短寿命オブジェクトを対象とした静的解析 GC の提案

小室　直[†]　阿部公輝[†]
{komuro,abe}@cacao.cs.uec.ac.jp

世代別 GC はほとんどのオブジェクトは生成後まもなくごみとなる短寿命オブジェクトであるという性質を用いて効率的にごみを回収するが，短寿命オブジェクトの判別を動的に行うため処理時間の予測が難しい．本研究では「短寿命オブジェクトの多くは有効範囲が静的に判別可能なオブジェクトで占められるのではないか」との予想をたて，その確かさを検証する．そして静的解析により短寿命オブジェクトの判別・回収を効率良く行う手法を提案する．予想の検証はオブジェクト指向スクリプト言語である Ruby 処理系 MRI にて行う．Ruby は近年開発・研究用途での利用が広まっているが GC の実装は未だ不十分であり，提案手法により性能向上が見込める．

Static Detection of Short-lived Objects for Effective GC

Sunao Komuro[†] and Kouki Abe[†]
{komuro,abe}@cacao.cs.uec.ac.jp

Generational GC schemes effectively collect garbage objects utilizing the fact that most objects die young. However, the time required for the collection is unpredictable because the schemes detect the short-lived objects dynamically. In this paper, we put a hypothesis that most of the short-lived objects are those whose scopes can be statically detected, and we verify the hypothesis experimentally. We also propose a method which effectively collects the short-lived objects detected by the static analysis. The experimental verification is conducted using MRI, an interpreter of object-oriented scripting language Ruby. Ruby has been widely used for research and development purposes, but no effective implementation of GC is currently available. The proposed method is expected to improve the performance of Ruby interpreter.

1. はじめに

近年では必要とされるプログラムの巨大化・複雑化にともないプログラム開発にかかる時間が増しており，プログラム開発効率を上げる研究は重要な課題となっている．プログラム開発においてはメモリ管理の複雑さがボトルネックの一つである．特に動的に確保した領域の管理はしばしばダングリングポインタ，メモリリークなどバグの元となり開発時間を肥大化させる．そこで動的に確保した領域を自動的に開放する Garbage Collection(GC) が広く用いられる[1]．

一般に GC により必要なメモリ領域は増加する．また，GC の間他の処理を停止する．更にいつ GC が発生するか予想できないためリアルタイムシステムに組み込むのが難しい．これらの問題を解決するために様々な手法が提案されている．中でも世代別 GC[2),3)] はほとんどのオブジェクトは生成後まもなくごみとなるという仮説 (世代仮説) に基づき短寿命オブジェクトを効率的に回収できる．しかしそのままでは保守的アルゴリズム[4)] を適用できないなどコストも大きい．

本研究では「短寿命オブジェクトの多くは有効範囲が静的に判別可能なオブジェクトで占められるのではないか」との予想をたて，その確かさを検証する．そして静的解析により短寿命オブジェクトの発見・回収を行う手法を提案する．予想の検証はオブジェクト指向スクリプト言語である Ruby[5)] 処理系 MRI にて行う．Ruby は近年開発・研究用途での利用が広まっているが GC の実装は未だ不十分であり，提案手法により性能向上が見込める．

以下第 2 章では GC の概要と関連研究について，第 3 章ではスコープローカルなオブジェクトが短寿命オブジェクトの多くを占めるとの予想と，予想に基づいた静的解析 GC について述べる．第 4 章で提案手法の

† 電気通信大学 情報工学科
Department of Computer Science, The University of Electro-Communications

検証実験とその結果，第 5 章で考察を述べ，第 6 章でまとめる．

2. GC の概要と関連研究

Mark Sweep GC[6] はルートセット (ヒープ以外の静的領域，スタック，レジスタを合わせた領域) から参照を再帰的に辿り到達可能であるオブジェクトを生きたオブジェクト，それ以外をごみオブジェクトと見なす Tracing GC アルゴリズムの一つである．実装が簡単であることもあり広く使われるが，単純にそのまま実装すると性能を求める場合には不十分である．

世代別 GC は世代仮説に基づき短寿命オブジェクトで占められる新世代領域と長寿命オブジェクトで占められる旧世代領域に分けて管理する GC である．世代仮説 (Generational Hypothesis，新しく生成されたオブジェクトは古いオブジェクトより生存率が低いとの仮説) は多数の研究者により検証され，その確かさが示されている．メモリ領域が枯渇した際にはまず GC 対象を新世代領域とした Minor GC を行う．旧世代領域を含めた全体を対象とした Major GC は Minor GC により十分な領域が確保できなかった場合のみに行われる．ある程度の回数 Minor GC を生き残ったオブジェクトは旧世代へ移動される．世代別 GC ではその処理の多くを生存率の低い短寿命オブジェクトを効率的に回収する Minor GC が占め，その結果単純な GC に比べて性能向上が見込める．しかし，機構の複雑さ，旧世代から新世代への参照を記録するためのライトバリアの書き込みコスト，保守的 GC (ポインタとなりうる値を持つオブジェクトはポインタと見なすアルゴリズム) との組み合わせが困難などの問題がある．

マルチコアプロセッサの普及に伴いスレッドエスケープ解析が注目されている[7]．生成されたオブジェクトが単一のスレッドで使われる場合にはスレッドローカルのヒープから解放してよい．このヒープは他のスレッドと独立に回収可能である．こうした考えの元にいくつかの実用的なエスケープ解析の解析精度について比較検討されているが，スコープローカルなオブジェクト短寿命オブジェクトの中のどれぐらいの割合を占めるかという問題には関心が払われていない．

Ruby は，手軽なオブジェクト指向プログラミングを実現するための種々の機能を持つオブジェクト指向スクリプト言語である[5]．強力なテキスト処理，プログラミングを容易にするシンプルな文法などの特徴があり，業務・研究用途に普及してきている．しかし，現在主に利用されている Ruby 処理系 MRI の GC は未だ不十分である．

3. 提 案 手 法

世代別 GC は短寿命オブジェクトを効率的に回収し性能向上が見込める一方で，そのコストも小さくはない．本研究では短寿命オブジェクトに対して「短寿命オブジェクトはその多くがスコープローカルなオブジェクトが占めるのではないか」との予想を検証し，予想に基づいた静的解析による短寿命オブジェクトの回収手法について提案する．

3.1 短寿命オブジェクトに関する予想

アセンブリ言語や C，C++といった言語によりプログラムを記述するプログラマは変数や関数がメモリ中のどの領域に格納されるか意識する必要がある．このようなプログラムでは，関数引数や返り値，一時変数などの寿命が短かくプログラムの構文からその変数の参照されうる範囲 (スコープ) が定まっている変数は通常スタック領域に確保される．

一方古くは Lisp から始まる高級プログラミングパラダイムにおいてはプログラマは変数や関数がメモリ中のどの領域に格納されるか意識する必要がない．この場合プログラマには，全ての変数や関数はヒープに格納され，プログラム終了まで参照できるオブジェクトに見える．このように，高級プログラミングパラダイムにおいては，C などでスタック領域に一時的に保持していたスコープローカルな変数についてもヒープ領域に格納する．全てのオブジェクトがプログラム終了時までヒープ領域に存在し続けるかのように見せるために GC が必要となる．一般に GC ではスコープローカルな変数についても動的回収の対象とする．特に世代別 GC では生存率の低い短寿命オブジェクトに着目し効率的に回収する．短寿命オブジェクトであるかどうかは動的に判定する．

GC において効率良くオブジェクトを回収するためには生存率の低い短寿命オブジェクトの判別が重要である．本研究では短寿命オブジェクトの多くはスコープローカルなオブジェクトであるとの予想をたてる．スコープローカルなオブジェクトはプログラムの構造から定まり，静的に判別が可能である．この予想が正しければ生存率の低い短寿命オブジェクトの多くが容易に判別できることとなり，GC の効率の向上に役立つと考えられる．

3.2 静的解析による短寿命オブジェクトの判別と回収

スコープローカルなオブジェクトはプログラムを解析することで，実行せずとも少ないコストで発見し，生きている範囲を調べることができる．そこでプログラムの静的解析により実行前に短寿命オブジェクトを判別し判別されたオブジェクトを使用後に回収する方法を示す．

Ruby 処理系 MRI において，プログラムは Ruby 言語で書く部分と C 言語で書くライブラリ部分があるので，それぞれについてローカルオブジェクトの判別および回収について述べる．

この手法が有効であるかどうかは，静的解析により判別されるスコープローカルなオブジェクトが短寿命オブジェクト全体の多くを占めるとの予想の妥当性に依存する．予想の検証は次章で述べる．

3.2.1 Ruby プログラムにおけるローカルオブジェクト

MRI は入力した Ruby プログラムを一旦構文木として構築した後に Ruby の Virtual Machine(VM) のバイトコードにコンパイルして実行する．直列化されたバイトコードは木構造を持つ複雑な構文木に比べ単純で解析しやすいため，Ruby プログラム中に出てくるローカルオブジェクトの判別・回収などはバイトコードに対して行うのが適切であると考えられる．

バイトコードにおいてローカルオブジェクトがどのように生成されるかを図 1 のような関数 str を例として考える．このプログラムの変数 `a` ，`b` はそれぞれ `b = a + a`，`"hoge#{b}"` という演算以降は使用されない．関数 str のコンパイル結果を図 2 に示す．0021 番地の tostring 命令は，図 3 のコードで定義され，与えられたオブジェクトを文字列化して返す．tostring で生成された文字列オブジェクトも 0022 番地の concatstrings で利用された後は使われることはない．これらのスコープローカルなオブジェクトは短寿命であるにも関わらず次回 GC まで生き残り続ける．

このように Ruby バイトコードプログラムにおけるローカルオブジェクトは，図 1 の `a`，`b` のようにプログラマが書くプログラム中に明示的に出現するものと，図 2 の tostring で生成される文字列オブジェクトのようにそうでないものがあるが，いずれも Ruby バイトコードコンパイラに次のような最適化を加えることで対応できる: それらのオブジェクトを使用後に回収するコードを追加する，または，オブジェクトをスタック割り当てにより生成する．

```
1 def str
2   a = 10.0
3   b = a + a
4   return "hoge#{b}"
5 end
```

図 1　関数 str(Ruby プログラム)

```
0000 trace            8
0002 trace            1
0004 putobject        10.0
0006 setlocal         a
0008 trace            1
0010 getlocal         a
0012 getlocal         a
0014 opt_plus
0015 setlocal         b
0017 putobject        "hoge"
0019 getlocal         b
0021 tostring
0022 concatstrings    2
0024 trace            16
0026 leave
```

図 2　関数 str(Ruby バイトコード)

```
 1 /**
 2   @c put
 3   @e to_str
 4   @j to_str の結果をスタックにプッシュする
 5  */
 6 DEFINE_INSN
 7 tostring
 8 ()
 9 (VALUE val)
10 (VALUE val)
11 {
12     val = rb_obj_as_string(val);
13 }
```

図 3　VM 命令 tostring

3.2.2 C 言語ライブラリにおけるローカルオブジェクト

MRI においてライブラリの多くは C 言語により記述されている．ローカルオブジェクトの中にはこの C 言語のレベルで生成されているものも多くある．

一例として図 4 に示す Bignum(多倍長整数) オブジェクトの加算を定義する関数 `rb_big_plus` をとりあげる．このコード中に出てくる rb_int2big は，Fixnum(固定長の整数) オブジェクトを引数として Bignum オブジェクトを生成する関数である．このコードの場合，5 行目で `rb_int2big(FIX2LONG(y));` により生成されたオブジェクト `y` は，この関数 `rb_big_plus` の終了後はアクセスされないが，次回 GC が発生するまではヒープ中に生き残り続ける．

```
 1 rb_big_plus(VALUE x, VALUE y)
 2 {
 3   switch (TYPE(y)) {
 4   case T_FIXNUM:
 5     y = rb_int2big(FIX2LONG(y));
 6     /* fall through */
 7   case T_BIGNUM:
 8     return bignorm(bigadd(x, y, 1));
 9
10   case T_FLOAT:
11     return DOUBLE2NUM(rb_big2dbl(x)
12       + RFLOAT_VALUE(y));
13
14   default:
15     return rb_num_coerce_bin(x, y, '+');
16   }
17 }
```

図 4 Bignum の加算メソッド定義 rb_big_plus

表 1 Ruby ベンチマークプログラムの説明

app_factorial	階乗関数を再帰的に繰り返す
app_raise	単純な組み込み例外の発生と補足を繰り返す
so_object	オブジェクトの生成を繰り返す
so_random	浮動小数点数の演算を繰り返す
app_strconcat	式展開により文字列の連結を繰り返す
so_concatenate	文字列の連結を繰り返す
so_count_words	入力されたテキストの行数，単語数，文字数をカウントする
so_exception	ユーザ定義の例外の発生と補足を繰り返す
so_lists	配列に関する基本的な演算を繰り返す
so_matrix	多次元配列に関する基本的な演算を繰り返す
vm2_array	配列リテラルにより配列の生成を繰り返す
vm3_thread_create_join	スレッドの生成を繰り返す
hoge app_mandelbrot	マンデルブロー集合計算をする

このようなごみオブジェクトは，C のライブラリソースコードをオブジェクト使用後に回収するよう変更するか，オブジェクトをスタック割り当てにより生成するものとする．プログラマに利用されるライブラリへのこのような変更は予め言語処理系実装に組み込むことができる．

4. 検 証 実 験

ここでは短寿命オブジェクトに関する予想の検証実験について方法と結果を述べる．

```
1 def str
2   a = GC.mark_local(10.0)
3   b = GC.mark_local(a + a)
4   return "hoge#{b}"
5 end
```

図 5 関数 str の Ruby プログラムにおけるマーキング

```
0000 trace            8
0002 trace            1
0004 getinlinecache   <ic>, 11
0007 getconstant      :GC
0009 setinlinecache   4
0011 putobject        10.0
0013 send :mark_local, 1, nil, 0, <ic>
0019 setlocal         a
0021 trace            1
0023 getinlinecache   <ic>, 30
0026 getconstant      :GC
0028 setinlinecache   23
0030 getlocal         a
0032 getlocal         a
0034 opt_plus
0035 send :mark_local, 1, nil, 0, <ic>
0041 setlocal         b
0043 putobject        "hoge"
0045 getlocal         b
0047 tostring
0048 concatstrings    2
0050 trace            16
0052 leave
```

図 6 関数 str の Ruby バイトコードにおけるマーキング

4.1 方　　法

Ruby Benchmark Suite[8] のうち表 1 に示す Ruby プログラムについて，手動でローカルオブジェクトをマークしてオブジェクト全体に対する比率を調べる．実験には MRI 1.9.0-4 に以下の変更を加えたものを用いた．

- オブジェクトにローカルオブジェクトであることを示す情報を付ける．
- 全オブジェクトの生成時刻，ごみになる時刻，ローカルオブジェクト情報を記録し，表示する．

Ruby プログラムに加える変更について 3.2.1 の図 1，図 2 で示すプログラムを例として説明する．生成されるローカルオブジェクトのうち，Ruby プログラムに明示的に出現するローカル変数 a，b に対しては図 5，図 6 に示すように GC.mark_local_object(x) としてマークする．VM 命令の tostring で生成されるローカルオブジェクトに対しては図 7 に示すように tostring 命令の定義を変更してマークする．

C 言語ライブラリに加える変更は，例えば図 4 のライブラリの例ではプログラム中のローカルオブジェク

```
1 DEFINE_INSN
2 tostring
3 ()
4 (VALUE val)
5 (VALUE val)
6 {
7     val = rb_obj_as_string(val);
8     MARK_LOCAL(val);
9 }
```

図 7　VM 命令 tostring におけるマーキング

```
 1  VALUE
 2  rb_big_plus(VALUE x, VALUE y)
 3  {
 4    switch (TYPE(y)) {
 5    case T_FIXNUM:
 6      y = rb_int2big(FIX2LONG(y));
 7      MARK_LOCAL(y);
 8      /* fall through */
 9    case T_BIGNUM:
10      return bignorm(bigadd(x, y, 1));
11
12    case T_FLOAT:
13      return DOUBLE2NUM(rb_big2dbl(x)
14              + RFLOAT_VALUE(y));
15
16    default:
17      return rb_num_coerce_bin(x, y, '+');
18    }
19  }
```

図 8　C 言語ライブラリにおけるマーキングの例

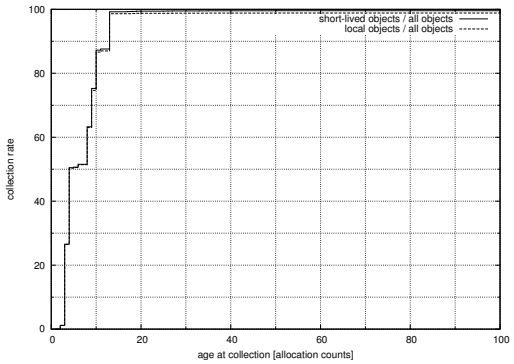

図 9　app_mandelbrot におけるオブジェクト回収率 (横軸はオブジェクトの寿命 (age at collection) を ruby のアロケーション回数 (allocatin counts) を単位としてあらわす)

ト y に対して図 8 に示すように MARK_LOCAL でマークする．

4.2 結　　果

ベンチマークプログラム app_mandelbrot と so_count_words を例として，生成されるオブジェクトの回収率を図 9，図 10 に示す．横軸はオブジェクトの寿命 (age at collection) を ruby のアロケーション回数 (allocatin counts) を単位としてあらわす．(ruby のアロケーションでは基本的に 5 ワード確保されるので，1 allocation count = 5 word と換算できる．) 縦軸はある寿命においてその寿命以下のオブジェクトのオブジェクト全体に対する割合をあらわす．図において，実線はオブジェクト全体の回収率，破線はローカルオブジェクトの回収率を示す．図からアロケーション回数約 20 でほとんど全てのオブジェクトがごみとなっていることがわかる．他のベンチマークについても同様である．

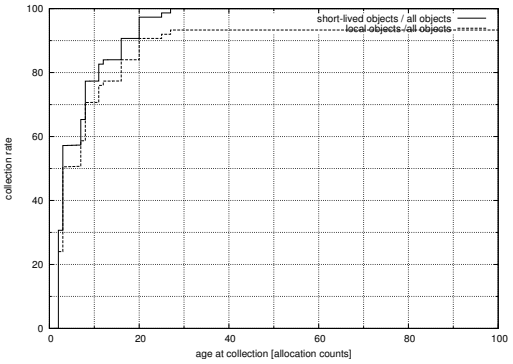

図 10　so_count_words におけるオブジェクト回収率

図 11 は表 1 のベンチマークプログラムにおける全オブジェクト中の短寿命オブジェクトの割合と全オブジェクト中のスコープローカルなオブジェクトの割合を示す．ここでは 100 アロケーション回数以下でごみとなるオブジェクトを短寿命オブジェクトとした．図からほとんどのオブジェクトが短寿命オブジェクトであることがわかる．また，少数のプログラムで例外はあるが，短寿命オブジェクトの多くをスコープローカルオブジェクトが占めていることがわかる．よって「短寿命オブジェクトはその多くがスコープローカルなオブジェクトが占めるのではないか」との予想は検証されたと言える．

5. 考　　察

実験結果から予想に反するベンチマークプログラムもいくつか存在する．その原因はローカルオブジェクトの中でマークできなかったものが存在するためである．それらマークできなかったオブジェクトに関しては以下のような理由が考えられる．

(1)　短寿命オブジェクトであり静的解析の段階で検出可能なスコープローカルオブジェクトである

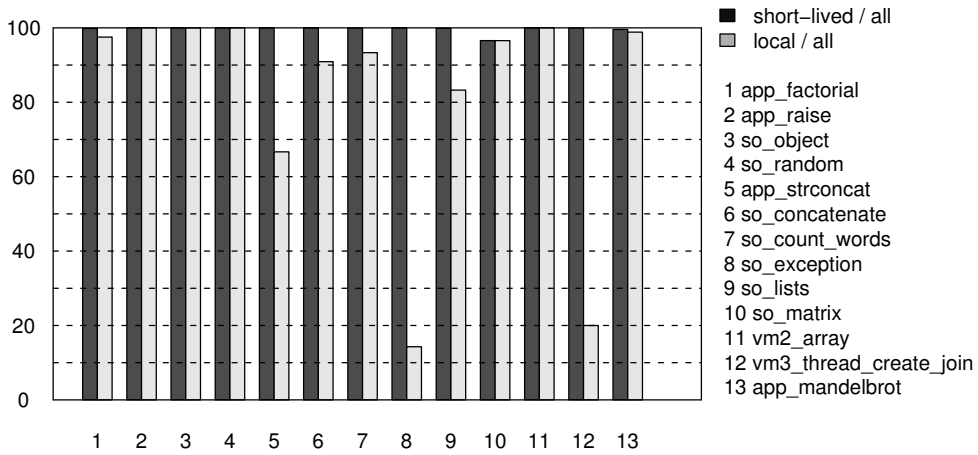

図 11 表 1 のベンチマークプログラムにおける全オブジェクト中の短寿命オブジェクトの割合と全オブジェクト中のスコープローカルなオブジェクトの割合

```
 1 $a = []
 2 class Float
 3     def +(y)
 4         $a.push(self)
 5         $a.push(y)
 6         self
 7     end
 8 end
 9 def str
10     a = 10.0
11     b = a + a
12     return "hoge#{b}"
13 end
14 p str() # "hoge10.0"
15 p $a # [10.0, 10.0]
```

図 12 メソッドの再定義の例

が，今回の実験ではマーク漏れしている．

(2) Ruby 言語において組み込みメソッドが再定義可能であることによりローカルオブジェクトであるかどうかが一意的に定まらない．

(3) 短寿命オブジェクトであるがスコープローカルではないものが存在し，入力により短寿命になる場合と長寿命になる場合があり，静的解析の段階では判別不可能である．

今回の実験の場合ローカルオブジェクトの判別とマーキングは手動で行ったため，マーク漏れは一番目の理由によるものがほとんどであり，自動化によってこのマーク漏れは減少できると考えられる．

ここでは二番目の Ruby 言語の特殊性について考察する．Ruby にはどんなクラスに対してもメソッドの再定義ができるという特徴がある．例えば図 12 に示すように組み込みクラスである浮動小数点数の加算メソッドについても再定義が可能である．関数 str は図 1 の定義と同じだが，この関数 str における変数 a はグローバル変数$a から参照されているためスコープローカルオブジェクトとはならない．このように組み込みの加算メソッドではローカルオブジェクトとなるが，再定義された加算メソッドではローカルオブジェクトとならない場合がある．よって Ruby において静的解析による自動回収を行うためには，原則として組み込みメソッドの再定義が行われていないものとし，再定義が行われた場合には静的解析が不可能であると判断する必要がある．

三番目のようなスコープローカルではない短寿命オブジェクトは静的には判別不可能である．

以上のように本手法を適用しても短寿命オブジェクトの全ては静的に判別できない．しかし従来の動的な GC 手法により回収すべきオブジェクトの量は減少する．ローカルオブジェクトを静的に解放する分空き容

量が確保され，GC の頻度が減るため性能は向上する．

6. おわりに

実験により短寿命オブジェクトに関する「短寿命オブジェクトはその多くがスコープローカルなオブジェクトが占めるのではないか」との予想の確かさが検証された．今後は C 言語ライブラリのローカルオブジェクトの更なる除去，バイトコードの解析によるローカルオブジェクトの自動判別および回収が考えられる．

参考文献

1) R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996.
2) 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 への世代別ごみ集め実装手法の改良とその評価. 情報処理学会論文誌, プログラミング, Vol. 43, No. 1, pp. 25–35, 2002.
3) M. Hirzel. Data layouts for object-oriented programs. *SIGMETRICS Perform. Eval. Rev.*, Vol. 35, No. 1, pp. 265–276, 2007.
4) H.-J. Boehm. Bounding space usage of conservative garbage collectors. In *Conference Record of the Twenty-ninth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 2002.
5) Ruby コミュニティ. オブジェクト指向スクリプト言語 Ruby. http://www.ruby-lang.org/ja/.
6) J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, Vol. 3, pp. 184–195, 1960.
7) K. Lee, X. Fang, and S. P. Midkiff. Practical escape analyses: how good are they? In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pp. 180–190, New York, NY, USA, 2007. ACM.
8) The Ruby Benchmark Suite Project. Ruby Benchmark Suite. http://groups.google.com/group/ruby-benchmark-suite.