，ごみを追跡してしまうという欠点がある．この欠点は，ごみを古い世代へ移動させ，古い世代の GC を引き起こす危険性がある．古い世代の GC が行われると GC 処理時間に影響を与えてしまう．もしこの影響がデータロードよりも GC 処理時間に影響を与えるのであれば，アドレスタイプのほうが GC 処理時間が速くなると考えられる．Ruby では，表 2 に示したとおり，古い世代の GC が起きる可能性はほとんどないことが分かる．そのため，Ruby ではごみを追跡するという欠点がデータロードよりも GC 処理時間に影響を与えることはないと考えられる．

また，Ruby で配列・ハッシュオブジェクトをアドレスタイプとして扱う場合，アドレス変更処理のオーバヘッドがあり，プログラムの実行時間に影響を与えると考えられる．Ruby では，配列オブジェクトの配列部分を拡張するとき，realloc を用いる．realloc はアドレスを変更するため，配列部分のアドレスがルートに保持されている場合，そのアドレスを変更しなければならない．ヤングタイプは若い世代をルートとするため，配列部分のアドレスが変更されても，このような処理は必要ない．

以上の理由から，Ruby で配列・ハッシュオブジェクトをヤングタイプとすることは妥当であり，アドレスタイプとするよりも高速にルートの処理ができると考える．Ruby で配列・ハッシュオブジェクトをヤングタイプとする場合とアドレスタイプとする場合の実際のデータロード差は 6.1 節，アドレス変更処理のオーバヘッドは 6.2 節で示し，議論する．

6. 性能評価

新手法を Ruby1.7.0（2001-05-11）版に実装し，その効果を測定した．評価環境を**表 3** に示す．以下の 6 方式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．

- ruby：オリジナルの Ruby.
- gen0：新手法を実装した Ruby．ただし，拡張ライブラリに対応していない．
- remb：すべてのオブジェクトをオールドタイプとして扱う Ruby．それ以外は，gen0 と同じ．
- addr：gen0 で，配列・ハッシュオブジェクトをアドレスタイプとして扱う Ruby.
- dgen：旧手法を実装した Ruby.
- gen1：新手法を実装した Ruby．拡張ライブラリに対応している．

表 3 評価環境
Table 3 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	

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 4 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 11) 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 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 2 万個．これは，Ruby 付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 8 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 5 万個．このプログラムは文献 12) で配布されている．

6.1 GC 処理時間

各方式の GC 処理時間を**表 4** に示し，結果について考察する．

タイプ別ルート選択の有効性を確認するため，gen0, remb，addr について比較する．remb はすべてのオブジェクトをオールドタイプとして扱うため，無駄な追跡が行われ，gen0，addr と比べ処理速度が遅い．addr では無駄な追跡は行われない．そのため，remb よりも処理速度が速い．しかし，配列・ハッシュオブジェクトをアドレスタイプとして扱うため，若い世代のオブジェクトを追跡するとき，データを 2 回ロードする必要があり，配列・ハッシュオブジェクトをヤングタイプとして扱う gen0 より処理速度が遅い．

次に，gen1 について考察する．gen1 は拡張ライブラリに対応している．そのため，拡張ライブラリで生成されるオブジェクトは毎回 GC の対象となるため，gen0 と比べ，処理速度が遅くなっている．しかし，拡張ライブラリ対応による処理速度の低下は少ないこと

表 4 GC 処理時間
Table 4 GC time.

	life	occur	occur2	compact
ruby	4912	1622	45113	1762
gen0	1610	500	5115	341
remb	2433	1199	38893	407
addr	1644	519	5195	342
dgen	2996	1821	76913	756
gen1	1637	509	5119	351

単位はすべて msec

表 5 実行時間
Table 5 Total execution time.

	life	occur	occur2	compact
ruby	37.69	7.50	99.52	5.65
gen0	34.55	6.43	59.66	4.18
remb	35.63	7.18	93.23	4.25
addr	34.77	6.54	62.16	4.21
dgen	36.11	7.85	131.71	4.62
gen1	34.61	6.47	60.45	4.21

単位はすべて sec

が分かる．また，拡張ライブラリを使用していない life 以外のテストプログラムでも gen0 よりも gen1 のほうが処理速度が遅くなっている．これは，Ruby が処理系内部において拡張ライブラリで生成されるオブジェクトを使用しているためである．

次に，dgen について考察する．dgen は処理速度が遅く，occur，occur2 ではオリジナルの方式よりも遅くなっている．この原因として以下の理由が考えられる．

- オブジェクトをすべてオールドタイプとして扱う．
- ヒープ領域拡張条件が他の方式と異なるため，GC が頻繁に行われる．

6.2 実行時間

各方式におけるテストプログラムの実行時間を**表 5** に示す．**図 7** は，オリジナルの Ruby の実行時間を 1 として，各方式での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 5，図 7 の結果について考察する．

タイプ別ルート選択の有効性を確認するため，gen0，remb，addr について比較する．remb は 6.1 節で述べたことが原因で，GC 処理時間が長くなり，gen0，addr と比べ実行時間が遅い．addr では 5.3 節で述べたとおり，アドレス変更処理のオーバヘッドがあるため，gen0 より実行時間が遅くなる．

次に，gen1 について考察する．6.1 節で述べたことが原因で，gen0 と比べ処理速度が遅い．しかし，拡張ライブラリ対応による処理速度の低下は少ない．

次に，dgen について考察する．dgen は occur，occur2 で，オリジナルの方式よりも実行時間が遅くなっている．これは 6.1 節で述べたことが原因である．

6.3 メモリ使用量

各方式の最大メモリ使用量を**表 6** に示し，結果について考察する．dgen2 は dgen のヒープ領域拡張条件を新手法と同様にした方式である．

まず，単方向リストでオブジェクトをつなげている

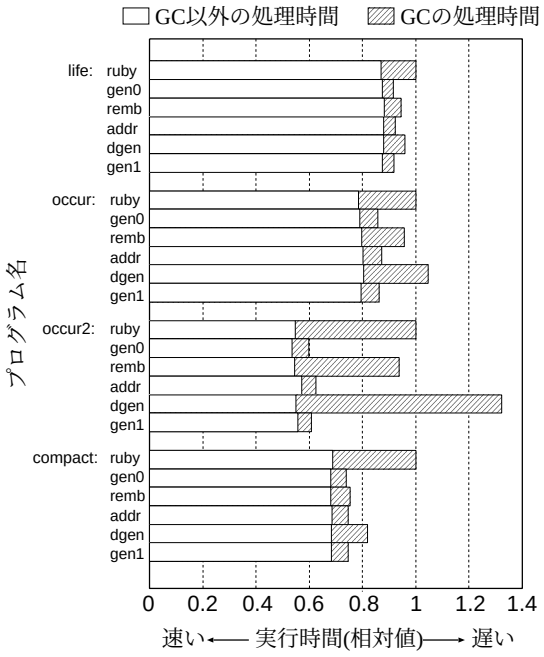

図 7 実行時間の比較
Fig. 7 Comparing execution time.

gen0，remb，addr，gen1 について考察する．各方式で，特に目立った違いはない．オリジナルと比較すると，life で 20%以下の増加であり，それ以外のテストプログラムでは，およそ 5%以下の増加となっている．オリジナルと比べ，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．

- 単方向リストのためのポインタ増加分．
- 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，オリジナルと比べ，無駄な領域が生じている．

次に，dgen について考察する．双方向リストでオブジェクトをつなげている dgen は，単方向リストでオブジェクトをつなげている方式と比べ，メモリ使用量が多くなるはずである．しかし，life，occur2 において，dgen は単方向リストでオブジェクトをつなげている方式と比べメモリ使用量が少ない．これは，dgen

表 6 メモリ使用量
Table 6 Memory usage.

	life	occur	occur2	compact
ruby	5700	2248	6148	4120
gen0	6696	2316	6468	4316
remb	6684	2320	6468	4300
addr	6700	2328	6472	4320
dgen	6288	2388	6296	4344
dgen2	6924	2388	6788	4516
gen1	6604	2332	6480	4320

単位はすべて KB

のヒープ領域拡張条件が他の方式と異なり，プログラム実行中に使用したオブジェクトの最大個数が少ないためである．そこで，プログラム実行中に使用したオブジェクトの最大個数が同じとなる dgen2 で比較する．単方向リストでオブジェクトをつなげている方式と比べ，dgen2 はオブジェクトごとのリスト 1 つ分だけメモリ使用量が多くなることが分かる．

7. 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，短時間で終わ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[13),14)]．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．

8. 結論および今後の展望

本論文では，文献 4) の実装手法の問題を解決する方法について述べた．また，タイプ別ルート選択を提案し，その有効性を示した．テストプログラムを用いて計測を行った結果，本手法はオリジナルの Ruby と比べ，GC 処理時間を最大 88.7%，プログラムの実行時間を最大 39.3%短縮した．

今後の展望として，以下があげられる．

- 古い世代のごみを減少：
  文献 15) 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．

謝辞 数々の示唆と助言をいただいた佐々木敬泰氏，Ruby の作者松本行弘氏，および論文に対する修正意見をいただいた査読者の方に心より感謝します．

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：プログラミング，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, *Comm. ACM*, Vol.26, No.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 Corpora-

tion, Western Research Lab (1989).
9) Baker, H.G.: The Treadmill, Real-time Garbage Collection without Motion Sickness, *ACM SIGPLAN Notices*, Vol.27, No.3, pp.66–70 (1992).
10) Ungar, D.M.: Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm, *ACM SIGPLAN Notices*, Vol.19, No.5, pp.157–167 (1984). Also published as ACM Software Engineering Notes 9, 3, *Proc. ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984, pp.157–167 (May 1984).
11) http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/~poffice/mail/ruby-list/6611
12) http://www.ruby-lang.org/en/raa.html
13) 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).
14) 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(PRO7), pp.24–31 (2000).
15) 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：プログラミング，Vol.41, No.SIG9(PRO8), pp.78–86 (2000).

(平成 13 年 5 月 31 日受付)
(平成 13 年 7 月 31 日採録)

木山 真人（学生会員）
1976 年生．1999 年広島市立大学情報科学部情報工学科卒業．現在同大学院博士課程在学中．スクリプト言語，オブジェクト指向言語の高速化手法に興味を持つ．

佐原 大輔
1976 年生．1999 年広島市立大学情報科学部情報工学科卒業．現在同大学院情報科学研究科情報工学専攻修士課程在学中．プログラミング言語処理系に興味を持つ．

津田 孝夫（正会員）
1957 年京都大学工学部電気工学科卒業．同大学工学部助手，助教授を経て，1972 年北海道大学工学部電気工学科教授．1979 年京都大学工学部情報工学科教授，1996 年同大学名誉教授，広島市立大学情報科学部教授．主な著書は「モンテカルロ法とシミュレーション」（培風社），「数値処理プログラミング」（岩波書店）．昭和 63 年および平成 3 年度情報処理学会論文賞受賞．ACM, SIAM 学会会員．磁気リコネクション，モンテカルロ法，コンパイラ技術，並列数値処理等に興味を持つ．