オブジェクト指向スクリプト言語Rubyへの世代別GCの実装と評価

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

木山 真人

目 次

第 1 章　はじめに **3**
1.1 研究の背景 3
1.2 研究の概要 4
1.3 本論文の構成 5

第 2 章　Ruby **7**
2.1 Ruby について 7
2.2 オブジェクトの管理方法 8

第 3 章　GC **10**
3.1 GC とは 10
3.2 ごみの判別方法による分類 11
3.2.1 トレースによる方法 11
3.2.2 リファレンスカウントによる方法 13
3.3 GC の性質による分類 13
3.3.1 世代別 GC 14
3.3.2 保守的な GC 15

第 4 章　Ruby の GC **16**
4.1 問題点 16
4.2 妥当性の調査 16
4.2.1 オブジェクトの生存期間 17
4.2.2 GC 処理時間の占める割合 18

第 5 章　実装 **20**
5.1 世代間の移動方法 20
5.2 世代間の参照検出 21
5.3 オブジェクトの殿堂入り 23

5.4 GC 起動とヒープ領域拡張のタイミング 23
5.5 拡張ライブラリへの対応 25
5.6 リサイクルドリストの追加 26

第 6 章 提案 **28**
6.1 ルート選択の問題 28
6.2 タイプ別ルート選択の提案 29
6.3 Ruby へのタイプ別ルート選択の実装 31

第 7 章 性能評価 **33**
7.1 評価環境 33
7.2 評価結果 33
7.3 考察 34
7.3.1 世代別 GC 適用の有効性 35
7.3.2 タイプ別ルート選択の有効性 37

第 8 章 関連研究 **43**

第 9 章 結論および今後の展望 **45**

第1章　はじめに

本研究では，オブジェクト指向スクリプト言語 Ruby[1, 3] のガベージコレクション (以下「GC」) に世代別 GC を実装した．また，タイプ別ルート選択を提案し，実装/評価を行なった．本章では，研究の背景，研究の概要，本論文の構成について述べる．

1.1　研究の背景

オブジェクト指向スクリプト言語は生産性と保守性が高いことから，今後のアプリケーション開発においてますます普及すると考える．現在，迅速なアプリケーション開発が求められている．迅速なアプリケーション開発を行なうためには，生産性と保守性が高い言語を用いてアプリケーション開発を行なうほうがよい．生産性と保守性が高い言語として，スクリプト言語がある．スクリプト言語にオブジェクト指向の考えを導入したオブジェクト指向スクリプト言語は，オブジェクト指向の考えを導入しているため，普通のスクリプト言語よりも生産性と保守性が高い．オブジェクト指向スクリプト言語を用いることで，迅速なアプリケーション開発が行なわれると考える．

オブジェクト指向スクリプト言語は，スクリプト言語とオブジェクト指向言語，両方の利点を持っているため，生産性と保守性が高い．スクリプト言語には，コード記述とプログラム実行の間の時間を短縮することができるという利点がある．また，スクリプト言語とオブジェクト指向言語には，既存のプログラムの再利用が容易，ソースコード量が少ないという利点がある．生産性を高めるためには，コード記述とプログラム実行間の時間短縮，既存のプログラムの再利用が重要である．保守性を高めるためには，プログラマが把握する部分のソースコード量の減少が重要である．

スクリプト言語は，コード記述とプログラム実行間の時間を短縮する．なぜなら，スクリプト言語は一般的にインタプリタ方式であり，ソースコードをコンパイルすることなしにプログラムの実行が可能だからである．特に，スクリプト言語は GUI を持ったアプリケーションなど，コード記述とプログラム実行を頻繁に繰り返すようなアプリケーション開発に向いている．コンパイル方式では，プログラムをコンパイルしないと実行できないため，インタープリタ方式に比べ，コード記述とプロ

グラム実行の間に時間がかかってしまう．

オブジェクト指向スクリプト言語は，既存のプログラムを再利用しやすい．Perl，Python，Ruby などの幅広く使われているスクリプト言語は，C による拡張機能を備えており，既存の C のライブラリをスクリプト言語側から使うことができる．既存の C のライブラリを使用することにより，スクリプト言語でその部分のプログラムを書く必要がなく，プログラムの実装時間の短縮につながる．また，オブジェクト指向言語ではクラスを継承する機構を用いることで，既存のプログラムを再利用することができる．オブジェクト指向言語は一般的にクラスを継承する機構が備わっている．既存のクラスだけでは対応できない問題があるときは，クラスを継承し，新しいクラスを作成する．既存のクラスを継承した新しいクラスでは，対応できない問題の部分のみを実装すればよい．

オブジェクト指向スクリプト言語は，プログラマが把握する部分のソースコード量が少ない．スクリプト言語で書いたプログラムは，C などで書いたプログラムに比べ，ソースコードの量が少ない．ソースコードをマシン語に直すと，ソースコード 1 行あたり，C はおよそ 5 行，スクリプト言語は 100 行ほどになるといわれている [2]．このことから，同じ作業をするプログラムを書いた場合，スクリプト言語で書いたプログラムのほうが C などで書いたプログラムに比べてソースコードの量が少ないと考えられる．また，オブジェクト指向言語では既存のクラスを使ってプログラムを書くと，既存のクラスの実装を把握する必要はなく，プログラマが把握するプログラムの部分が少なくなる．

1.2 研究の概要

本研究は，オブジェクト指向スクリプト言語の高速化が目的である．オブジェクト指向スクリプト言語を高速化するため，実行時間に与える影響が大きい GC に注目し，GC の高速化を行なう．本論文では，広く利用されているオブジェクト指向言語 Ruby に世代別 GC[12] を実装し，GC の高速化を行なう．さらに，世代別 GC をもとにオブジェクトのタイプに着目し，GC の処理時間短縮を目指すタイプ別ルート選択を提案する．これは Ruby に特化したものではなく，世代別 GC を実装している一般の言語に適用できる手法である．また，世代別 GC を実装した Ruby にタイプ別ルート選択を実装し，性能評価を行なう．

オブジェクト指向スクリプト言語の高速化は重要である．オブジェクト指向スクリプト言語は，生産性と保守性が高いことから，今後のアプリケーション開発において，ますます普及すると考える．しかし，オブジェクト指向スクリプト言語は，プログラムの実行速度が遅いという問題がある．この問題を解決するために，プログラムの実行時間に与える影響が大きい GC に注目し，GC を高速化することでプログラムの実行時間を高速化する．すなわち，GC を高速化することで，オブジェクト指向スクリプト言語を高速化する．

オブジェクト指向言語において，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オブジェクトを多数生成したり，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，オブジェクト指向スクリプト言語である Ruby 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で実装されている．

Ruby で大規模なアプリケーションを動かす場合，GC の処理時間が長くなるという問題がある．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しか使用されていなかったため，実行時間における GC 処理時間の占める割合は少なかった．しかし，Ruby はその生産性と保守性の高さから，小規模なプログラムだけでなく，大規模なプログラムにも使用されるようになった．一般に大規模なプログラムはオブジェクトを多数生成するため，Ruby の GC では実行時間における GC 処理時間の占める割合が大きくなる．これは，Ruby の GC がマークスイープ法であり，GC の処理時間がオブジェクト数に比例してしまうためである．

大規模なプログラムでの GC の処理時間を短縮するため，Ruby に世代別 GC を実装する．世代別 GC は「生成されたオブジェクトのほとんどは寿命が短い」という性質を利用し，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 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 GC の対象とする方法である．世代別 GC では，プログラムの規模が大きくなっても，通常は若い世代の領域のみ GC を行なうため，1 回の GC 処理時間が短く，GC 処理時間の実行時間に占める割合が小さくなる．

世代別 GC では，ルート (root) から参照されているオブジェクトの大半が古い世代にあると，無駄なマーキングが行なわれ，GC 処理時間に影響を与えるという問題がある．そこで，この問題を解決する方法として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，(1) 参照元の古い世代のオブジェクトをルートとするか，(2) 参照先の若い世代のオブジェクトをルートとするか，どちらかを選択する方法である．参照先が頻繁に変化するような古い世代のオブジェクトがルートとなる場合は (1) を選択し，参照を多数持つような古い世代のオブジェクトがルートとなる場合は (2) を選択する．

1.3 本論文の構成

本論文は以下の各章から構成される．

第 2 章 Ruby

本論文で世代別 GC を実装するオブジェクト指向スクリプト言語 Ruby について簡単に述べ，Ruby におけるオブジェクトの管理方法について述べる．

第 3 章 GC

本論文で扱う手法を中心に，一般的な GC の手法について述べる．

第 4 章 Ruby の GC

Ruby の GC の問題点について述べる．また，ベンチマークプログラムを用いて Ruby の GC を評価することで，問題点を明らかにし，世代別 GC の適用することの妥当性を調査する．

第 5 章 実装

Ruby に世代別 GC を実装する場合の問題点とその解決方法を述べる．

第 6 章 提案

新たに提案するタイプ別ルート選択について述べる．タイプ別ルート選択の必要性と方法について述べ，Ruby にタイプ別ルート選択を実装する場合の方法について述べる．

第 7 章 評価

ベンチマークプログラムを用いて，GC 処理時間，メモリ使用量を測定し，本方式を評価する．また，他の方式との比較を行なうことで本方式の評価を行なう．

第 8 章 関連研究

本論文に関連のある 3 つの研究について述べる．

第 9 章 結論および今後の展望

本論文をまとめ，今後の課題を述べる．

第2章　Ruby

本章では，本論文で世代別 GC を実装する，オブジェクト指向スクリプト言語 Ruby について述べる．Ruby とはどのような言語であるかについて述べ，Ruby におけるオブジェクトの管理方法について述べる．

2.1　Ruby について

Ruby はオブジェクト指向スクリプト言語である．その処理系は C 言語で記述されている．特徴を以下に挙げる．

- すべての値がオブジェクト
  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．すべての値がメッセージを送る統一的なインターフェースを持つため，値によって特別な扱いをする必要がない．よって，既存のプログラムをクラスライブラリとして再利用できる可能性が増える．

- 変数や式に型がない
  静的に型が決定される言語では，変数の型宣言や式の型変換がプログラムを複雑にし，わかりにくくすることがある．しかし，Ruby には変数や式に型がないため，プログラムが複雑になることはない．Ruby では，オブジェクト自身が自分の型に合わせて実行時に適切な処理を行なう．また，式に型がないため，変数だけでなく配列なども任意の値を格納することができ，柔軟なデータ構造を表現することができる．

- 豊富なクラスライブラリ
  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．これらのライブラリで提供される機能を組み合わせて用いることで，さまざまな用途のプログラムを書くことができる

- C による拡張機能
  C 言語用の API (Application Programming Interface) があり，Ruby から C で記述された関数を使用したり，C から Ruby の機能を使用したりすることが

容易にできる．この拡張機能を使えば，既存の C のライブラリを Ruby から使用でき，柔軟で高速なアプリケーションを構築することができる．また，Ruby で作ったアプリケーションで処理時間がかかる部分を，C で書いたプログラムに置き換えることもできる．

- 移植性が高い
  Ruby の処理系はすべて C 言語で記述されているため，可搬性が高く，多くの種類の UNIX，DOS，Windows 上で動作している．

2.2 オブジェクトの管理方法

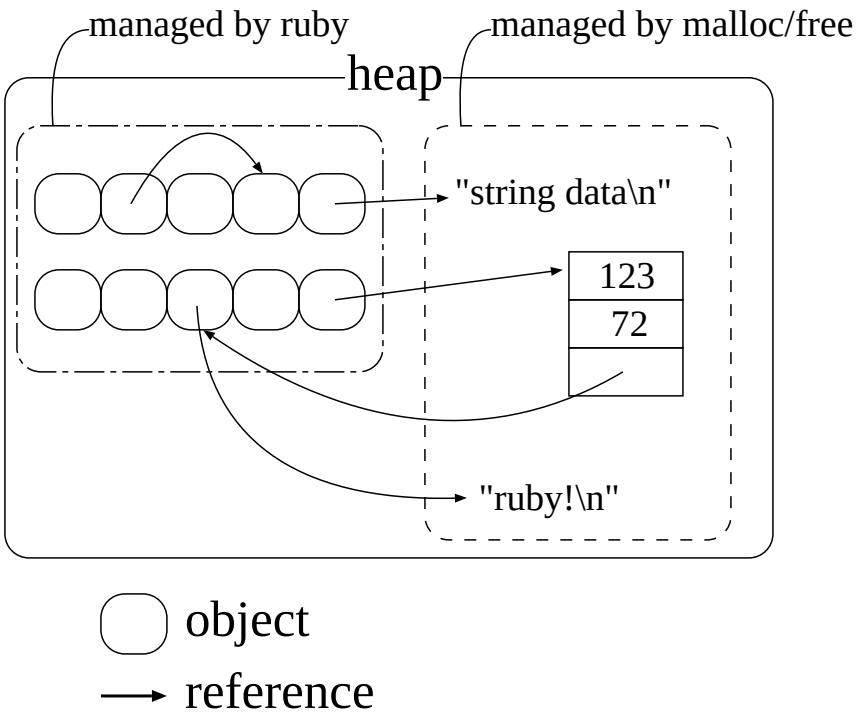

図 2.1: ヒープ構造

図 2.1 に，Ruby におけるヒープ構造を示す．オブジェクトの割り当て/解放などは Ruby が管理し，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 が管理している．Ruby では管理しやすいように，すべてのオブジェクトを一定の大きさに固定している．オブジェクトは，そのオブジェクトのクラス，オブジェクトのタイプ，マーク用のフラグ，文字列へのポインタ，配列の長さなどを持っている．オブジェクトが一定の大きさに固定されているため，文字列や配列などの可変部分のデータはオブジェクトに含まれず，オブジェクトから参照されている場所にある．

オブジェクトの管理は以下の手順で行なわれる．

1. オブジェクトの配列をヒープ領域から確保する．
2. 配列の各要素をリストとしてリンクする．この未使用のオブジェクトのリストをフリーリストと呼ぶ．
3. プログラムの実行に従って，オブジェクトの生成が必要となると，フリーリスト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なう．
4. フリーリストのオブジェクトを使い切るとGCが行なわれ，不要となったオブジェクトは可変部分のデータの解放などが行なわれたのち，フリーリストにリンクされる．
5. GC終了後，領域が不足していると判断したら，新たにオブジェクトの配列をヒープ領域から確保し，そのオブジェクトをフリーリストにリンクする．

第3章　GC

本章では，本論文で扱う手法を中心に，一般的なGCの方式について述べる．GCの方式は，不要なオブジェクトをどうやって判別するかという「ごみの判別方法」とどのような条件でGCを行なうかという「性質」の2面から分類できる．ごみの判別方法と性質は，ほぼ直交する概念であり，それぞれの判別方法に対して，さまざまな性質を備えた研究および実装がある．以降，GCとは何かについて述べ，ごみの判別方法とGCの性質について述べる．

3.1　GCとは

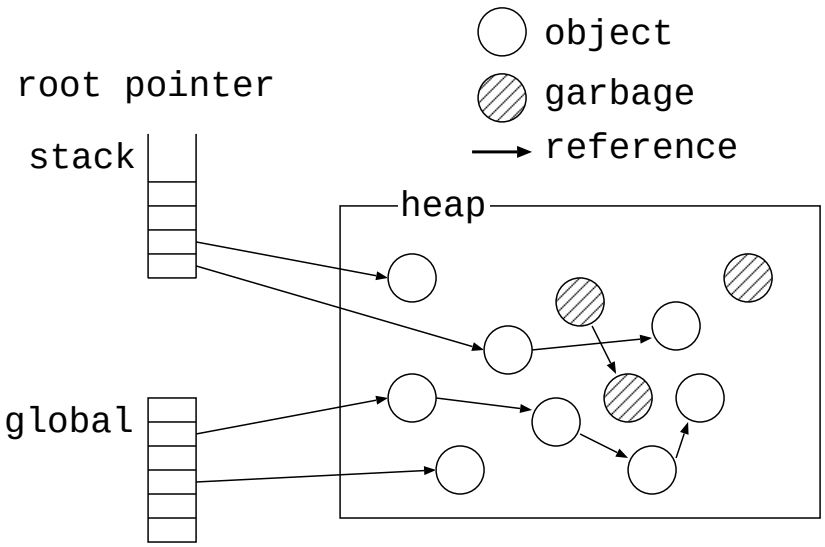

図 3.1: ガベージコレクション

GCとは，動的に確保したメモリ上のオブジェクトのうち，不要なオブジェクトを自動的に検出/解放/再利用するための機構である．GCは不要なオブジェクトを検出するため，プログラム実行中に存在するすべてのオブジェクトについて参照関係を調べる．そして，将来的に参照されることのないと判断したオブジェクトをごみとして回収する．

参照関係を調べる出発点となるのが，ルートポインタと呼ばれるポインタ群であ

る．ルートポインタは実行中のプログラムのスタック領域，ヒープ領域，CPU レジスタ，グローバル変数領域にあるポインタの集合である．ルートポインタで指されるオブジェクトは将来参照される可能性がある．また，ルートポインタから指されているオブジェクトの中にポインタが含まれていた場合，そのポインタが指しているオブジェクトも将来参照される可能性がある．以下同様に，参照される可能性のあるオブジェクトの中に含まれるポインタで指されるオブジェクトは，すべて参照される可能性がある．この手順で探索を続けていき，参照される可能性がないとされたものが不要なオブジェクトとなる（図 3.1）．

3.2 ごみの判別方法による分類

ごみの判別方法は，以下の 2 つに大別できる．

- トレースによる方法
- リファレンスカウントによる方法

以降，トレースによる方法，リファレンスカウントによる方法について述べる．

3.2.1 トレースによる方法

トレースによる GC とは，ルートポインタからオブジェクトをたどっていき，たどれなかったオブジェクトをごみと判断して回収する方法である．この方法は，マークスイープ法とコピー法の 2 つに大別できる．

マークスイープ法

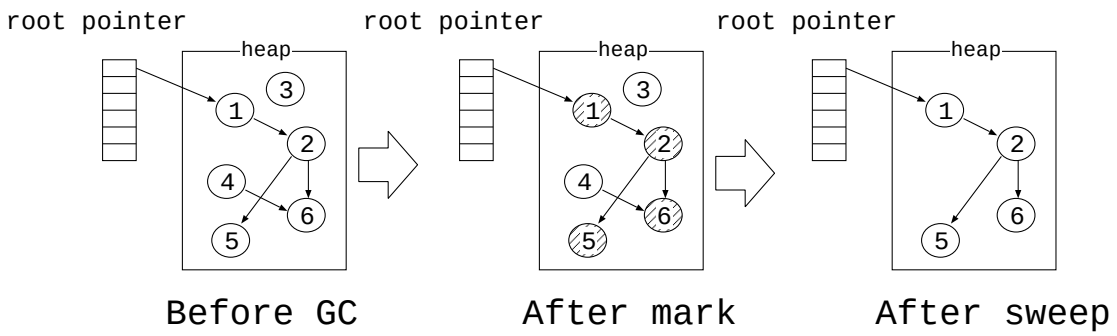

図 3.2: マークスイープ法

ルートポインタからオブジェクトをたどっていくときに，たどり着いたオブジェクトにマークをつけ，ルートポインタからたどれるすべてのオブジェクトにマークをつけたあと，メモリ領域全体を走査してマークのないオブジェクトをごみであると判断し，回収する方法を，マークスイープ法という (図 3.2) .

この方法は，オブジェクトにマークをつける作業と，マークが付いてないオブジェクトを回収する作業を行なうため，同じ領域を 2 度走査するという手間がかかる . また，メモリ領域全体を走査するため，メモリ領域が大きいと，GC の処理に時間がかかってしまう .

コピー法

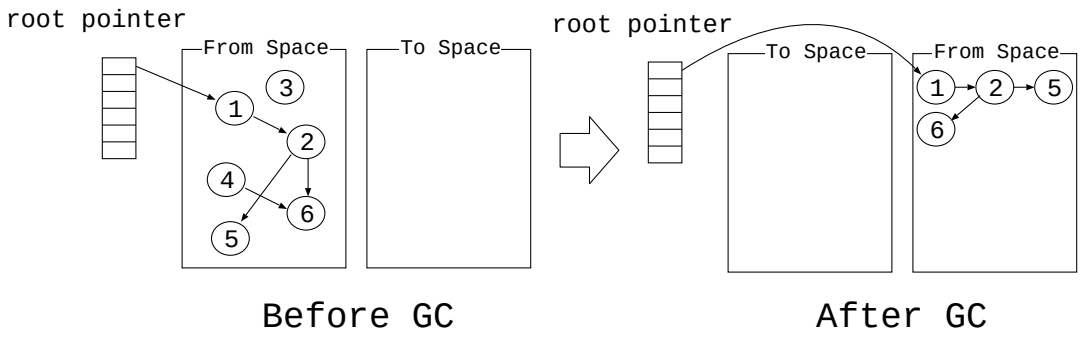

図 3.3: コピー法

ルートポインタからオブジェクトをたどっていくときに，たどり着いたオブジェクトを別の領域にコピーし，ルートポインタからたどれるすべてのオブジェクトをコピーしたあと，その領域に残っているオブジェクトをごみであると判断し，回収する方法を，コピー法という (図 3.3) .

この方法はマークスイープ法と異なり，メモリ領域全体を走査する必要がないため，GC の処理時間はコピーするオブジェクト，すなわち，必要なオブジェクトの数に比例する . また，オブジェクトをコピーするときに詰めて配置すれば，メモリ領域のフラグメンテーションを回避することができ，参照の局所性も向上する .

しかし，この方法はオブジェクトをコピーするため，オブジェクトの存在する場所が変化してしまい，そのオブジェクトを参照していたすべてのオブジェクトの内容を書き換える必要がある . また，メモリ領域を分けて使うため，実際に使えるメモリ領域はマークスイープ法に比べて少なくなる .

3.2.2　リファレンスカウントによる方法

自分を参照しているオブジェクト数を，オブジェクトがカウンタとして持っており，オブジェクトへの参照を操作するたびにカウンタを増減させ，カウンタが 0 になった時点でそのオブジェクトをごみであると判断/回収する方法を，リファレンスカウント法という．

この方法には，以下の利点がある．

- オブジェクトが不要になった時点で即座に回収できる
- そのオブジェクトの情報だけで，ごみになったかどうかの判断が可能

また，以下の欠点がある．

- 循環参照や自己参照などのループ状の参照関係を持つオブジェクト群を解放できない (図 3.4)
- オブジェクトが参照を変更するたびにカウンタの更新が必要となるため，参照の変更が頻繁に起きると管理コストが大きくなる
- GC のためのコードがプログラム全体に分散してしまい見通しが悪くなる
- オブジェクト内にカウンタのための領域が必要となる

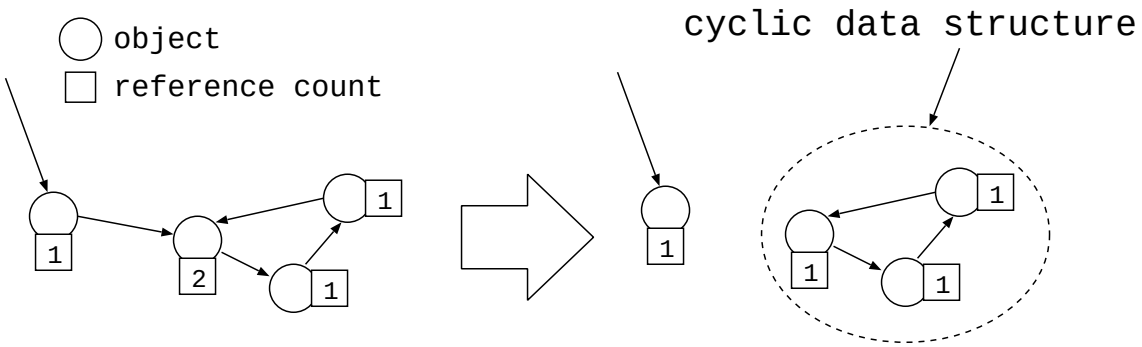

図 3.4: 循環参照

3.3　GC の性質による分類

GC の性質は，世代別 GC，保守的 GC，並行 GC，実時間 GC の 4 つに分類され，それぞれを組み合わせることが可能である．本節では，本論文で扱う方法の世代別 GC[9] と保守的 GC[10] について述べる．

3.3.1 世代別 GC

存在している期間でオブジェクトを分類し，その分類に基づいて異なる領域にオブジェクトを割り当てる方法を，世代別 GC という．

オブジェクトには「ほとんどのオブジェクトは生成後すぐに回収され，ある程度存在するオブジェクトはさらに長期に渡って存在する」という性質があることが知られている．世代別 GC では，ルートポインタから到達可能なオブジェクトを生きてるオブジェクトといい，ごみとなったオブジェクトを死んでいるオブジェクトという．また，ある程度生きたオブジェクトを長寿命であるといい，生成後すぐに死んでしまうオブジェクトを短寿命であるという．

マークスイープ法では，メモリ領域がすべて使用されるまで GC を行なわないため，短寿命のオブジェクトが多い場合は，死んでいるオブジェクトをメモリに保存してることになり，メモリを無駄に使用していることになる．コピー法では，マークスイープ法に比べ使用できるメモリ領域が少ないため，GC が頻繁に起きる．よって，短寿命なオブジェクトはすぐに回収されるが，長寿命なオブジェクトは GC のたびにコピーされるため，無駄にコピーしていることになる．これらの無駄を回避するには，最近生成された新しいオブジェクトのみ GC の対象とし，長寿命なオブジェクトを GC 対象外とすればよい．この考えに基づいた GC が世代別 GC である．

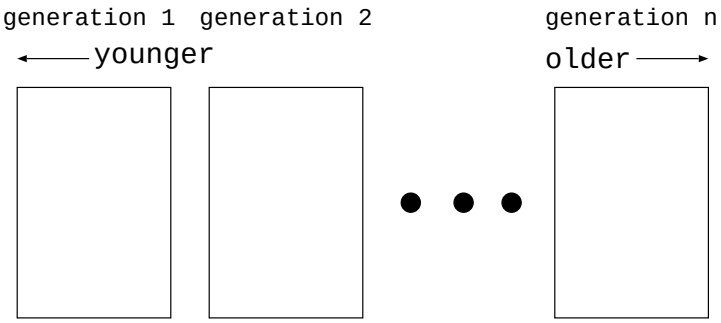

図 3.5: 世代別 GC のメモリ構造

世代別 GC では，オブジェクトの寿命に応じてオブジェクトを分類し，異なる領域に割り当てる（図 3.5）．オブジェクトの生成は，最も若い世代から行なわれる．GC をある一定回数生き残ったオブジェクトはひとつ上の世代に移動され，GC の対象外となる．通常の GC は最も若い世代のみを対象とするが，若い世代の GC をした後でも若い世代に空き領域が不足した場合，古い世代の GC を行なう．世代別 GC では，通常最も若い世代のみを GC の対象とするため，マークスイープ法に比べて 1 回の GC 処理時間が短いという特徴がある．

3.3.2 保守的なGC

ルートポインタが存在するスタック領域やメモリ領域で，ポインタと数値の区別ができない状況のもと行なわれるGCを，保守的なGCという．

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ポインタをルートとし，このルートから到達可能なオブジェクトを必要なオブジェクトと判断する．スタックやレジスタ上のポインタを他のデータと区別して管理するためには，Lispや Smalltalkで用いられている，タグを用いたポインタコーディングなどの手法を用いる必要がある．

C言語は，スタックやレジスタ上のポインタを他のデータと区別して管理していないことから，スタックやレジスタ上のルートを識別することが問題となる．

この問題を解決するには，ポインタとその他のデータを明確に区別せず，「ポインタと同様のビットパターンを持つデータをすべてポインタと判断する」ようにする．このようにすると，本当のポインタ以外にも一部のデータがポインタと誤認されてしまう．しかし，データがポインタと誤認された場合でも，誤認されたポインタによって参照される不要なオブジェクトが解放されずに残るだけで，必要なオブジェクトは必ず保持できるため，プログラムの実行に支障はない．ただし，誤認の起こったオブジェクトの分だけメモリが無駄に使用される．

第4章　RubyのGC

本章では，Ruby の GC の問題について述べる．その問題を解決するため，Ruby に世代別 GC の考えを導入することを検討する．ベンチマークプログラムを用いて評価を行なうことで，Ruby の GC に世代別 GC の考えを導入することの妥当性を調査する．

4.1　問題点

Ruby でオブジェクトを多数生成するようなアプリケーションを実行する場合，プログラムの実行時間に占める GC の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．マークスイープ法では，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不要なオブジェクトの回収に時間がかかるからである．

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．「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」という性質が，オブジェクトにあることが知られている．この性質を利用し，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，通常は寿命の短い世代の領域のみを頻繁に GC を行ない，領域が少なくなったときに長寿命領域の GC を行なうという手法が，基本的な世代別 GC の考え方である．世代別 GC では通常，寿命の短い世代の領域のみ GC を行なうため，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もない．

Ruby に世代別 GC の考えを導入することで，Ruby における GC の問題が解消されると考える．

4.2　妥当性の調査

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る舞いが「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

表 4.1: 評価環境

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 version 2.95.2	
ruby	1.6.2 (2000-12-25)	

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，オブジェクトの生存期間，GC 処理時間の占める割合の評価を行なう．

4.2.1 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する．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．評価環境を表 4.1 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．

- life: 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．140 世代まで計測．

- occur: 単語の出現頻度計測
  単語の出現頻度を調べるプログラム．Ruby のソースコードすべてを調べる対象とする．このプログラムは ruby-1.6.2.tar.gz の中にある．

- hanoi: ハノイの塔
  ハノイの塔を解く過程を HTML 形式で出力するプログラム．塔の数を 10 とする．このプログラムは文献 [4] で配布されている．

- parser: 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．入力するファイルは 240Kbytes の HTML ファイル．このプログラムは文献 [4] で配布されている．

- xml: XML のパーサー
  XML ファイルをパースして DOM ツリーを作るプログラムである．入力するファイルは 530Kbytes．このプログラムは文献 [4] で配布されている．

表 4.2: 生存期間の長さが 0 の割合

プログラム	割合
life	98.33%
occur	98.03%
hanoi	99.08%
parser	96.17%
xml	81.01%

表 4.3: GC 処理時間の占める割合

プログラム	割合	実行時間 (sec)	GC の処理時間 (msec)
life	27.64%	51.82	14321
occur	19.72%	7.24	1428
hanoi	10.56%	19.89	2101
parser	8.87%	2.30	204
xml	60.49%	26.49	16023

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 4.2 に示す．表 4.2 から，ほどんどのオブジェクトが生成後すぐに回収されていることがわかる．このことから，Ruby 上でのオブジェクトの振る舞いは「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．したがって，Ruby においても世代別にオブジェクトを管理することが有効であると考えられる．

4.2.2 GC 処理時間の占める割合

第 4.2.1 節で用いたベンチマークプログラムを使って，プログラムの実行時間に占める GC 処理時間の割合を計測した．評価環境は第 4.2.1 節と同様である．計測結果を表 4.3 に示す．

life や xml では実行時間に占める GC 処理時間の割合が大きく，parser や hanoi では小さいことがわかる．これは，life や xml 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，1 回の GC 処理時間が長くなり，実行時間に占める GC 処理時間の割合が大きくなるためだと考えられる．また，parser や hanoi は使用するオブジェクトの数が少ないため，1 回の GC 処理時間が短くなり，実行時間に占める GC の割合が小さくなるためだと考えられる．

表 4.3 の結果から，GC の処理時間を短縮することで，life や xml 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の短縮に大きな影響を与えることができると考えられる．

表 4.2，表 4.3 の結果から，Ruby の GC に世代別の考えを導入することは十分有効であり，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．

第5章　実装

本章では，Ruby に世代別 GC を実装するときに問題となった点と，その解決方法を述べる．Ruby に世代別 GC を実装する場合，

- 世代間の移動方法（第 5.1 節）
- 世代間の参照検出（第 5.2 節）
- オブジェクトの殿堂入り（第 5.3 節）
- GC 起動とヒープ領域拡張のタイミング（第 5.4 節）
- 拡張ライブラリへの対応（第 5.5 節）
- リサイクルドリストの追加（第 5.6 節）

が問題となった．

5.1　世代間の移動方法

世代別 GC ではオブジェクトを世代別に管理するため，オブジェクトを若い世代から古い世代へ移動させる必要があるが，オブジェクトの移動をどのように行なうかという問題がある．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，異なる領域に割り当て，通常はもっとも若い世代だけを GC の対象とする．若い世代で GC をある一定の回数生き残ったオブジェクトは長寿命と判断し，古い世代へ移動させなければならない．世代別 GC ではオブジェクトをコピーして世代を移動することが一般的である．

オブジェクトをコピーして世代間を移動することは，現在の Ruby で行なうには困難である．困難な理由は，Ruby の GC が保守的 GC であるためである．Ruby の処理系は C 言語で記述されているため，スタックやレジスタ上では数値とポインタの判別ができない．そのため，Ruby の GC は保守的な GC である．ヒープ領域では数値とポインタの判別はできるため，厳密にいうと，Ruby の GC は半保守的な GC である．そのため，ヒープ領域からのみ参照されているオブジェクトは，コピーして世代間を移動することが可能である．しかし，現在 Ruby ではオブジェクトのア

ドレス変化を考慮していない実装となっているため，コピーして世代間を移動しようとするとソースコードの大幅な変更が必要となる．

そこで，本論文では単方向リストを用いて世代間の移動を実現する．世代間の移動をコピーではなく，単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なう．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．しかし，この方法はオブジェクトごとに単方向リストのための領域を必要とするため，従来方式よりメモリ使用量が増加する．また，若い世代，古い世代の領域はそれぞれ 1 つずつ用意した．

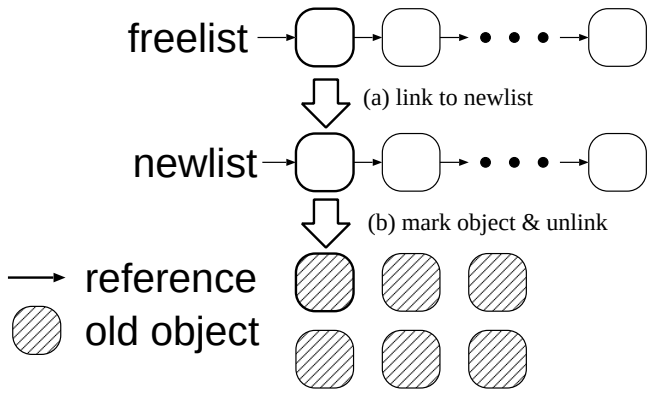

図 5.1: オブジェクトの移動

図 5.1 で世代間の移動方法を説明する．freelist は，未使用のオブジェクトが繋がったリストであり，newlist は，若い世代のオブジェクトが繋がったリストである．オブジェクトの生成，GC は以下の手順で行なわれる．

1. オブジェクトの生成が必要となったとき，オブジェクトを freelist から newlist に繋げる (図 5.1-a)．
2. freelist に繋がってるオブジェクトがなくなると，GC が行なわれる．
3. ルートから到達可能なオブジェクトにマークが付けられる．
4. newlist をたどって，マークが付いてないオブジェクトは freelist につなげ，マークが付いているオブジェクトは newlist から外される (図 5.1-b)．

5.2 世代間の参照検出

世代別 GC で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を参照したとき，それを検出しなければならないが，どのような方法でそれを検出するかが問題となる．世代別 GC では通常の GC のとき若い世代の領域のみ GC の対象とするため，古い世代のオブジェクトから参照されている若い世代のオブジェクトを必要なオブジェクトと判断しなければならない．古い世代のオブジェクトが若い世代のオ

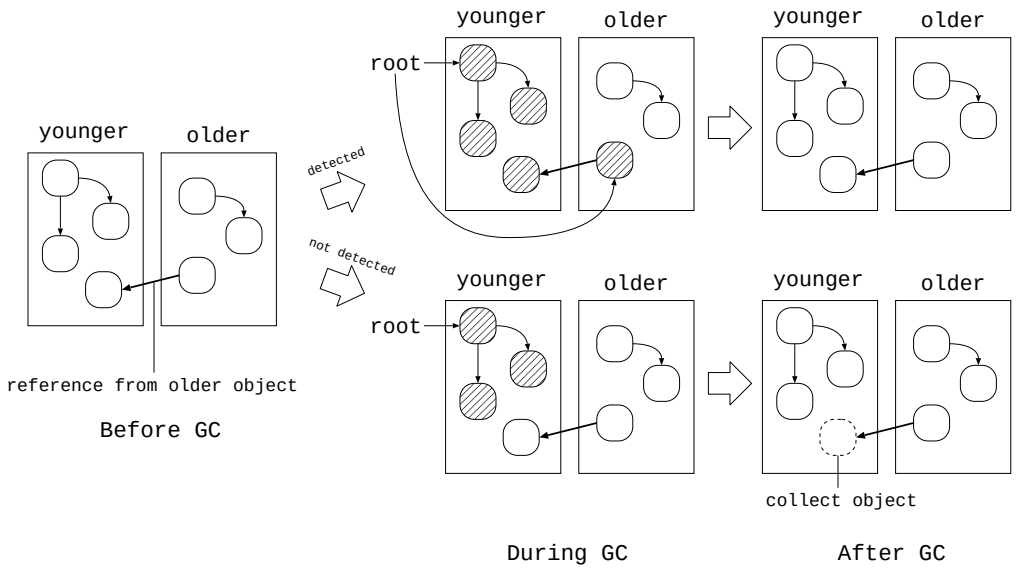

図 5.2: 参照の検出

ブジェクトを参照したことを検出し，次の GC のときその古いオブジェクトをルートとしなければ，古い世代のオブジェクト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，回収される可能性がある (図 5.2) .

世代間の参照を検出するため，文献 [8] では OS の提供するシグナル機構を利用している．この方法は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ため，ソースコードの変更箇所が少なくなる．しかし，OS の発するシグナルを利用しているため，世代間の参照を検出する時間が長くなる．この方法の手順を以下に示す．

1. 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．
2. OS が発するシグナルによって，その部分に書き込みが行なわれることを検出する．
3. 書き込みが行なわれるアドレスを調べ，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

世代間の参照を検出するため，文献 [5, 6, 7] では仮想メモリのダーティビット情報を利用している．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．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る．標準的なマシンでは 1 ページは 4096 バイトであるため，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，若い世代を参照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．また，この方法はダーティビット情報を利用するため，OS に強く依存する．Ruby は多く

の OS 上で動作するため，OS に強く依存する方法を用いることはできない．

本論文では，世代間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なうようにソースコードを変更する．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考慮すれば，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．Ruby のソースコード中で参照の書き換えを検出する箇所は 46 ヵ所ある．また，世代別 GC を実装するために追加したコードは 203 行である．

5.3 オブジェクトの殿堂入り

世代別 GC では若い世代のオブジェクトがある一定の回数の GC を生き残った場合，そのオブジェクトを古いの世代へ移動させる．この移動を殿堂入りといい，オブジェクトが殿堂入りするまでの GC の回数を Advancement Threshold （ AT ）という．AT をいくつにするかという問題がある．AT の設定は GC の処理時間とメモリ使用量に大きな影響を与える．AT の設定が大きすぎると古い世代とみなしてもよいはずのオブジェクトが若い世代に存在し続け，なかなか GC の対象外とならず無駄なマーキングが起きるため，GC の処理時間が長くなるという問題がある．また，AT の設定が小さすぎると古い世代でごみとなってしまうものが増え，1) 古い世代の領域が無駄に消費されてしまうため，メモリ使用量が増える，2) 古い世代も含めた GC が頻繁に起き，GC の処理時間が長くなるという問題がある．古い世代には生きているオブジェクトが多く，古い世代の領域は一般的に大きいため，古い世代の GC には長い時間がかかってしまう．そのため，古い世代の GC はなるべく起きないほうがよい．世代別 GC では，最適な AT の設定を行なうことが重要である．

本論文では，AT の設定を 1 とする．AT の設定を 1 とする理由は以下の 3 つである．

- 第 4.2.1 項の表 4.2 の観測結果から，AT の設定が 1 でも十分若い世代でオブジェクトを回収することが可能であると考える．
- AT の設定が 1 なら，オブジェクトが内部に持っているフラグを活用することができる．しかし，AT の設定を 2 以上にすると，オブジェクト内部のフラグに領域が余っていないため，AT 用の領域を新たに追加しなければならない．これはメモリ使用量を増加させてしまう．
- AT の設定が 1 だと若い世代に存在するオブジェクトは GC を 1 回生き残るとすべて古い世代へ移動するため，世代間の参照の管理が容易となる．

5.4 GC 起動とヒープ領域拡張のタイミング

GC 起動とヒープ領域拡張をいつ行なうかという問題がある．GC を行なうタイミングは，GC 処理時間とメモリ使用量に影響する．また，プログラムがオブジェクトを多数使用すれば，ヒープ領域を拡張しなければならない．世代別 GC には，若い世代の領域のみ GC の対象とする「通常の GC」とすべての領域を GC の対象とする「古い世代の GC」がある．通常の GC のタイミングが早いと，古い世代でごみとなるオブジェクトが増加する．通常の GC では若い世代のオブジェクトのみ GC の対象としているため，古い世代のオブジェクトのごみを回収することはできない．そのため，古い世代のごみを回収するには，古い世代の領域に GC を行なう必要がある．古い世代の領域は，一般的に若い世代の領域に比べて大きな領域が割り当てられているため，古い世代の GC を行なうと非常に時間がかかり，GC 処理時間が長くなる．しかし，古い世代の GC を行なわないと古い世代のごみが回収できず，メモリ使用量が増加してしまう．

従来方式では，フリーリストにリンクされているオブジェクトがなくなったとき，GC を行なっている．また，GC を行なった後，フリーリストにリンクされているオブジェクトが FREE_MIN 個以下となったとき，ヒープ領域から新たに HEAP_SLOTS 個分のオブジェクトの配列を獲得し，フリーリストにリンクすることでヒープ領域の拡張を行なっている．FREE_MIN と HEAP_SLOTS 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，4096 と 10000 である．

世代別 GC では，通常の GC のタイミングと古い世代の GC のタイミングの 2 つを決めなければならない．また，ヒープ領域拡張のタイミングも決めなければならない．これらのタイミングは GC 処理時間とメモリ使用量に影響する．GC 処理時間とメモリ使用量を考慮に入れて，本方式では，通常の GC，古い世代の GC，ヒープ領域拡張を以下のようなタイミングで行なっている．

- 通常の GC は，フリーリストにリンクされているオブジェクトがなくなったとき行なう．
- 古い世代の GC は，通常の GC 後，フリーリストにリンクしているオブジェクトが FREE_MIN_MINOR 個以下となったとき行なう．FREE_MIN_MINOR は新たに定義した定数であり，2000 である．
- ヒープ領域拡張のタイミングは，古い世代の GC 後，フリーリストにリンクしているオブジェクトが FREE_MIN 個以下となったとき行なう．ヒープ領域から新たに HEAP_SLOTS 個分のオブジェクトの配列を獲得し，フリーリストにリンクすることでヒープ領域の拡張を行なう．

FREE_MIN_MINOR は，小さすぎても大きすぎても GC 処理時間の増加やメモリ使用量の増加につながる．そこで，本方式では GC 処理時間とメモリ使用量を考慮に入れて，FREE_MIN_MINOR を 2000 としている．また，ヒープ領域拡張のタイミングは，早すぎても遅すぎても GC 処理時間の増加やメモリ使用量の増加につながる．本方

式では，オブジェクトを世代別に管理するため，従来方式のオブジェクトに比べて，リンク用の領域分だけオブジェクトが大きい．そのため，プログラム中に使用されるオブジェクト数が同数であれば，本方式の方がメモリ使用量は大きくなってしまう．メモリ使用量を同じにするためには，本方式のプログラム中に使用されるオブジェクト数を少なくすればよい．しかし，オブジェクト数が少ないと頻繁に GC が行なわれ，GC 処理時間が長くなってしまう．また，オブジェクト数が多いとメモリ使用量が大きくなってしまう．GC 処理時間とメモリ使用量を考慮に入れると，プログラム中に使用されるオブジェクトの数は従来方式と同数がもっともよいと考える．そこで，本方式は従来方式の GC と同様のオブジェクト数をプログラム中に使用できるようなタイミングでヒープ領域を拡張する．古い世代の GC は，従来方式の GC と同様に，すべての領域を GC の対象としている．そのため，古い世代の GC 後，フリーリストにリンクしているオブジェクトが FREE_MIN 個以下となったとき，ヒープ領域の拡張を行なえば従来方式と同様のオブジェクト数となる．

5.5 拡張ライブラリへの対応

拡張ライブラリで生成されるオブジェクトも GC の対象となるため，世代間の参照を検出しなければならないが，それにどのように対応するかという問題がある．速く実行したい部分を C で書いたプログラムに置き換えたり，既存の C のライブラリを利用するために，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．世代別 GC では，古い世代から若い世代への参照を検出しなければならない．Ruby の処理系内部では，第 5.2 節で述べた方法で世代間の参照を検出している．拡張ライブラリでも，世代間の参照を検出しなければならない．拡張ライブラリへ対応する方法として，以下の 2 つの方法が考えられる．

拡張ライブラリで検出 拡張ライブラリを作るユーザが，世代間参照検出のためのコードを使用する方法．以下に利点と欠点を挙げる．

- 利点
  - 処理系でオブジェクトを管理する方法に比べ，実現が容易．
- 欠点
  - 拡張ライブラリ作成者が世代間の参照を検出する箇所を見つけ，その箇所で世代間参照検出のためのコードを使用しなければならず，拡張ライブラリ作成者の負担となる．
  - 既存の拡張ライブラリを変更する必要がある．

処理系でオブジェクトを管理 処理系が拡張ライブラリで生成されるオブジェクトを特別なリストに繋いで管理する方法．拡張ライブラリで生成されるオブジェク

トを特別なリストを用いて古い世代に移動させず，毎回 GC の対象とするため，古い世代から若い世代への参照を検出する必要がなくなる．以下に利点と欠点を挙げる．

- 利点
  - 拡張ライブラリ作成者の負担がない．
  - 既存の拡張ライブラリの変更が必要ない．
- 欠点
  - 拡張ライブラリで検出する方法に比べて，管理が複雑．
  - 拡張ライブラリで生成されるオブジェクトを毎回 GC の対象とするため，マーキングの時間が拡張ライブラリで検出する方法に比べて，長くなる．

本論文では，処理系でオブジェクトを管理する方法によって，拡張ライブラリで生成するオブジェクトへ対応する．処理系でオブジェクトを管理する方法のほうが，拡張ライブラリ作成者への負担がないため，生産性の向上につながると考えられるためである．

5.6 リサイクルドリストの追加

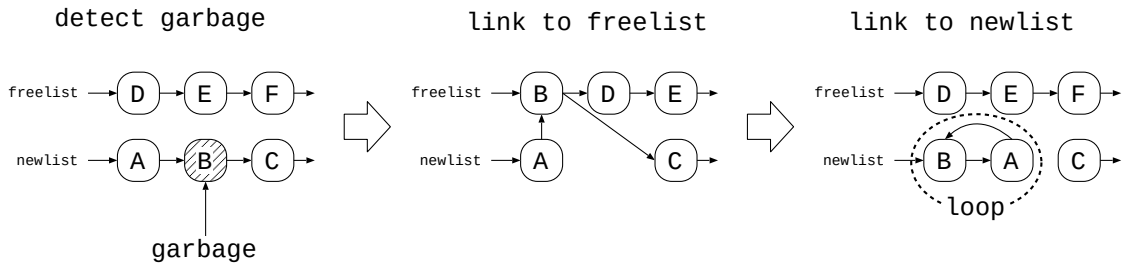

図 5.3: オブジェクト再利用問題

ごみとなったオブジェクトをどのように再利用するかという問題がある．Ruby では GC 以外のときにも，明確にごみと判断されたオブジェクトを回収し再利用している．従来方式では，ごみと判断されたオブジェクトを単にフリーリストに繋いで再利用している．しかし，この方法は本方式では問題となる．第 5.1 節で述べているように，本方式は世代間の移動を単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なっている．そのため，単にフリーリストに繋ぐだけでは問題がある．もし，newlist に繋がっているオブジェクトが明確にごみと判断され，そのオブジェクトがオブジェクトの生成に使われると，newlist にループができてしまう (図 5.3)．

この問題を解決する方法として，以下の 3 つの方法が考えられる．

- 単方向リンクではなく，双方向リンクでオブジェクトを繋ぐ
- ごみと判断されたオブジェクトを再利用しない
- newlist に繋がっているオブジェクトがごみと判断された場合，フリーリストではなく違うリストに繋ぐ

双方向リンクでオブジェクトを繋ぐ方法は，オブジェクトごとにリンクのための領域が必要となるため，単方向リストよりもメモリ使用量が増えてしまう．また，ごみと判断されたオブジェクトを再利用しない方法は，オブジェクトを再利用しないため，GC の回数が増えてしまい，GC 処理時間が長くなると考えられる．

そこで，本論文では新たにリサイクルドリストというリストを用意し，newlist にあるオブジェクトがごみと判断された場合，そのオブジェクトはリサイクルドリストに繋ぐ．

第6章　提案

本章では，新しく提案するタイプ別ルート選択について述べる．タイプ別ルート選択は，世代別 GC を実装している一般の言語に適用可能である．

6.1　ルート選択の問題

世代別 GC ではスタック領域やヒープ領域だけでなく若い世代を参照している古い世代のオブジェクトもルートとして GC を行なう必要があるが，多数のオブジェクトへの参照を持つ古い世代のオブジェクトをルートとして GC を行なう場合，そのルートの参照しているオブジェクトの大半が古い世代にあると無駄なマーキングで GC の処理時間が長くなるという問題が発生する．世代別 GC では，通常の GC のとき，若い世代の領域のみを GC の対象とするが，若い世代の領域外から参照されているオブジェクトも，必要なオブジェクトとして判断しなければならない．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た場合，なんらかの方法でこれを検出する．そして，若い世代の GC を行なうとき，古い世代のオブジェクトから参照される若い世代のオブジェクトを必要なオブジェクトとして判断しなければならない．この判断を行なう方法として，エントリーテーブルを用いる方法と，リメンバードセットを用いる方法がある．

エントリーテーブルを用いる方法とは，古い世代のオブジェクトから若い世代のオブジェクトを参照するときは，世代ごとにある間接参照のためのテーブルを使って参照し，GC のときはこのテーブルをルートの一部として，若い世代の GC を行なう方法である (図 6.1)．こうすることで，古い世代のオブジェクトから参照されている若い世代のオブジェクトは，必要なオブジェクトであると判断される．しかし，この方法は処理系が参照を操作するとき，直接参照か間接参照か判定しなければならず，そのオーバーヘッドを無視することができない．この方法は，専用のハードウェアでオーバーヘッド部分を短時間で処理できる Lisp マシンで採用されている．

リメンバードセットを用いる方法とは，古い世代のオブジェクトから若い世代のオブジェクトを参照するときは，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を用意し，GC のときはこのテーブルをルートの一部として，若い世代の GC を行なう方法である (図 6.2)．こうすることで，古い世代のオブジェクトから参照されている若い世代のオブジェクトは，必要なオブジェ

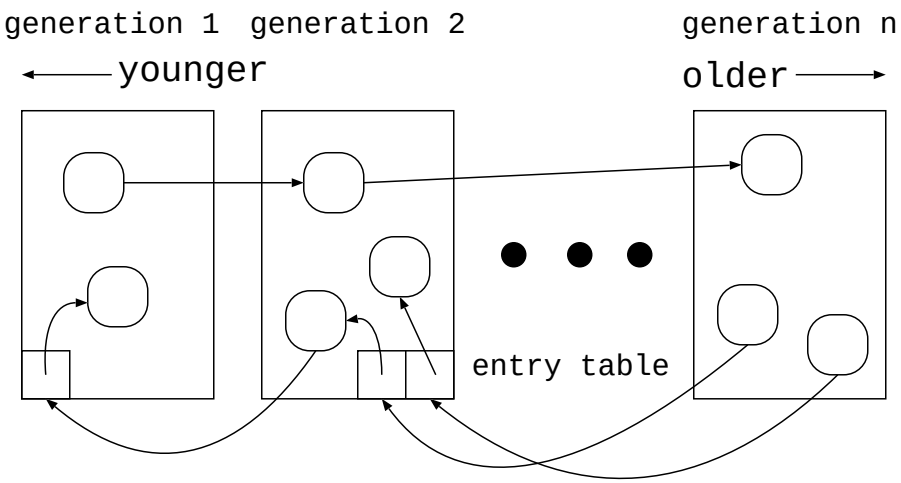

図 6.1: エントリーテーブル

クトであると判断される．この方法は，処理系が参照を操作するとき，直接参照か間接参照か判定する必要がないため，オーバーヘッドが少ない．この方法は，オーバーヘッドが少ないため，汎用機で用いられる方法である．

Ruby ではオーバーヘッドが少ないという理由により，リメンバードセットを用いる方法を採用する．しかし，リメンバードセットを用いる方法は，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 GC 処理時間が長くなるという問題がある．ルートから参照されているオブジェクトは GC のときすべてマーキングの対象となる．ルートから参照されている若い世代のオブジェクトはマーキングする必要があるが，ルートから参照されている古い世代のオブジェクトはマーキングする必要はない．もしルートから参照されているオブジェクトの大半が古い世代のオブジェクトなら，必要のない無駄なマーキングを多く行なってしまい，GC 処理時間が長くなってしまう．

6.2 タイプ別ルート選択の提案

ルートが参照しているオブジェクトの大半が古い世代にあるときに起きる無駄なマーキングをなくすため，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，参照先の若い世代のオブジェクトをルートとするか，参照元の古い世代のオブジェクトをルートとするか，どちらかを選択する方法である．参照先の若い世代のオブジェクトをルートとする方法と参照元の古い世代のオブジェクトをルートとする方法の利点と欠点を以下に挙げる．

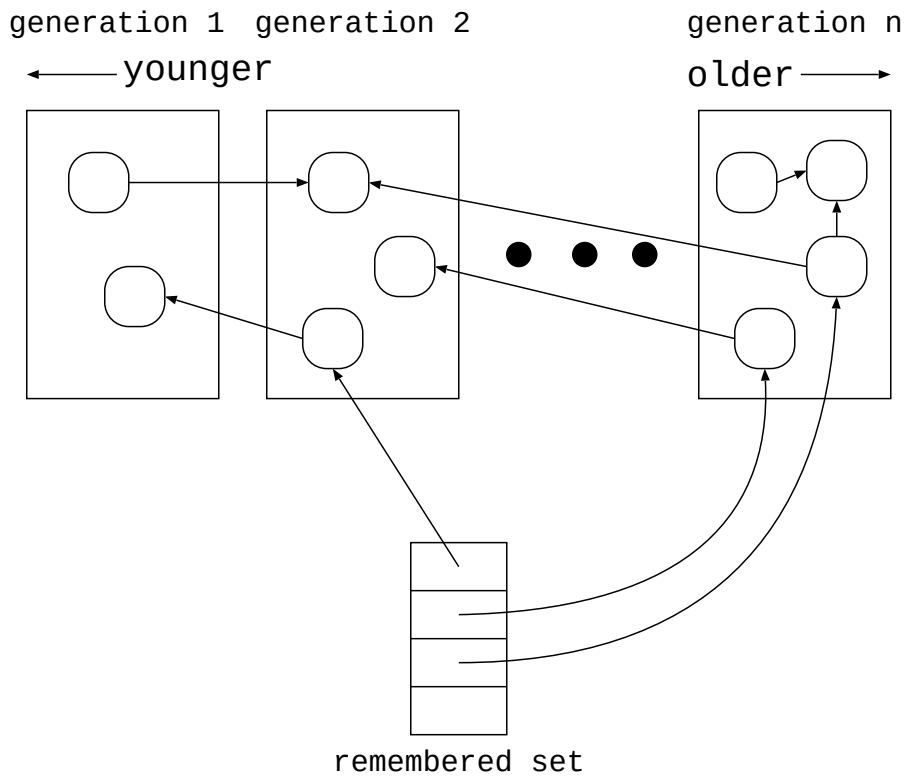

図 6.2: リメンバードセット

- 参照先の若い世代のオブジェクトをルートとする方法
  - 利点
    - ルートが参照しているオブジェクトの大半が古い世代にあっても，無駄なマーキングが起きない．
  - 欠点
    - 参照先が頻繁に変化するようなオブジェクトの参照先をルートとすると，若い世代の GC が起きたとき，そのルートがすでにごみとなっている可能性がある．
    - 参照先が頻繁に変化すると，管理にコストがかかる．
- 参照元の古い世代のオブジェクトをルートとする方法
  - 利点
    - 参照先が頻繁に変化しても，管理にコストがかからない．
  - 欠点
    - ルートが参照しているオブジェクトの大半が古い世代にあったとき，無駄なマーキングが起きる．

従来の世代別 GC の実装は，参照先の若い世代のオブジェクトをルートとする方法，参照元の古い世代のオブジェクトをルートとする方法のどちらかで実装されていた．しかし，参照が頻繁に変化するようなオブジェクトがあると，参照先の若い世代のオブジェクトをルートとする方法では問題があり，参照しているオブジェクトの大半が古い世代にあると，参照元の古い世代のオブジェクトをルートとする方法では問題がある．参照が頻繁に変化するようなオブジェクトには参照元の古い世代のオブジェクトをルートとする方法を適用し，参照しているオブジェクトの大半が古い世代にあるオブジェクトには参照先の若い世代のオブジェクトをルートとする方法を適用することで，この問題が解決できる．

6.3 Ruby へのタイプ別ルート選択の実装

第 6.2 節で提案したタイプ別ルート選択を Ruby に実装した．本節では，その実装方法について述べる．

Ruby では，配列/ハッシュオブジェクトのみ，参照先の若い世代のオブジェクトをルートとし，それ以外は参照元の古い世代のオブジェクトをルートとした．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．ゆえに，配列・ハッシュオブジェクトがルートとなるとき，参照しているオブジェクトの大半が古い世代にある可能性がある．配列・ハッシュオブジェクト以外のオブジェクトは，他のオブジェクトへの参照を限られた数しか持つことができない．ゆえに，配列・ハッシュオブジェクト以外がルートとなるとき，参照しているオブジェクトの大半が古い世代にある可能性はない．また，参照しているオブジェクトが古い世代にあるとしても，その数は少ない．このことから Ruby では，配列・ハッシュオブジェクトがルートとなるときは参照先の若い世代のオブジェクトをルートとし，配列・ハッシュオブジェクト以外がルートとなるときは参照元の古い世代のオブジェクトをルートとする．

Ruby にタイプ別ルート選択を実装するために，refreshedlist と referencedlist という新しいリストを追加した (図 6.3)．refreshedlist と referencedlist の説明と，ルートとなるオブジェクトがどのように管理されるか，を以下に挙げる．

refreshedlist 配列・ハッシュオブジェクト以外のルートとなるオブジェクトを管理するリストである．参照元となるオブジェクトが，refreshedlist に繋がる．ルートの管理方法を以下に挙げる．

- 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，古い世代のオブジェクトが配列・ハッシュオブジェクトでないことを判断する．
- 古い世代のオブジェクトから，古い世代を表すマークを外す．
- そのオブジェクトを refreshedlist に繋げる．

referencedlist ルートとなる配列・ハッシュオブジェクトを管理するリストである．参照先のオブジェクトを参照するリスト．ルートの管理方法を以下に挙げる．

- 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，古い世代のオブジェクトが配列・ハッシュオブジェクトであることを判断する．
- 若い世代のオブジェクトへの参照を referencedlist に保持する．

refreshedlist に繋がっているオブジェクトと referencedlist から参照されているオブジェクトをルートとして，若い世代の GC を行なう．refreshedlist は，第 5.1 節で述べた単方向リストのための領域を使って繋がっている．しかし，referencedlist は，オブジェクトが newlist に繋がっているため単方向リストのための領域が使えず，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．

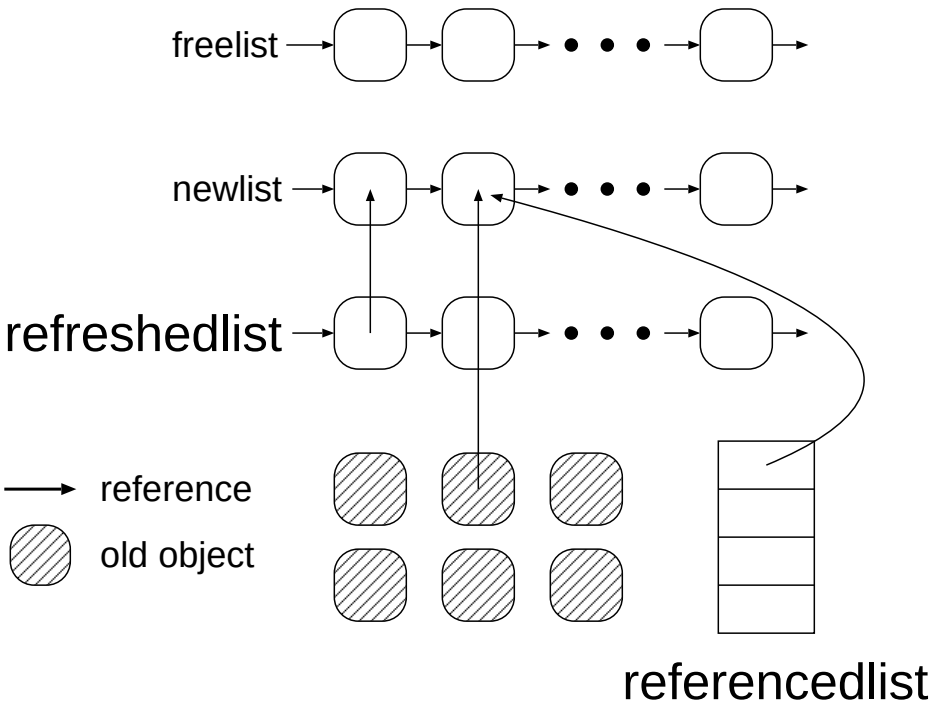

図 6.3: refreshedlist と referencedlist

第7章　性能評価

本章では，世代別 GC とタイプ別ルート選択を実装した，本方式の性能を評価する．従来方式に比べ本方式のほうが，実行時間は最大 37.6%減少，GC の処理時間は最大で 82.2%減少，メモリ使用量は最大で 12.6%増加した．

7.1　評価環境

第 4.2.1 項と同様の評価環境で，第 4.2.1 項の 5 つのベンチマークプログラムを実行し，評価を行なった．タイプ別ルート選択の有効性を調べるため，refreshedlist のみで実装した方式と referencedlist のみで実装した方式でも同様の性能評価を行なった．また，第 5.6 節で本方式が採用した方法の有効性を調べるため，recycledlist を実装していない方式の性能評価も行なった．

以降の図，表中の mark は従来方式，gen1 は refreshedlist のみで実装した方式，gen2 は recycledlist を実装していない方式，gen3 は referencedlist のみで実装した方式，gen4 は本方式を表す．各方式で何が実装されているかを表 7.1 に示す．○ は実装していること，× は実装していないことを示している．

表 7.1: 実装状況

ラベル	refreshedlist	referencedlist	recycledlist
mark	×	×	×
gen1	○	×	○
gen2	○	×	×
gen3	×	○	○
gen4	○	○	○

7.2　評価結果

ベンチマークプログラム実行時間の計測結果を表 7.2，GC の処理時間の計測結果を表 7.3，メモリ使用量の計測結果を表 7.4 に示す．図 7.1 は，mark 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

表 7.2: 実行時間

	life	occur	hanoi	parser	xml
mark	51.82	7.24	19.89	2.30	26.49
gen1	41.06	6.65	19.31	2.24	16.59
gen2	43.97	6.66	19.47	2.28	17.06
gen3	40.04	7.02	19.34	2.31	16.72
gen4	40.05	6.24	19.28	2.24	16.54

単位はすべて sec

表 7.3: GC の処理時間

	life	occur	hanoi	parser	xml
mark	14321	1428	2101	204	16023
gen1	4193	964	1310	144	5859
gen2	6862	964	1456	156	6105
gen3	2489	1209	1352	175	5846
gen4	2556	557	1322	148	5848

単位はすべて msec

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．図 7.2 は，mark のメモリ使用量を 1 として，各方式でのメモリ使用量の相対値を示したグラフである．値が小さいほどメモリ使用量が少ないことを示している．

7.3 考察

GC の性能評価は慎重に行なうべきである．GC 処理時間とメモリ使用量は GC の性能を評価する重要な要素であり，トレードオフの関係にある．GC を行なわなければ，GC 処理時間はかからないが，メモリ使用量は増加する．また，GC を頻繁に行なえば不要なオブジェクトは即座に回収され，メモリ使用量は最低限必要な量ですむが，GC が頻繁に起きるため，GC 処理時間が長くなる．このようなトレードオフが存在するため，GC 処理時間を速くし，メモリ使用量を少なくすることは困難である．また，GC 処理時間のみで GC の性能評価を行なうことはできない．

本節では，まずはじめに Ruby に世代別 GC を導入することの有効性について考察を行なう．従来方式と世代別 GC の考えを適用した 4 つの方式の GC 処理時間とメモリ使用量を比較することにより，世代別 GC 導入の有効性を示す．

つぎに，本方式の有効性について考察を行なう．世代別 GC の考えを適用した 4

表 7.4: メモリ使用量

	life	occur	hanoi	parser	xml
mark	5972	2240	2024	2384	31804
gen1	6696	2300	2128	2452	32764
gen2	6508	2300	2064	2424	32860
gen3	6884	2332	2316	2460	32920
gen4	6724	2304	2124	2448	32888

単位はすべて Kbytes

つの方式の GC 処理時間とメモリ使用量を比較することにより，本方式の有効性を示す．

7.3.1 世代別 GC 適用の有効性

Ruby に世代別 GC を適用させることは有効であった．従来方式では，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と GC の処理時間が長くなり，実行時間が長くなるという問題があった．しかし，世代別 GC を適用した 4 つの方式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しても GC の処理時間が長くなることはなく，従来方式よりも実行時間が減少した．

表 7.3 から，従来方式よりも世代別 GC の考えを適用した 4 つの方式のほうが GC 処理時間が速いことがわかる．理由として，以下の 2 つが考えられる．

- 世代別 GC では，GC のとき通常は若い世代のみを GC の対象とし，古い世代のマーキングを行なわない．従来方式では世代に関係なくマーキングを行なっている．そのため，世代別 GC のほうが従来方式に比べ GC のときのオブジェクトのマーキングが少ない．

- 世代別 GC では，GC のとき通常は若い世代の領域のみ不要オブジェクト回収の対象としている．従来方式では使用しているすべての領域を不要オブジェクト回収の対象としている．そのため世代別 GC のほうが従来方式に比べ不要オブジェクト回収時間が少ない．

また図 7.1 から，parser や hanoi に比べて life や xml のほうが GC 処理時間の減少する割合が大きいことがわかる．これはオブジェクトを多数生成するようなプログラムのほうが，世代別 GC を適用することによる効果が大きいためだと考えられる．従来方式の GC 処理時間はメモリ領域にあるオブジェクトの数に比例するが，世代別 GC の GC 処理時間は若い世代の領域にあるオブジェクトの数に比例する．したがって，若い世代の領域にあるオブジェクトの数とメモリ領域にあるオブジェクト

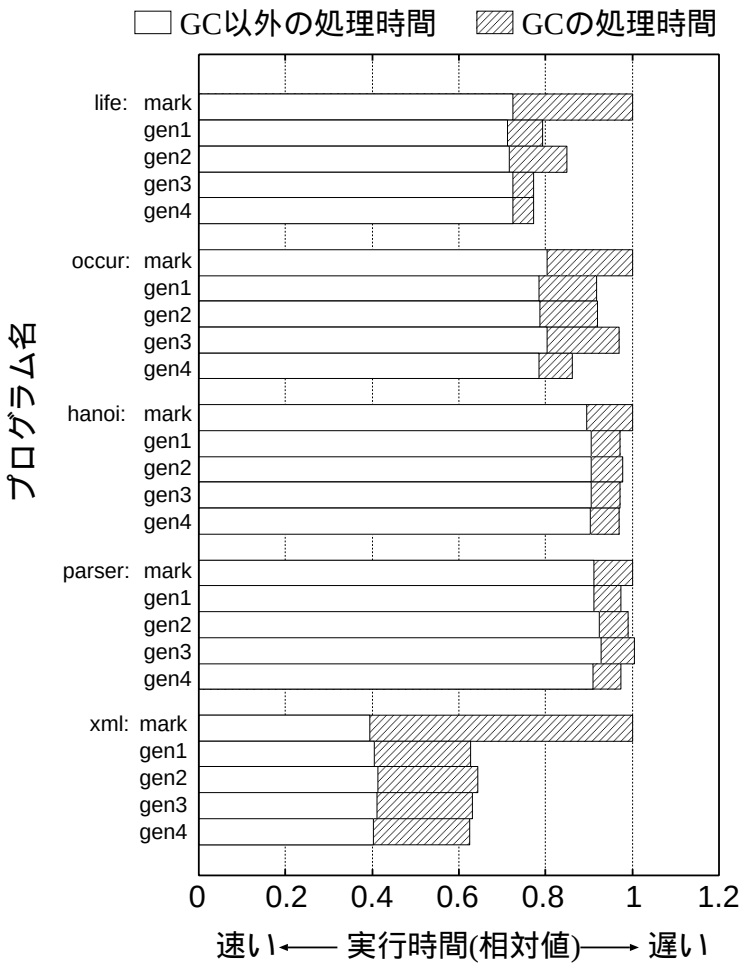

図 7.1: 実行時間の比較

の数の差が大きくなるような life，xml のほうが，世代別 GC にすることの効果があると考えられる．

表 7.4 から，従来方式よりも世代別 GC の考えを適用した 4 方式のほうがメモリ使用量が多いことがわかる．理由として，以下の 3 つが考えられる．

- 単方向リストのための領域をオブジェクト毎に追加．
- 通常の GC では古い世代で不要となったオブジェクトは回収されないため，従来方式よりも無駄な領域が生じる．
- referencedlist を管理するための領域が必要となる．

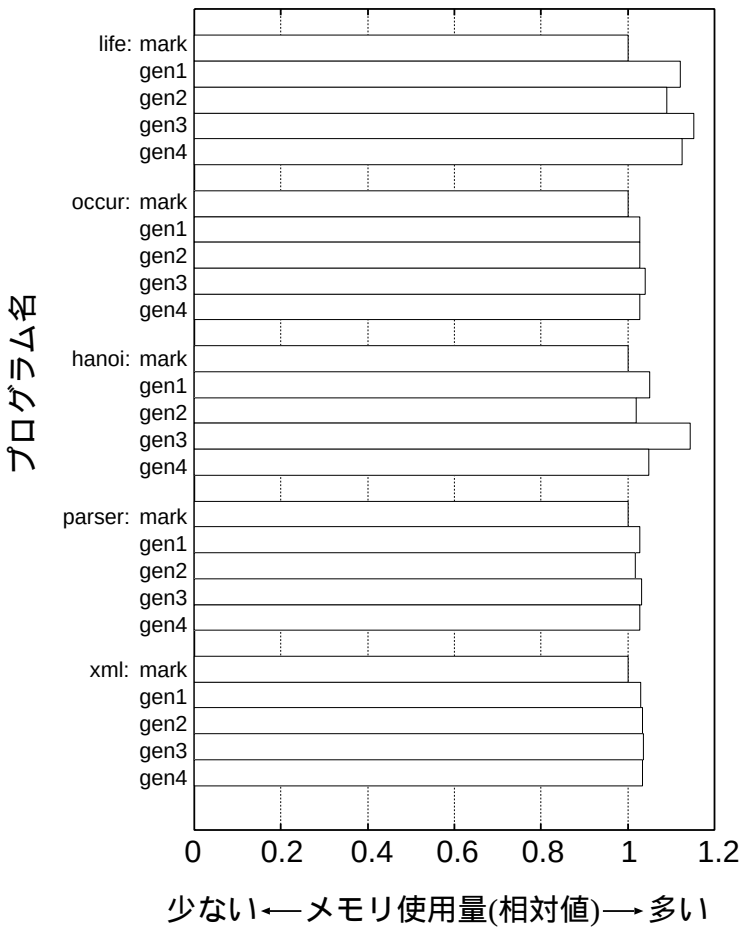

図 7.2: メモリ使用量の比較

7.3.2 タイプ別ルート選択の有効性

世代別 GC にタイプ別ルート選択を適用させることは有効であった．GC 処理時間とメモリ使用量の 2 つの計測結果から，本方式がもっとも優れていると考えられる．以降，評価を行なったベンチマークプログラムの結果を用いて，本方式の有効性について考察を行なう．hanoi，parser，xml では，タイプ別ルート選択の有効性が顕著に現れていないため，まとめて考察を行なう．

life

表 7.3 と図 7.1 から，gen3，gen4 の GC 処理時間はほぼ同じ速さとなり，gen1 の GC 処理時間は gen3，gen4 に比べて遅いことがわかる．また，表 7.4 と図 7.2 から，gen1，gen4 のメモリ使用量はほぼ同じとなり，gen3 のメモリ使用量は gen1，gen4 に比べて多いことがわかる．

life は，配列もしくはハッシュオブジェクトが多量のオブジェクトを参照するようなプログラムである．そのため，参照元の配列もしくはハッシュオブジェクトがルートとなると無駄なマーキングによって GC 処理時間が遅くなる．gen1 は参照元をルートとする方式，gen3 は参照先をルートとする方式，gen4 は配列とハッシュオブジェクトのみ参照先をルートとする方式である．そのため，life では gen1 は無駄なマーキングをするため GC 処理時間が遅くなるが，gen3 と gen4 は無駄なマーキングをしないため GC 処理時間が速くなると考えられる．

gen3 はルートへの参照を保持する領域を動的に確保している（図 6.3）．そのため，ルートが多くなるとルートへの参照を保持する領域が大きくなり，他の方式に比べてメモリ使用量が多くなると考えられる．

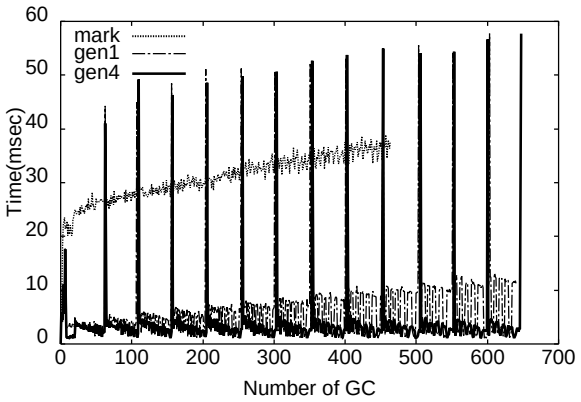

図 7.3: 1 回の GC 処理時間 (life)

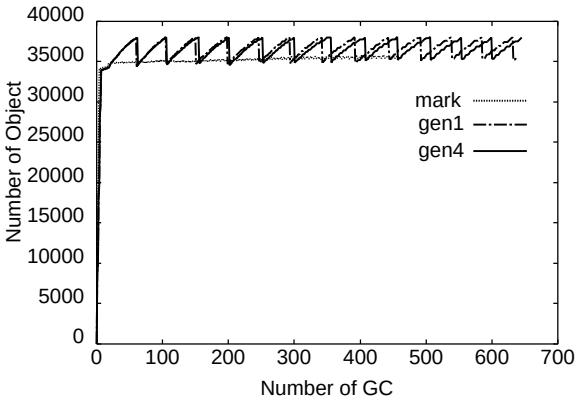

図 7.4: 生き残るオブジェクト数 (life)

図 7.3 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 7.4 は横軸が GC の回数，縦軸はそのときの GC 終了後，必要だと判断されたオブジェクトの数，すなわち，生き残るオブジェクト数を表している．gen2 は gen1 のグラフを横に伸ばしたようなグラフになり，同様の考察が得られるため，図 7.3，図 7.4 では省略した．また，gen3 は gen4 と同一のグラフになるため省略した．

図 7.3，図 7.4 から，以下のことがわかる．

- mark のグラフは GC1 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．これは，従来方式がすべての領域を GC の対象とするためである．

- gen1，gen4 のグラフでは，従来方式より 1 回の GC 処理時間が長い箇所がある．これは古い世代の GC が行なわれた箇所である．古い世代の GC は，古い世代のオブジェクトのマークを外すなどの処理があるため，GC 処理時間が長くなる．

- gen1 のグラフは GC の回数が大きくなるにつれて，gen4 に比べ，1 回の GC 処理時間が長くなっている．これは，gen1 が参照元をルートとしているため，

無駄なマーキングが起きていると考えられる．

occur

表 7.3 と図 7.1 から，gen4 の GC 処理時間が最も速く，gen1 と gen2 の GC 処理時間はほぼ同じ速さとなり，gen3 の GC 処理時間は gen1，gen2，gen4 に比べて遅いことがわかる．また，表 7.4 と図 7.2 から，gen1，gen2，gen4 のメモリ使用量はほぼ同じとなり，gen3 のメモリ使用量は gen1，gen2，gen4 に比べて多いことがわかる．

occur は，オブジェクトの参照先が頻繁に変化するようなプログラムである．そのため，参照先の若い世代のオブジェクトをルートとすると，GC のときルートとしたオブジェクトはすでに参照元の古い世代のオブジェクトから参照されておらず，実際は不要なオブジェクトになっている可能性がある．しかし，ルートとなっているため参照されていると誤認され，古い世代へ移動させられる．そうすると，オブジェクトの回収率が下がり，古い世代の GC の回数が増え，GC の処理時間が遅くなると考えられる．

gen3 のメモリ使用量が他の方式に比べて多くなるのは，life の場合と同様の理由が考えられる．

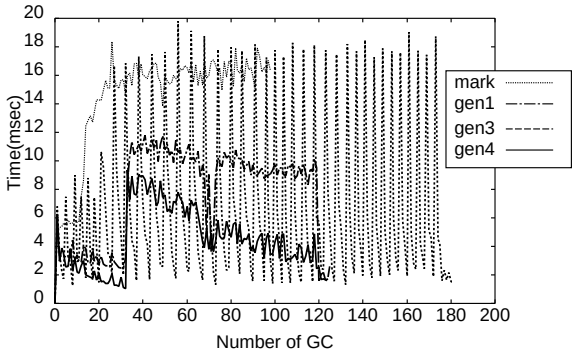

図 7.5: 1 回の GC 処理時間 (occur)

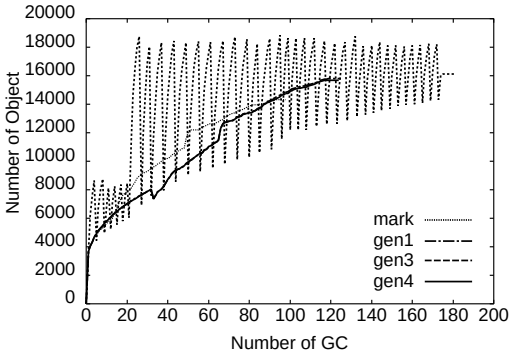

図 7.6: 生き残るオブジェクト数 (occur)

図 7.5 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 7.6 は横軸が GC の回数，縦軸はそのときの GC 終了後，生き残るオブジェクト数を表している．図 7.3，図 7.4 では，gen2 は gen1 と同様のグラフになるため省略した．

図 7.3，図 7.4 から，以下のことがわかる．

- gen3 のグラフで頻繁に古い世代の GC が起きている．これは，gen3 が参照先をルートとしてるため，GC のとき不要なオブジェクトが古い世代に移動され，若い世代で不要なオブジェクトを回収できず，古い世代の GC が頻繁に起きると考えられる．

- GC の回数が 40 回目ぐらいから，gen4 に比べて gen1 の 1 回の GC 処理時間が長い．これは，40 回目ぐらいから gen1 で無駄なマーキングが起きているためと考えられる．

hanoi，parser，xml

表 7.3 と図 7.1 から，hanoi，parser，xml では gen1，gen2，gen3，gen4 の GC 処理時間に大きな差はないことがわかる．また，表 7.4 と図 7.2 から，hanoi，parser では gen2 のメモリ使用量が gen1，gen3，gen4 に比べて少なく，hanoi，parser，xml では gen3 のメモリ使用量が gen1，gen2，gen4 に比べて多いことがわかる．

hanoi，parser，xml では，多量のオブジェクトを参照するようなオブジェクトが少なく，参照先が頻繁に変化するようなオブジェクトが少ないため，gen1，gen2，gen3，gen4 の GC 処理時間に大きな差がなかったと考えられる．また，parser は GC の回数が少ないため，GC 処理時間の差があまりなかったと考えられる．

hanoi，parser では gen2 のメモリ使用量が gen1，gen3，gen4 に比べて少ない．これは，gen2 では GC 以外のとき明確に不要なオブジェクトとわかるオブジェクトを回収し，再利用していないため，GC が頻繁に行なわれ，不要なオブジェクトの回収が即座に行なわれる．そのため，メモリ使用量が少ないと考えられる．gen3 のメモリ使用量が gen1，gen2，gen4 に比べて多くなるのは，life の場合と同様の理由が考えられる．

図 7.7，図 7.9，図 7.11 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 7.8，図 7.10，図 7.12 は横軸が GC の回数，縦軸はそのときの GC 終了後，生き残るオブジェクト数を表している．図 7.11，図 7.12 では，gen1，gen2，gen3 は gen4 と同様のグラフになるため省略した．

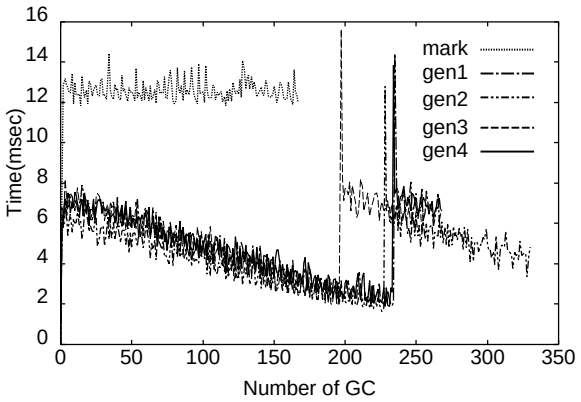

図 7.7: 1 回の GC 処理時間 (hanoi)

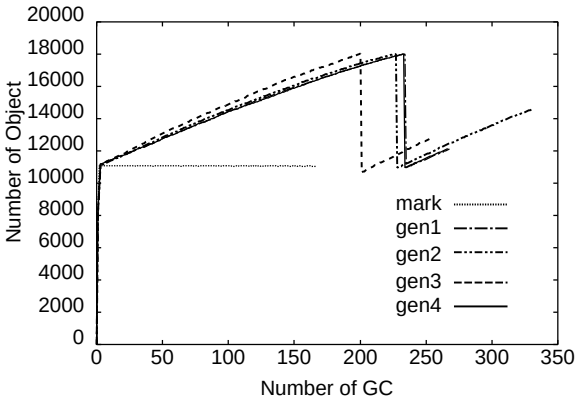

図 7.8: 生き残るオブジェクト数 (hanoi)

図 7.7，図 7.8 から，以下のことがわかる．

- gen1，gen2，gen4 は GC の回数が 240 回目ぐらいのとき古い世代の GC を行

なっている．gen3 は GC の回数が 200 回目ぐらいのとき古い世代の GC を行なっている．これは gen3 が参照元をルートとしてるため，若い世代で不要なオブジェクトを回収できず，古い世代の GC が早く行なわれたと考えられる．

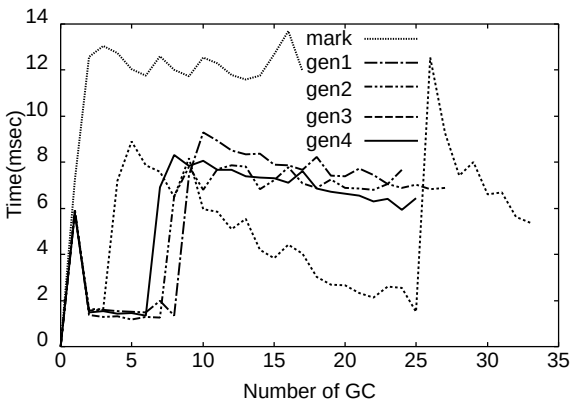

図 7.9: 1 回の GC 処理時間 (parser)

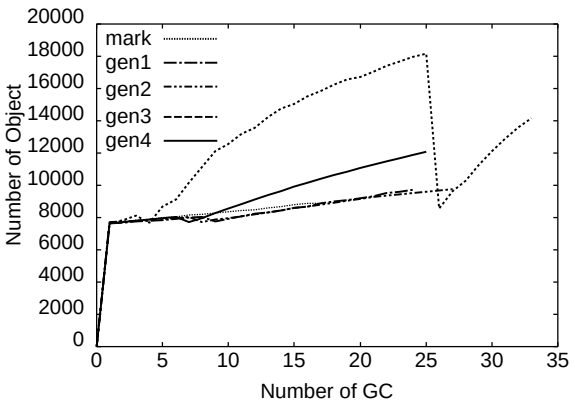

図 7.10: 生き残るオブジェクト数 (parser)

図 7.9，図 7.10 から，以下のことがわかる．

- gen3 のグラフは GC の回数が 26 回目ぐらいのときに古い世代の GC を行なっているが，gen1，gen2，gen4 は古い世代の GC を行なうことなくプログラムが終了している．これは，gen3 が参照元をルートとしてるため，若い世代で不要なオブジェクトを回収できず，古い世代の GC が行なわれたと考えられる．図 7.10 の gen3 のグラフからも，不要なオブジェクトが古い世代へ移動していることがわかる．

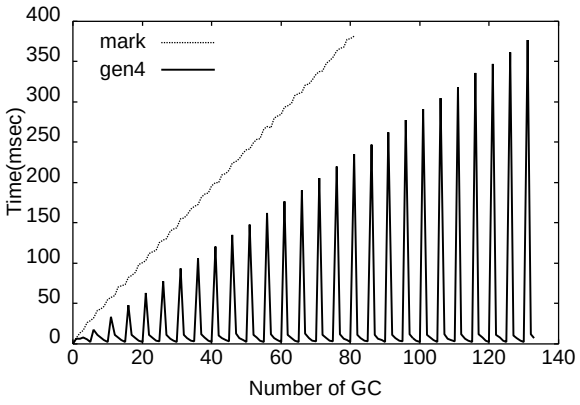

図 7.11: 1 回の GC 処理時間 (xml)

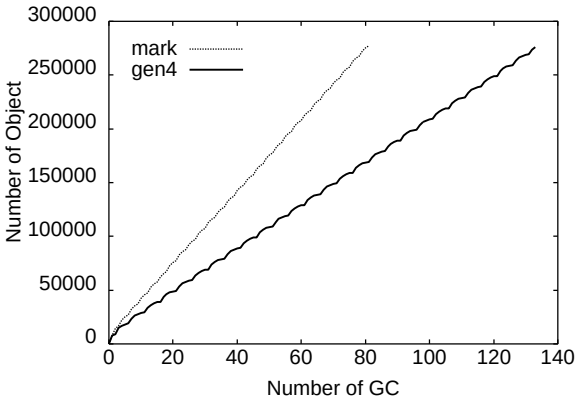

図 7.12: 生き残るオブジェクト数 (xml)

図 7.11，図 7.12 から，以下のことがわかる．

- オブジェクトの数がプログラム終了まで増え続けるプログラムである．

- gen4 のグラフは古い世代の GC が繰り返され，徐々に GC 処理時間が長くなっていることがわかる．

第8章　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大規模なプログラム開発にも用いられ，高速化の必要がある．大規模なプログラムでは，実行時間に占めるGC処理時間の割合が増えてしまう．そこで，GCの高速化が必要となる．

Treadmill GC[14]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[11]．Treadmill GCは実時間GCである．Treadmill GCはコピー法を改良したもので，GCの処理時間は生きているオブジェクトに比例する．そのため，生きているオブジェクトが多くなると，実時間性に乏しくなる．生きているオブジェクトが多くなったとしても，実時間性を確保するため，小池らはTreadmill GCに世代別GCの考えを導入している．本方式は実時間GCではないが，小池らの方式は，実時間GCである点が異なる．また，小池らの方式は世代間の移動を双方向リストの付け換えによって行なうが，本方式は単方向リストの付け換えによって行なっている．そのため，本方式のほうがオブジェクト毎の管理コストが少ない．

GNU Emacsに世代別GCを実装する研究が，小林らによって行われている[5, 6, 7]．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行なっている．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ーヘッド削減を目的としている．本方式は，世代間の移動を単方向リストの付け替えによって行なうが，小林らの方式は，オブジェクトをコピーすることで世代間の移動を行なう点が異なる．小林らの方式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，本方式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．

安松らによって，保守的なGCに世代別GCの考えを適用した方法が提案されている[13]．オブジェクトテーブルという，オブジェクトを参照するテーブルを導入し，オブジェクトを間接参照することで，領域のコンパクションを可能としている．本方式は，世代間の移動を単方向リストの付け替えによって行なうが，安松らの方式は，オブジェクトをコピーすることで世代間の移動を行なう点が異なる．安松らの方式はオブジェクトを間接参照するため，間接参照のオーバーヘッドが生じるが，

本方式はオブジェクトを直接参照するため，間接参照のオーバーヘッドはない．

第9章　結論および今後の展望

本論文では，Ruby に世代別 GC を実装する場合の方法について述べた．また，世代別 GC を実装している一般の言語に適用可能なタイプ別ルート選択を提案し，Ruby に実装した．ベンチマークプログラムを用いて計測を行なった結果，本方式は従来方式に比べ，GC 処理時間を最大 82.2%，プログラムの実行時間を最大 37.6%短縮した．また，計測結果から，本論文で提案したタイプ別ルート選択の有効性を示した．

本方式の特徴として，以下が挙げられる．

- タイプ別ルート選択を実装している．
- 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．

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:
  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，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．
- オブジェクトの数が増え続けるようなプログラムに対応:
  xml のようなオブジェクトの数が増え続けるプログラムは，本方式であっても実行時間に占める GC 処理時間の割合が多い．オブジェクトの数が増え続けることを自動的に検出し，不要になったオブジェクトを回収することをやめ，メモリ領域を拡張し続ければ，GC 処理時間の短縮ができると考える．

参考文献

[1] http://www.ruby-lang.org/

[2] John Ousterhout. Scripting: Higher level programming for the 21st century. IEEE Computer, 31(3):23-30, March 1998.

[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(7):37-42, 1997

[6] 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

[7] 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

[8] 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(6):31–36, 1997

[9] 中西 正和，田中 詠子. 特集：< ごみ集めの基礎と最近の動向 > 世代別ごみ集め. 情報処理, Vol.35, No.11, pp.1014-1019, 1994

[10] 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

[11] 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, vol.40, SIG7(PRO4), pp.1–8, 1999

[12] R. Jones, R. Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management, John Wiley and Sons Ltd(1996).

[13] 安松 一樹，栗原 聡，土居 範久. あいまいなルートを持つジェネレーショナルスキャベンジング. コンピュータソフトウェア, vol.10, No.5, pp.15–28, 1993

[14] Henry G. Baker. The Treadmill, Real-time Garbage Collection without Motion Sickness. ACM SIGPLAN Notices, vol.27, No.3, 1992