東京大学
情報理工学系研究科 創造情報学専攻
修士論文

メモリ管理用メタプログラムを効率よく動かすための Buffered Garbage Collection

Buffered Garbage Collection helping meta-programs efficiently run for memory management

山崎 徹郎
Tetsuro Yamazaki

指導教員　千葉 滋 教授

2017 年 1 月

概要

計算途中のオブジェクトを変形させることで，メモリ消費の削減や計算性能の向上が可能な場合がある．このような変形はしばしば Garbage Collector (GC) を改造することで実現されるが，この改造は容易ではない．本研究では簡単にこのような改造ができる GC として，リフレクティブ GC を提案する．リフレクションは実行時の計算を言語処理系に反映して処理系の動作を変更するための言語機構であり，リフレクティブ GC はこの言語機構を GC に適用したものである．リフレクティブ GC を素朴に実装すると，リフレクション計算が生み出すオブジェクトを回収できず簡単にメモリ不足になってしまう．この問題を解決するため、本研究では Buffered GC を新たに開発した。Buffered GC は，メモリオーバーヘッドの原因であるリフレクティブ GC 自身が作り出すオブジェクトを，専用のナーザリであるバッファ領域を経由して通常ヒープ領域へ配置することでメモリオーバーヘッドを抑える．

Abstract

Deforming an object under calculation, can sometimes reduce memory consumption and improve computing performance. Such deformation is often implemented by customizing Garbage Collector (GC), but this customization is not easy. In this research, we propose a Reflective GC as a GC that enables easily customization. Reflection is a language mechanism for changing the behavior of a language processing system by reflecting the result of runtime calculation to that language processing system, and the reflective GC enables this language mechanism for the GC. Naive implementation of Reflective GC cannot collect objects generated by reflective calculation, and hence it often causes memory shortage. To solve this problem, we have developed the Buffered GC. The Buffered GC reduces memory overhead by placing the objects generated by the reflective GC itself, which is the cause of memory overhead, in the normal heap area via the buffer area, which is a dedicated nursery space.

目次

第 1 章

はじめに

HTTP サーバーアプリケーションでは，似たような内容のリクエストを大量に扱うため，同一内容の文字列が大量に作成される．String Deduplication[1] は GC (Garbage Collection) と連動して同一内容の文字列を共通化することでメモリの無駄を削減する手法であり，HTTP サーバーアプリケーションなどで効果を上げている．このように，アプリケーションの実行中に一部のオブジェクトを同じ意味を表す異なるオブジェクトに変形するオブジェクトエボリューション [2, 3] によって，メモリ消費の削減や計算性能の向上が実現できる．

オブジェクトエボリューションは，しばしば GC を改造することで実現される．インプレースに，メモリ上でそのオブジェクトを表現しているデータを書き換えることで実現できる変形ならば変形自体は容易だが，例えばフィールドを追加するような変形はインプレースには実現できない．インプレースではない変形を実現する方法として，メモリ上の異なる場所に変形後のオブジェクトを作成し，参照グラフ上でオブジェクトを置き換える方法が考えられる．しかし，参照グラフ上でオブジェクトを置き換えるためには，そのオブジェクトを指している全ての参照を発見，修正する必要がある．オブジェクトの移動を伴う GC アルゴリズムでは，アルゴリズムの一部として移動したオブジェクトを指す全ての参照を修正するため，オブジェクト置換のための修正も同時に行うことでオブジェクトエボリューションを簡単，高速に実現することが可能である．

しかし，GC の改造は決して容易な仕事ではない．例えば，OpenJDK7 における GC アルゴリズムの一つである Garbage First GC は 4 万行のソースコードからなる巨大なプログラムであり，改造するべき場所を見つけるだけでも一苦労であり，また GC を改造したために発生するバグを追跡，修正するためにはより広範囲の知識が要求される．そのため，オブジェクトエボリューション機構の開発は非常に大きなコストを必要とする．GC の構築を簡単化するための研究も行われている [4, 5] が，オブジェクトエボリューションなどに応用できるような柔軟さはない．本研究ではリフレクションによる GC 改造インターフェースを提供することでオブジェクトエボリューション開発にかかるコストを削減する．

リフレクティブに設計されたプログラミング言語では，実行時に行われた計算を言語処理系に反映することで言語処理系の振る舞いを変更することができる．例えば，LISP ではマクロによって実行時手続きを評価器の振る舞いに反映することが可能であり，また，Ruby では

Class#class_eval によって実行時手続きをクラスの定義に反映することが可能である．リフレクティブな GC を持つプログラミング言語ならば，実行時手続きを GC の振る舞いに反映することで，処理系の実装に関する知識がなかったとしてもオブジェクトエボリューションを開発することが可能である．

本研究ではリフレクティブ GC を利用したオブジェクトエボリューション実現の第一歩として，GC によるオブジェクトの移動に実行時手続きを反映させ，移動の途中でオブジェクトを差し替える機構の実現を考える．リフレクティブ GC では GC の途中でリフレクトされた実行時手続き (以下メタプログラム) が起動されるため，GC の途中でメモリアロケーションや GC を行う．しかし，GC の途中で作られたオブジェクトは普通，処理系内部のデータとして実行時オブジェクトとは異なる方法で管理されるため，これらのオブジェクトを実行時オブジェクトと同じセマンティクスの下で管理できる GC アルゴリズムがどのようなものなのか，知られていない．

一つのナイーブな方法として，メタプログラムを実行するために十分なヒープを残した状態で GC を開始することにし，メタプログラムが消費したメモリは次回の GC まで管理しないという方針も考えられるが，この方針にはヒープの使用効率が悪いという問題があると予測される．近年の高速なメモリアロケーションと GC を背景に大量のオブジェクトを作成，廃棄する言語処理系では関数呼出し時の引数リストや，関数内のローカル環境も一つのオブジェクトとして作成される場合があり，メタプログラムを起動するだけでも一定量のヒープを消費する．そのような言語処理系でオブジェクト移動時リフレクションを実現した場合，オブジェクトの移動はヒープ中に存在するオブジェクトの数だけ行われるため，メタプログラムによるヒープ消費は元のヒープのサイズと同等か，それ以上の可能性もあると予測される．

そこで本研究では，メタプログラムが生成するオブジェクトを一旦バッファに配置し，一定期間を生き延びたオブジェクトだけを通常のヒープに合流させる Buffered GC を開発し，メタプログラムによるヒープ消費の軽減を試みた．また Buffered GC の性能を測定する実験を行い，前述のナイーブな方法と比べてメモリ性能が向上していることを確認した．

以下，第 2 章では研究の背景にある諸概念について紹介し，リフレクティブ GC がはらむ問題を確認する．第 3 章では実験のために開発した言語処理系の実装と Buffered GC の詳細を示す．第 4 章では Buffered GC のメモリ性能を計測する実験の結果と考察を示す．そして，第 5 章で本論文のまとめと今後の課題について述べる．

第 2 章

リフレクションと GC

本研究はプログラム中から自身を管理する GC を改造することが可能なリフレクティブ GC の実現を目指している．GC のリフレクションは開発されていないが，GC 以外では様々なリフレクションが知られている．これらのリフレクションについて，2.1 節でいくつかの例を通して紹介する．2.2 節では GC について説明する．リフレクティブ GC のアプリケーションとして，同一内容文字列を自動的に同一オブジェクト化する String Deduplication が考えられる．処理系を改造せず，プログラム変換によって String Deduplication を実現する実験を行ったので，2.3 節ではこの実験について説明する．処理系を改造するならば，GC を改造することで String Deduplication のようなオブジェクトエボリューション機構を実現することができるが，GC の改造は容易ではない．2.4 節ではリフレクションによって GC を改造する例を見るとともに，リフレクティブ GC を実現する上での課題に触れる．

リフレクティブなプログラミング言語では，実行時の手続きを言語処理系に反映することができる [6]．そのような言語機能をリフレクションと呼ぶ．リフレクティブに設計されたプログラミング言語は言語機能をユーザー定義で拡張できるので，小規模な言語仕様で表現力の高いプログラミング言語が実現できる．また，リフレクションを用いた言語拡張はライブラリとして扱うことができるので，ユーザーは使いたい拡張を選択して導入することができる．例えば 3-LISP[7] では，3-LISP 上の手続きを評価器の一部に反映させ，特定の構文木に関する解釈を変更することができる．また，Smalltalk ではクラスを表現するオブジェクトを普通のオブジェクトであるかのように操作し，メソッドを追加したり削除したりすることができる．リフレクションは計算時間にオーバーヘッドを発生させるため，普通リフレクティブな設計は言語処理系の一部のみで採用する．

メモリ上の使い終わった領域を自動的に解放する言語機能を GC (Garbage Collection) と呼び，これを実装したものを GC (Garbage Collector) と呼ぶ．これらは両者ともに GC と略されるが、言語機能そのものを指す場合は Garbage Collection，言語機能の実装を指す場合は Garbage Collector であるとみなす．リフレクティブな GC の設計法は明らかではない．リフレクトされる手続きには実行時オブジェクトへの参照が含まれるため，リフレクティブ GC は処理系を構成するオブジェクトと計算上のオブジェクトを同時に管理する必要がある．しかしリフレクトされた手続きは GC の一部として，GC の最中に実行されるため，GC の最中に

```
1 (define throw (lambda-reflect [[arg] env cont]
2   (eval arg env id)))
3 (define catch (lambda-reflect [[arg] env cont]
4   (cont (eval arg env id))))
```

図 2.1. lambda-reflect による大域脱出の定義

ヒープを消費し，GC しなければならない状況が発生する．これは通常の言語処理系では起きない現象であり，このような状況下で正しく振る舞う GC アルゴリズムは知られていない．

ナイーブな対策として，リフレクトされたプログラムを実行するのに十分なヒープを残した状態で GC を開始し，GC の最中にヒープ不足を発生させない方法が考えられる．この方法は巨大なヒープを必要とし，また，ヒープがリフレクトされた手続きを実行するのに十分な大きさか判断できない問題もある．

2.1　リフレクションの例

リフレクティブなプログラミング言語では実行時の手続きを言語処理系に反映することができる．そのような言語機能をリフレクションと呼ぶ．リフレクティブなプログラミング言語では言語機能の拡張をライブラリとして定義することができる．そのため，LISP や Smalltalk といったリフレクティブな言語は言語仕様が小規模であるにもかかわらず，多様な DSL が実現可能な表現力の高い言語となっている．リフレクティブなプログラミング言語を設計するためには，リフレクティブにしたい部分をその言語自身によって実装することが有効である．本節ではいくつかのリフレクションを紹介する．

2.1.1　3-LISP におけるリフレクション

3-LISP における procedual reflection は最も初期のリフレクションの一つである．3-LISP の評価器は 評価される式，評価時の環境，評価後の継続 の三つ組を受け取ると，式の評価結果で継続を呼び出し，その結果を返すように設計されている．lambda-reflect によって作られた評価器と同じインターフェースのラムダは，その評価時に評価器の代わりにラムダ自身が使用される．

この機能を利用すると大域脱出のような機構も実現できる．lambda-reflect を利用した大域脱出の例を図 2.1 に示す．id は引数をそのまま返す恒等関数である．throw は継続を無視し，引数を評価した結果を返す．そのため，throw 以降の計算は無視され，throw によってプログラムが終了したかのように振る舞う．catch は引数を評価する際，継続として継続が存在しない事を表す id を渡し，その結果で自身の継続を呼び出す．そのため，throw に無視できる継続は catch が渡した id までであり，throw の引数は catch 継続に渡される．

図 2.2 に throw/catch の使用例を，図 2.3 にその出力を示す．この例では，catch が受け取る継続は引数を出力し，その後 ”after catch” を出力するような関数であるが，catch は引数

```
1 (begin
2   (print "before␣catch")
3   (print (catch (begin
4     (print "before␣throw")
5     (throw "thrown")
6     (print "after␣throw"))))
7   (print "after␣catch"))
```

図 2.2. lambda-reflect を利用した大域脱出の例

```
1 "before␣catch"
2 "before␣throw"
3 "thrown"
4 "after␣catch"
```

図 2.3. lambda-reflect を利用した大域脱出の例の出力

を評価する時，自身の継続ではなく，id を継続として渡す．そのため，throw が受け取る継続は ”after throw” の出力のみであり，”after catch” の出力を含まないため，throw によって無視される継続は ”after throw” の出力のみとなる．

2.1.2 Smalltalk におけるリフレクション

Smalltalk は様々なリフレクションをサポートする言語であり，クラス定義にもリフレクションを使用する．図 2.4 に Smalltalk における Pair クラスの定義を示す．Pair はフィールドとして first と second を持ち，それぞれ getFirst，getSecond メッセージで取り出し，setFirst，すことができるようなクラスである．一行目は Object クラスオブジェクトに subclass メッセージを送ることで Object を継承した Pair クラスを新たに定義している．二行目では Pair クラスオブジェクトに instanceVariableNames メッセージを送ることで Pair クラスのフィールドが first と second の二つであると定義している．

また setFirst，setSecond メッセージで値を更新することもできる．Smalltalk では ^は return を，”...”はコメントを表す．

2.1.3 Ruby におけるリフレクション

Ruby は Smalltalk の流れを汲むプログラミング言語の一つであり，Smalltalk のように豊富なリフレクションを備えている．図**??**に Ruby における Pair クラスの定義を示す．def initialize はコンストラクタの定義であり，引数に受け取った first と second をインスタンス変数に格納する．また，attr_accessor を利用して first と second の getter と setter を作成する．Ruby の class 文は定義されるクラス自身をレシーバとして実行される実行時手続きであり，例えば attr_accessor は Pair クラスオブジェクトのプライベートメソッドを呼び出している．attr_accessor は特別な文ではなく Class クラスが持つ private メソッドであるため，

```
1  Object subclass: #Pair.
2  Pair instanceVariableNames: 'first second'.
3  Pair extend [
4    getFirst  [ ^ first.  ]
5    getSecond [ ^ second. ]
6    setFirst:  value [ first  := value. ]
7    setSecond: value [ second := value. ]
8  ].
9
10 | p |
11 p := Pair new.
12 p setFirst:  'first'.
13 p setSecond: 'second'.
14 (p getFirst)  printNl.  "=> 'first'"
15 (p getSecond) printNl.  "=> 'second'"
```

図 2.4. Smalltalk におけるリフレクションの例

```
1  class Pair
2    def initialize( first, second )
3      @first = first
4      @second = second
5    end
6    attr_accessor :first, :second
7  end
```

図 2.5. Ruby における Pair クラスの定義

send を使用した方法でも呼び出すことができる．図 2.7 に send を使用して Pair クラスの attr_accessor を呼び出し，third フィールドの getter/setter を作成する例を示す．

このようなメソッドが定義できるのは，Ruby のクラスがリフレクティブに設計されているおかげである．attr_accessor は特別な文ではなくごく普通のメソッドであるため，似たようなメソッドをユーザーが追加することもできる．図 2.8 に，フィールドが正の値だった場合だけ Ruby では，全てのクラスは Class クラスのインスタンスである．そのため，Class クラスをオープンクラスしてメソッドを追加することで，クラスオブジェクトに対して呼び出すことができるメソッドを定義できる．attr_inspect_getter メソッドは引数に受け取った名前の，同名のインスタンス変数を inspect した結果を返す getter のようなメソッドを定義する．メソッド定義には通常の def 文ではなく，define_method メソッドを使用する．通常メソッド名というものは処理系内部のものであり，実行時のものではないので，実行時オブジェクトが表す名前のメソッドを定義するためにはリフレクティブな操作を呼ぶ必要があるため，通常の def 文を使うことができないためである．

```
1 pair = Pair.new( 'first', 'second' )
2 p pair.first   #=> "first"
3 p pair.second  #=> "second"
```

図 2.6. Ruby における Pair クラスの使用例

```
1 Pair.send :attr_accessor, :third
2 p pair.third  #=> nil
3 pair.third = 'third'
4 p pair.third  #=> "third"
```

図 2.7. send を使用した attr_accessor の呼び出し

```
1 class Class
2   def attr_relu_getter( attr )
3     define_method attr do
4       r = instance_variable_get( "@#{ attr }" )
5       if r > 0 then r else 0 end
6     end
7   end
8 end
```

図 2.8. attr_relu_getter の定義

```
1 class A
2   attr_relu_getter :value
3   def initialize( value )
4     @value = value
5   end
6 end
7
8 a1 = A.new( 20 )
9 a1.value  #=> 20
10 a2 = A.new( -100 )
11 a2.value  #=> 0
```

図 2.9. attr_relu_getter の使用例

2.2 ガベージコレクション (GC)

使い終わったオブジェクトを自動的に回収する言語機能を GC (Garbage Collection) と言う．また，言語機能として GC を持つプログラミング言語を managed であると言う．使い終わった不要なオブジェクトを死亡したオブジェクトと呼び，それ以外のオブジェクトを生存しているオブジェクトと呼ぶ．あるオブジェクトが生存しているかどうかはルートと呼ばれるオ

ブジェクトから参照を辿って到達できるかどうかによって判断する．managed な言語では普通，ルートから到達不可能なオブジェクトはユーザープログラムから参照することができないため，回収してしまってもユーザープログラムに影響を与えないためである．

本節では GC アルゴリズムの中でも広く知られている，基本的な 5 つのアルゴリズムを紹介する．近年のより複雑な GC もこれらのアルゴリズムを組み合わせたり，並列化したりしたものであると考えられるものが少なくない．

2.2.1　マーク&スイープ

マーク&スイープは最も単純な GC アルゴリズムの一つである．マーク&スイープのアルゴリズムはマークフェイズとスイープフェイズからなる．マークフェイズでは，ルートオブジェクト集合 (以下ルートセット) から探索を行い，到達可能だったオブジェクトの生存フラグをセットする．スイープフェイズではヒープ中のオブジェクトを走査し，生存フラグが立っていないオブジェクトに占有されていたメモリを解放する．

生存フラグはオブジェクトヘッダに書き込む実装が簡単である．ビットマップテーブルと呼ばれる，ヒープ上のアドレスと生存フラグを対応付けるテーブルを作成する方法も知られている．

コンパクションを行わないため，マーク&スイープアルゴリズムで管理されるヒープは使用中の領域と未使用の領域が混在することになる．そのため，マーク&スイープアルゴリズムで管理されるヒープはフリーリストなどの方法でどの領域が使用中であるかを管理する必要がある．フリーリストによって管理されるヒープでは，メモリアロケーションのたびにフリーリストの探索が行われるため，メモリアロケーションによる計算時間へのオーバーヘッドが大きい．

2.2.2　マーク&コンパクション

マーク&コンパクション [8] はコンパクションを行う最も単純な GC アルゴリズムの一つである．マーク&コンパクションアルゴリズムはマークフェイズ，参照修正フェイズ，コンパクションフェイズの 3 フェイズからなる．マーク&コンパクションのマークフェイズでは，マーク&スイープのマークフェイズと同様に，生存しているオブジェクトを探索し，生存フラグをセットする．マーク&コンパクションのコンパクションフェイズでは，生存しているオブジェクトがヒープの先頭から順に並ぶよう，配置をずらしていく．しかし，単純にずらしてしまうとオブジェクト中に保存されている参照が壊れてしまう．マーク&コンパクションアルゴリズムでは，コンパクションフェイズの前の参照修正フェイズで，各オブジェクトの移動先アドレスを計算し，参照を修正することでこの問題を回避する．計算した各オブジェクトの移動先は forward ポインタと呼ばれ，オブジェクトヘッダに書き込む実装が有名である．マーク&コンパクションで管理されるヒープでは free ポインタと呼ばれる，使用中 (かも知れない) 領域と未使用領域の境界以降は全て空き領域であることが保証される．そのため，メモリアロケー

ションは free ポインタの移動のみで実現され，高速である．マーク&コンパクションのデメリットは，GC による計算時間へのオーバーヘッドが大きい点である．

2.2.3 コピー GC

コピー GC[9, 10] は全ての生存しているオブジェクトをコピーすることで GC とコンパクションを同時に行う GC アルゴリズムである．コピー GC はヒープを new 領域と old 領域，二つの領域に分割して管理する．コピー GC に管理されるヒープではメモリアロケーションは old 領域に行い，new 領域は空の状態を維持する．コピー GC が始まると，マーク&スイープアルゴリズムにおけるマークフェイズのようにルートセットから探索を行うが，到達可能なオブジェクトを発見した時に生存フラグをセットせず，代わりに発見したオブジェクトを new 領域へとコピーする．死亡したオブジェクトはコピーされず，コピーされたオブジェクトは new 領域の先頭から順に並ぶため，コピーの後の new 領域は old 領域の死亡オブジェクトを回収し，コンパクションも行ったような状態になっている．コピーの後に new 領域と old 領域の役割を入れ替えることで，オブジェクトは old 領域に存在し，new 領域が空であるというコピー GC の不変条件が守られる．ヒープ全体を走査するマーク&スイープやマーク&コンパクションとは異なり，コピー GC の計算量は死亡したオブジェクトの数に影響されない．そのため，特にオブジェクトの生成/廃棄の頻度が高いアプリケーションではコピー GC が計算性能的に優れていると考えられる．コピー GC はコンパクションが行われ，かつマーク&コンパクションより高速であるが，その代わり最大でもヒープの半分しか使用することができない．

2.2.4 参照カウント法

参照カウント法 [11] は GC が始まってから各オブジェクトの生死を判定する他の GC アルゴリズムとは異なり，オブジェクトが死亡した瞬間を検出し，メモリを解放する．参照カウント法のアイデアはそれぞれのオブジェクトに，自分を指すような参照がいくつあるのかを記憶させ，これが 0 になった瞬間に解放する，というものである．参照をカウントするカウンタはオブジェクトヘッダに配置する．新しい参照が作られる，または参照が廃棄されるたびに参照されるオブジェクトのカウンタを操作することで，実際の参照数とカウンタの値の整合性を保つ．

参照カウント法は循環ゴミを回収できないという問題が知られている．循環ゴミとは，外部からの参照はないが，内部で循環的に参照が存在するオブジェクト群である．循環ゴミのそれぞれのオブジェクトの参照カウントは，循環ゴミ内部の参照が存在するために回収できないが，しかし循環ゴミは外部からの参照がないため，ユーザープログラムからアクセスできず，死亡したオブジェクトであると考えられる．この問題を回避するため，参照カウント法で管理されるヒープはしばしば他の GC アルゴリズムと併用することで循環ゴミを回収する．

2.2.5　世代別 GC

多くのオブジェクトは非常に短い期間で死亡し，また，グローバル変数のように非常に長い寿命を持つオブジェクトもまた多いという．この性質を利用し，若いオブジェクトと長命なオブジェクトを異なる戦略で管理する GC アルゴリズムを世代別 GC[12] と言う．世代別 GC は，ヒープを young 領域と old 領域に分けて管理する．メモリアロケーションは young 領域に行い，young 領域はコピー GC などのアルゴリズムで管理される．この GC によって短命なオブジェクトを回収することができる．young 領域で規定回数の GC を生き延びたオブジェクトは長命であると判断し，old 領域に移される．また，世代別 GC では old 領域で死亡したオブジェクトを回収するために，時折 old 領域も含めてすべての領域を GC する．世代別 GC は長命なオブジェクトを対象外とすることで，生存しているオブジェクトの数が性能に直結するコピー GC の性能を向上したものと考えることができる．

世代別 GC は old 領域を参照することなく young 領域を GC するために，old 領域から young 領域を指す参照を全て記憶する．この参照の集合はリメンバードセットと呼ばれる．リメンバードセットを正しく保つためには参照を書き換えるたびに新しい参照が old 領域から young 領域を指すものかどうかを調べる必要がある．このようにオブジェクトへの書き込みをフックする仕組みを write バリアと言う．世代別 GC はリメンバードセットをメンテナンスするための追加のオーバーヘッドを発生させるため，多くの場合で高速に GC できるが，そうでないアプリケーションも存在する．

世代別 GC には nursery 領域と呼ばれる，作られた直後のオブジェクトだけを配置する領域を持つ実装もある．そのような世代別 GC ではメモリアロケーションは nursery 領域に行われ，nursery 領域のオブジェクトは初回の GC で young 領域にコピーされる．

2.3　プログラム変換によるオブジェクトエボリューション機構

本研究の第一歩として，プログラム変換によるオブジェクトエボリューションの実現に取り組んだので，これを紹介する．言語処理系を改造せずにオブジェクトエボリューションを実現する方法としてプログラム変換を利用する方法が考えられる．プログラムのソースコード，あるいはコンパイル後のバイトコードの書き換えによってプログラムに機能を追加する技術をプログラム変換と呼ぶ．プログラム変換によってフックをプログラムに挿入し，このフックを利用してオブジェクトを実行時に変形することでオブジェクトエボリューションを実現することが可能である．このプログラム変換によって String Deduplication 機構を実現し，メモリ削減性能と実行時間を計測する実験を行った．この機構は同一文字列だけでなく，包含関係にある文字列も共通化を行う．この実験によって，プログラム変換によるオブジェクトエボリューション機構の性能向上は難しいということが明らかになった．

膨大な量のソースコードから類似したコード片を見つける大規模コードクローン検出はソフトウェアリポジトリマイニングにおける重要な研究テーマの一つである．このようなコードク

ローン検出の中でも Type 3 と呼ばれる種類のコードクローンの検出は，しばしば大量の計算資源を必要とする．コードクローンは，主にソースコードのコピー＆ペーストにより作られるが，ペーストされたコード片に多少の文が挿入されたり，一部の文が削除または修正される場合，元のコード片と完全には一致しなくなる．このようなコードクローンは Type 3 と分類されるが，それを検出するためにはソースコードを抽象構文木に変換して，制御フローやデータフローのグラフも作るなどし，抽象構文木の類似する部分木を探すことになる．これは計算時間やメモリを大量に消費する．

実際に Type 3 コードクローン検出器である Scorpio [13] を用いた我々の予備実験では，入力サイズである解析対象のプログラムの大きさに対して約 200 倍のメモリを消費する場合があった．我々は大規模コードクローン検出のために，スーパーコンピュータを用いて大規模並列処理をおこなうことを計画しているが，実現のためにはクローン検出器が必要とするメモリ量を減らすことが必要である．大規模並列計算機では，ノード計算機あたりに多数の計算コアを搭載するので，計算コアあたりのメモリ量が相対的に少なくなるからである．例えば富士通 FX10 スーパーコンピュータの場合，ノード計算機あたり 16 コアかつ 32 GB メモリを搭載するので，計算コアあたり 2 GB しかメモリがない．

本章は，我々が Scorpio の消費メモリ量の削減に向けて検討している Java の文字列オブジェクトの実装の工夫，およびそのためにおこなった実験の結果を報告する．Scorpio の動作を調査した結果，Scorpio は実行中，解析対象のプログラムの無数の断片を文字列として保持しており，消費メモリの実に約半分が文字列オブジェクトによって占められていたからである．2 つの文字列オブジェクトがあり，それらが表す文字列の一方が他方を包含している場合，2 つの文字列オブジェクトは内部情報を共有して消費メモリを減らすことができる．この手法をとる場合，実行中に包含関係にある文字列オブジェクトを多数発見できれば消費メモリは減少するが，多く発見しようとするとそれだけ実行速度が低下する．我々はいくつかの設定の下，消費メモリと実行速度の測定をおこない，実行速度の低下をおさえつつ消費メモリを減少させる方法を検討した．

	ファイル数	合計サイズ
Compress	253	1.9MB
Collections	497	3.9MB
Validator	128	1.1MB

表 2.1. 入力データセット

	Compress		Collections		Validator	
共有範囲	実行時間	最大メモリ	実行時間	最大メモリ	実行時間	最大メモリ
(a) 共有なし	40sec	1.55GB	125sec	1.64GB	12sec	1.55GB
(b) 全フィールド	9,090sec	239MB	27,500sec	909MB	1,830sec	322MB
(c) 全 private フィールド	316sec	926MB	695sec	1.17GB	68sec	1.54GB
(d) 特定のフィールド	306sec	1.13GB	692sec	1.20GB	62sec	1.55GB

表 2.2. 実験結果 (最大メモリはメモリ消費が最大の時に確保したヒープのサイズ)

2.3.1　文字列オブジェクトの実装法

Java 言語の文字列オブジェクトの実装方法は，文字列処理が多いプログラムの場合，そのメモリ消費量を決める重要な要素である．メモリ消費を少しでも減らすため，Java 6 までは例えば substring メソッドにより，ある文字列から部分文字列を切り出した場合，部分文字列のオブジェクトは元の文字列のオブジェクトと内部の char 配列を共有していた．この配列には文字列の各文字の文字コードが格納される．この実装は Java 7 Update 6 からは変更され，部分文字列のオブジェクトは独立した char 配列を内部に持つようになり，元の文字列のオブジェクトと配列を共有することはなくなった．これは，長い文字列からごく短い文字列を部分文字列として切り出す場合，長い文字列全体を格納する char 配列が共有されるのを避けるためである．さもないと長い文字列がゴミになった後も短い文字列が生きている間はその char 配列はゴミとして回収されず無駄である．

我々は，substring メソッドで切り出した部分文字列のオブジェクトだけでなく，任意の文字列オブジェクトが新しく生成される度に他の文字列オブジェクトとの間に文字列の共通部分がないか調べ，ある場合には char 配列を共有させる手法を検討している．これによりメモリ消費量は抑えられるが，共通部分を調べるために実行時オーバーヘッドを伴う．そこでプログラムの限られた範囲で作られる文字列オブジェクトの間でだけ，文字列の共通分がないか調べる手法を考えた．この手法は，共通部分がある文字列オブジェクトはプログラムの狭い範囲で主に生成されるという仮説に基づいている．

2.3.2　実験

我々は Scorpio を用い，文字列の共通部分がないか調べるオブジェクトが生成される範囲を色々に変えて，消費メモリ量と実行速度を測定した．解析対象のプログラムとして Apache Commons プロジェクトから大きさの異なるサブプロジェクトを選んで用いた (表 2.1)．Scorpio は Intel Core i7-4770S (3.10GHz)，Ubuntu 14.04LTS，OpenJDK 7 で実行した．

実験では，生成時に他と共通部分がないか調べる特別な文字列クラスを実装し，プログラム中に現れる String クラスを置き換えた．置き換えのためにプログラム変換器を作成し，機械的に置き換えた．元の String クラスと新しい文字列クラスは互換性がないので，必要に応じて両者間の変換を実行するコードを挿入した．置き換えの範囲は次の通りである．(a) まったく置換せずに全て元の String クラスを利用，(b) 全てのクラスの全てのフィールドを新しい文字列クラスに置換，(c) 全てのクラスの private フィールドを新しい文字列クラスに置換，(d) 事前の分析で選んだ特定のフィールドだけを新しい文字列クラスに置換．なお新しい文字列クラスは，同じクラスの全てのオブジェクトとの間で文字列の共通部分がないかを探す．全てのオブジェクトとの間で探すのではなく，一部の限られたオブジェクトとの間でだけ探して探索時間を短くするような工夫はおこなっていない．

実験結果を表 2.2 に示す．実行時間と最大消費メモリの間にはトレードオフがあることが見

て取れる．確かに全てのフィールドを新しい文字列クラスに置換する (b) と，消費メモリ量を大幅に削減できた．一方で実験した方針 (c)，(d) では，新しい文字列クラスに置換する範囲を狭めると，実行時間は改善するものの，消費メモリ量が増大した．実行速度を改善しつつ (b) に比べて消費メモリ量を増やさない方針は見つけられなかった．

2.4 リフレクションによる GC の改造

処理系を改造するならば，GC を改造することでオブジェクトエボリューションを実現することができる．しかし，GC の改造は決して容易な仕事ではない．例えば，OpenJDK7 における GC アルゴリズムの一つである Garbage First GC は 4 万行のソースコードからなる巨大なプログラムであり，改造するべき場所を見つけるだけでも一苦労であり，また GC を改造したために発生するバグを追跡，修正するためにはより広範囲の知識が要求される．そのためオブジェクトエボリューション機構を新たに開発しようとすると，非常に大きなコストが必要となる．アプリケーションごとに適切なオブジェクトエボリューション機構を導入できるよう，簡単に GC を改造できる機構が求められる．

GC がオブジェクトを移動させるたびに呼び出されるコールバック関数が登録できるならば，オブジェクトエボリューションを簡単に実現することができる．例として，図 2.10 にそのような機構を用いて定義した String Deduplication プログラムを示す．この例では，最初の 5 行で引数の文字列と同一内容の代表オブジェクトを返す intern 関数を定義している．次に，コールバック関数として登録する my-callback を定義している．register-gc-callback で my-callback を登録すると，GC がオブジェクトを移動させるたびに my-callback にコールバックし，移動させるオブジェクトが文字列だった場合，intern によって同一内容の代表オブジェクトで置き換えられる．結果として，それ以降に示されているように string-append のような動的な方法で作られた文字列でも，GC の裏で同一内容の代表オブジェクトで置換することができる．

このようなコールバック機構はリフレクションの一種である．例に挙げた String Deduplication プログラムでは，言語機構である GC に実行時手続きとして定義された my-procedure を反映させていると解釈できる．例えば C++ 言語で記述された Scheme 処理系では，GC は処理系の一部として C++ 言語で実装され，コールバック関数は Scheme 言語で記述されるのが普通である．コールバック関数を登録するという操作は，見方を変えるとプログラムの途中にプログラムを追加していると考えることもできる．すなわち，例に挙げたようなコールバック機構を実現する GC は一部分がその GC 自身によって管理されるプログラミング言語で記述されることになる．そのためこのような GC はその実行中に，自身が管理するべきオブジェクトを追加で生み出してしまう．メモリ消費を削減するためのオブジェクトエボリューション機構が大きなメモリオーバーヘッドを発生させないため，GC 自身が生み出すオブジェクトを適切に管理する方法を検討する必要がある．

```
1  (define intern-table (make-hash))
2  (define intern (lambda (str)
3    (if (hash-has-key? intern-table str)
4      (hash-ref intern-table str)
5      (begin (hash-set intern-table str str)
6             str))))
7  (define my-callback (lambda (obj)
8    (if (string? obj) (intern obj) obj)))
9  (register-gc-callback my-callback)
10
11 (define str1 "an instance")
12 (define str2 (string-append "an " "instance"))
13 (print (eq? str1 str2))  ;;=> #f
14 (gc)
15 (print (eq? str1 str2))  ;;=> #t
```

図 2.10. GC コールバックによる String Deduplication

第 3 章

リフレクティブ GC

オブジェクトエボリューション機構を簡単に開発可能な言語処理系を実現するため，自身が管理するプログラムから改造することが可能な GC であるリフレクティブ GC を提案する．さらにリフレクティブ GC の一つとして，コンパクション中に行われるオブジェクトのコピーをコールバック関数の登録によって改造することができる Buffered GC を開発した．リフレクティブ GC は GC 自身が計算のために大量のオブジェクトを生成し，回収しなければ大量のメモリを消費してしまうため，Buffered GC ではこれらのオブジェクトを一度 Buffer 領域という小さな領域を経由してから通常のヒープに配置することでメモリ消費を抑える．

リフレクティブ GC は自身が管理するプログラミング言語で記述された手続きを反映可能な GC であり，反映された手続きを実行するためにオブジェクトを作成する．特にオブジェクトエボリューションのためにコンパクション中に行われるコピーを改造した GC では，改造部分の呼び出しが膨大な回数行われるため，膨大な量のオブジェクトが作成されることが予測される．プログラムの性能向上を目指す本研究では，リフレクティブ GC 自身によって作成されるオブジェクトの効率的な管理が重要である．

Buffered GC は，GC 自身が作成するオブジェクトを一度バッファー領域という小さな領域に配置し，一定時間を生き延びたオブジェクトだけを通常ヒープ領域へ移動させる GC アルゴリズムである．世代仮説 [14] によると，多くのオブジェクトは非常に短い時間でその役目を終え，GC によって回収される．GC が作成したオブジェクトを同じ GC ループで回収することは難しいが，バッファー領域を経由して通常ヒープへ合流させることで GC 自身が作成するオブジェクトによるメモリ消費を大幅に削減できることができることが予想できる．

さらに実験のために，簡単な Scheme 処理系と Copying GC をベースとした Buffered GC である Buffered Copying GC を開発した．3.1 節では実装した Scheme 処理系について，3.2 節では Buffered Copying GC について詳しく説明する．

3.1 Scheme 処理系

リフレクティブ GC のメモリ効率を計測する実験のために，まず，GC を改造しやすい小規模な Scheme 処理系を C++ 言語で実装した．この処理系はコピー GC によってメモリを管

```
1  class SchemeObject {
2  public:
3    SchemeObject *forwardPointer;
4  public:
5    SchemeObject():forwardPointer( nullptr ){}
6  public:
7    virtual ~SchemeObject(){}
8  public:
9    virtual SchemeObject *apply( SchemeObject *args,
10                                 VM *vm,
11                                 SchemeObject *env ){
12     /* applyable ではないオブジェクトが apply された */
13     /* エラーを出力して終了 (例外機構は実装していない) */
14   }
15 };
```

図 3.1. SchemeObject クラスの定義

理する．メモリ効率を調べる実験のために開発したため，計算性能は重視されていない．例えば，数値型はアンボックス化されず，Scheme レベルの関数呼び出しは Scheme スタックだけでなく，C++ スタックも消費する．また，ルートセットの統一的な管理機構は存在せず，半手動で管理している．そのため，ルートセットを詳細に制御することが可能であり，生存期間中にメモリアロケーションが発生しない参照にはルートセットに追加せずに済ましているものがある．

3.1.1　Scheme Object

今回実装した Scheme 処理系では Nil，Symbol，Cons，True，Integer，PrimOp，Quote，Function，Array，String，Hashtable，Env がプリミティブ型として提供される．True は真を表す #t が唯一の値であるような型である．PrimOp はプリミティブ手続きを表す型であり，if や define は PrimOp 型の値として扱われる．Env は環境を表す Hashtable のラッパーであり，ユーザーは直接操作することはできない．

コピー GC はオブジェクトを移動した時，古いオブジェクトに移動先を表すポインタを書き残す．このポインタを forward pointer と呼ぶ．今回の実装ではそれぞれのオブジェクトが forward pointer をオブジェクトヘッダに持つ．そのため，Scheme オブジェクトを表すクラスは図 3.1 のように定義した．それぞれのプリミティブ型は SchemeObject を継承したクラスとして表現される．また，それぞれのプリミティブ型は Scheme ヒープ上に作成したインスタンスを返す static メソッド newXXX を定義した．

以下，それぞれのプリミティブ型の定義を説明する．

Nil は nil が唯一の値であり，それ以外の属性を持たない．そのため，C++ 上で Nil を表す SchemeNil クラスはシングルトンクラスとして定義される．Nil の値は唯一 nil のみなので，SchemeNil インスタンスは静的変数として定義し，GC とは関係しない．SchemeNil ク

```
1  class SchemeNil : public SchemeObject {
2  private:
3    SchemeNil(){
4      forwardPointer = this;
5    }
6  public:
7    static SchemeNil *getInstance(){
8      static SchemeNil instance;
9      return &instance;
10   }
11 };
```

図 3.2. SchemeNil クラスの定義

ラスの定義を図 3.2 に示す．forwardPointer を this に設定しておくことで，GC の中で特別な処理なしに SchemeNil インスタンスのコピーを抑止することができる．Nil オブジェクトは Scheme ヒープに配置されないので，他のプリミティブ型のように newInstance メソッドではなく，getInstance メソッドでインスタンスを得る．同じくシングルトンクラスである SchemeTrue の定義も同様である．

SchemeInteger は Scheme 上の整数値を表す型である．大小比較や四則演算を行うことができる．SchemeInteger は内部にその値を表す int を持つ SchemeObject として実装した．このように数値型のオブジェクトを作成する方針はオブジェクトをパッキングしていると言う．対象的に数値型のオブジェクトを作成せず，数値型オブジェクトを指すポインタに埋め込んでしまう方針をアンパッキングと言う．アンパッキングした方が計算性能が向上することが知られているが，今回の実験では計算性能には関心がないため，実装が単純であるパッキングの方針を採用した．

SchemeInteger の定義を図 3.3 に示す．SchemeInteger に限らず，SchemeObject はコンストラクタを公開しない．間違えて C++ スタックや C++ ヒープに配置してしまうことを避けるためである．メモリは VM#allocate メソッドを利用してアロケートする．VM#allocate は必要ならば GC を行い，ヒープから指定されたサイズのメモリを確保して返す．VM#allocate はメモリの初期化は行わないので，明示的に placement new などの方法で初期化する必要がある．

SchemeSymbol は Scheme 上の変数名を表すクラスである．本実装における SchemeSymbol は自身が表す名前を char 配列として保持する．主に処理系がソースコードを解析し，S 式に変換する過程で作成される．ユーザーは Quote を利用することで Symbol にアクセスすることができる．

Symbol は生成時に intern し，アドレス比較のみで同値判定できる設計が一般的だが，今回の実装では計算性能には関心がないため，intern は行っていない．そのため，二つのシンボルを比較するためには毎回内部の文字列を調べる必要がある．

SchemeSymbol の定義を図 3.4 に示す．SchemeSymbol が内部に保持する文字列の長さは

```
1  class SchemeInteger : public SchemeObject {
2  public:
3    int value;
4  private:
5    SchemeInteger( int value_ ):value( value_ ){}
6  public:
7    static SchemeInteger *newInstance( VM *vm, int value ){
8      SchemeInteger *result = \
9        vm->allocate( sizeof( SchemeInteger ) );
10     new( result ) SchemeInteger( value );
11     return result;
12   }
13 };
```

図 3.3. SchemeInteger クラスの定義

静的には決定できないため，名前文字列は通常の方法ではフィールドに格納できない．C++ 的な作法では std::string オブジェクトとしてフィールドに格納するが，コピー GC は回収したオブジェクトのデストラクタを呼び出すことはできないため，この方法ではメモリリークが発生してしまう．幸い本実装では，SchemeObject は C++ の通常の方法でインスタンス化せず，newInstance メソッドによってインスタンス化される設計であるため，newInstance メソッド内で適切な大きさのメモリブロックをアロケートすることができる．その結果，名前を表す文字列は C++ が管理するオブジェクトの外に配置されるので，利用するために，故意に領域外アクセスをする．SchemeSymbol で最後に定義されるフィールドは char name[1] なので，name[i] によって領域外に配置された名前文字列にアクセスすることができる．SchemeSymbol クラスは C++ が想定する普通のクラスではないので，sizeof 演算子や継承などの操作は直感と異なる結果を引き起こす．

本実装が提供するプリミティブ型である，Array と String を実装したクラス SchemeArray と SchemeString は同様の実装で実現される．

Cons はペアを表す型である．プログラムを表す S 式の構築に使われるほか，ユーザーが主に利用するデータ構造でもある．Scheme 上の Cons を表すクラス SchemeCons はフィールドに SchemeObject * 型の car と cdr を保持する．

SchemeCons の定義を図 3.5 に示す．SchemeCons::newInstance は引数に SchemeObject を指すポインタを取るが，新しいインスタンスを配置するためのメモリをアロケートするために GC が発生した結果，指している先のオブジェクトが移動してしまい，ポインタが無効化されてしまう場合がある．VM#observe はそのような状況を回避するためのメソッドである．VM は observe に渡されたポインタのアドレスを記憶し，GC によってオブジェクトが移動した時，監視しているポインタの値を適切に書き換えることで整合性を保つ．VM#observe が返す ObjectObserver はポインタが監視される期間を表すオブジェクトであり，ObjectObserver が廃棄された時，ポインタの監視も終了する．ObjectObserver はムーブコンストラクタによって別の変数に監視期間を引き継ぐこともできる．VM#observe が

```
1  class SchemeSymbol : public SchemeObject {
2  public:
3    int  length;
4    char name[1];
5  private:
6    SchemeSymbol( std::string const &name_ )
7      :length( name_.length() ){
8      std::copy( name_.begin(), name_.end(), name );
9    }
10 public:
11   static SchemeSymbol *newInstance( VM *vm,
12                                     std::string const &name ){
13     size_t size = \
14       sizeof( SchemeSymbol ) + ( length - 1 ) * sizeof( char );
15     SchemeSymbol *result = vm->allocate( size );
16     new( result ) SchemeSymbol( length, name );
17     return result;
18   }
19 };
```

図 3.4. SchemeSymbol クラスの定義

```
1  class SchemeCons : public SchemeObject {
2  public:
3    SchemeObject *car;
4    SchemeObject *cdr;
5  private:
6    SchemeCons( SObject *car_, SObject *cdr_ )
7      :car( car_ ), cdr( cdr_ ){}
8  public:
9    static SchemeCons *newInstance( VM *vm,
10                                   SchemeObject *car,
11                                   SchemeObject *cdr ){
12     ObjectObserver carObserver = vm->observe( car );
13     ObjectObserver cdrObserver = vm->observe( cdr );
14     SchemeCons *result = vm->allocate( sizeof( SchemeCons ) );
15     new( result ) SchemeCons( car, cdr );
16     return result;
17   }
18 };
```

図 3.5. SchemeCons の定義

ObjectObserver オブジェクトを返すことができるのはこの引き継ぎ機能を利用しているためである．ObjectObserver をポインタ変数と同じスコープに配置することで，ポインタが有効なすべての期間で監視下に置くことができる．

Scheme 上のクォートされたオブジェクトを表す SchemeQuote クラスも，フィールドは quoted 一つだけではあるが，同様の実装で実現される．

```
1  class SchemePrimOp : public SchemeObject {
2  public:
3  private:
4    SchemePrimOp(){
5      forwardPointer = this;
6    }
7    virtual SchemeObject *apply( SchemeObject *args,
8                                 VM *vm,
9                                 SchemeObject *env ) = 0;
10 };
```

Scheme 上の手続きは組み込みと Scheme レベル関数に分類することができる．組み込み手続きは処理系に組み込まれた手続きであり，手続は Scheme レベルではなく，C++ レベルで定義される．本実装が提供する組み込み手続きには print，define，if，nil?，quote，begin，lambda などがある．Scheme レベル関数は Scheme レベルで定義された関数であり，lambda によって作成される．本実装ではマクロはサポートしていない．

SchemePrimOp は Scheme 上の組み込み手続きを表すクラスである．それぞれの SchemePrimOp の実態は apply メソッドとして，派生先で定義される．それぞれの SchemePrimOp 派生クラスはシングルトンクラスであり，SchemeNil や SchemeTrue と同様に扱う．そのため，SchemePrimOp クラスは forwardPointer を this に設定する．また，それぞれの SchemePrimOp 派生クラスはインスタンスを返す getXXX 静的メソッドを持つ．

SchemeFunction は Scheme レベル関数を表すクラスである．クラスの定義を図 3.6 に，呼び出し時のセマンティクスを定義する SchemeFunction::apply の定義を図 3.7 に示す．SchemeFunction::newInstance の定義は割愛したが，SchemeCons のものと同様である．SchemeFunction は仮引数リスト，呼び出されたときに実行される手続きを表す S 式，定義時の環境をフィールドに持つ．定義時の環境はレキシカルスコープを実現するために必要である．

SchemeObject#apply 中で使用される isNil や asCons は SchemeObject を便利に使うためのユーティリティ関数である．isXXX は SchemeObject が XXX 型かどうかを検査し，asXXX は dynamic_cast SchemeXXX クラスに型変換したした結果を返す関数である．どちらも実行時型情報を使って判定を行う．

SchemeObject#apply は，そのオブジェクトが先頭要素であるようなリストを評価するとき，評価器によって呼び出される．引数は未評価の引数リスト，VM オブジェクト，呼び出し元環境の三つ組みである．SchemeFunction#apply は，今回の呼び出しのためのローカル環境 (あるいはスタックフレーム) を作成し，順に評価した引数をローカル環境に登録し，その後に関数の本体を定義時の環境の下で評価する．SchemeFunction 自身もまた Scheme ヒープにアロケートされる SchemeObject であるため，SchemeFunction#apply の途中で GC が起きると this ポインタが無効化されてしまい，フィールドアクセスが未定義動作となってしまう．本実装ではフィールドをローカル変数でハイディングし，ローカル変数を監視することでこの

```
1  class SchemeFunction : public SchemeObject {
2  private:
3    SchemeObject *params;  /* 仮引数リスト */
4    SchemeObject *body;    /* 関数本体を表す S 式 */
5    SchemeObject *env;     /* 定義時の環境 */
6  private:
7    SchemeFunction( ... ){ ... }
8  public:
9    static SchemeFunction *newInstance( ... ){ ... }
10   SchemeObject *apply( ShcemeObject *args,
11                        VM *vm,
12                        SchemeObject *env );
13 };
```

図 3.6. SchemeFunction の定義

問題を回避した．また，定義時環境を表す env はメモリアロケーション後に参照されず，監視しなくても正しく動作するため，監視していない．return eval(...) 以降に SchemeObject を参照することはないため，eval 呼び出しの前に監視を終了することで VM が監視するポインタを減らし，一部のオブジェクトを回収できるようになる．監視対象を手動で管理しているため，このような細かい最適化の余地はまだ多い．

SchemeEnv は Scheme 上の環境を表すクラスである．SchemeSymbol が表す名前と SchemeObject のペアを記憶するハッシュテーブルとして実現した．SchemeEnv は Scheme 上のスコープや SchemeFunction と連動した寿命を持つため，SchemeObject として実装することで寿命を簡単に管理することができる．SchemeObject は C++ ヒープへの参照を持つことができないため，ハッシュテーブルも SchemeObject として実装する必要がある．

SchemeHashtable の定義を図 3.8 に，SchemeEnv の定義を図 3.9 に示す．SchemeHashtable の実装は大半を割愛したが，簡単なクローズドハッシュを実装した．キーバリューペアを格納する配列もまた SchemeObject である必要があるので，SchemeArray を使用している．SchemeEnv は親環境を表す parent と名前と値の対応を記憶するハッシュテーブル table をフィールドに持つ．SchemeEnv#getVariable メソッドで名前を引くことができる．評価器がシンボルを評価するために利用する．SchemeEnv#getVariable メソッドは自身の table に名前が格納されていた場合，自身の table から値を取り出し，返す．存在しなかった場合は，親環境に問い合わせることでスコープチェーンを実現している．スコープチェーンがなければ，lambda や let のようなローカル環境を作る手続きの中から外の環境を見ることができない．SchemeEnv#getVariable メソッドは内部でメモリアロケーションをしないため，引数に VM を要求しない．SchemeEnv#setVariable メソッドで新しい名前を定義することができる．setVariable は途中でキーバリューペアを表す Cons セルを作り，クローズドハッシュをリハッシュすることもあるため，引数に VM を要求する．

```
1  SchemeObject *SchemeFunction::apply( SchemeObject *args,
2                                       VM *vm,
3                                       SchemeObject *env ){
4    SchemeObject *params     = this->params;
5    SchemeObject *body       = this->body;
6    SchemeObject *lexicalEnv = this->env;
7    SchemeObject *localEnv   = nullptr;
8
9    ObjectObserver argsObserver     = vm->observe( args );
10   ObjectObserver envObserver      = vm->observe( env );
11   ObjectObserver paramsObserver   = vm->observe( params );
12   ObjectObserver bodyObserver     = vm->observe( body );
13   ObjectObserver localEnvObserver = vm->observe( localEnv );
14
15   localEnv = SchemeEnv::newInstance( lexicalEnv );
16   while( !isNil( params ) ){
17     SObject *value = eval( asCons( args )->car, vm, env );
18     SObject *key   = asCons( params )->car;
19     localEnv->setVariable( vm, key, value );
20     params = asCons( params )->cdr;
21     args   = asCons( args   )->cdr;
22   }
23
24   return eval( body, vm, localEnv );
25 }
```

図 3.7. SchemeFunction::apply の定義

```
1  class SchemeHashtable : public SchemeObject {
2  public:
3    int size;
4    SchemeObject *kvpairs;
5    ...
6    static SchemeHashtable *newInstance( VM *vm,
7                                         int initialCapacity ){
8      ...
9    }
10   bool hasElement( SchemeObject *key ){ ... }
11   SchemeObject *getElement( SchemeObject *key ){ ... }
12   void setElement( VM *vm,
13                    SchemeObject *key,
14                    SchemeObject *value ){ ... }
15   void deleteElement( SchemeObject *key ){ ... }
16 };
```

図 3.8. SchemeHashtable の定義

```
1  class SchemeEnv : public SchemeObject {
2  public:
3    SchemeObject *parent;  /* 親環境 */
4    SchemeObject *table;   /* ハッシュテーブル */
5  private:
6    SchemeEnv( SchemeObject *parent_, SchemeObject *table_ )
7      :parent( parent_ ), table( table_ ){}
8  public:
9    static SchemeEnv *newInstance( VM *vm, SchemeObject *parent ){
10     ...
11     SchemeObject *table = SchemeHashtable::newInstance( vm, 7 );
12     new( result ) SchemeEnv( parent, table );
13     return result;
14   }
15   SchemeObject *getVariable( SchemeObject *key ){
16     if( table->hasElement( key ) ){
17       return table->getElement( key );
18     }else if( parent ){
19       /* key は管理下の変数ではなかった */
20       /* 親環境に問い合わせる */
21       return parent->getVariable( key );
22     }else{
23       /* key という名前の変数は存在しない */
24       /* エラーメッセージを出力して終了 */
25     }
26   }
27   void setVariable( VM *vm, SchemeObject *key, SchemeObject *value ){
28     table->setElement( vm, key, value )
29   }
30 };
```

図 3.9. SchemeEnv の定義

3.1.2 評価器

評価器は Visitor パターンを利用して実装した．SchemeObject の Visitor である SchemeObjectVisitor の定義を図 3.10 に示す．Visitor パターンを利用するためには SchemeObject#accept が定義されている必要があるが，その定義は引数の Visitor オブジェクトの，自身のクラスに対応する visit メソッドに this ポインタを渡すような自明なものであるため，前節の SchemeObject の定義では割愛した．

評価器の定義を図 3.11 に示す．eval 関数は呼び出し側を簡潔に記述するためのアダプタであり，評価器の実体は EvalVisitor である．Scheme の評価器は Integer や Nil など多くのデータ構造について，変形せずにそのまま返す．変形が行われるのは Symbol，Cons，Quote のいずれかが評価された時である．Symbol を評価するということは変数を参照するということを意味し，評価される Symbol オブジェクトが表す名前には現在の環境でどのような値が

```
1       class SchemeObjectVisitor {
2       public:
3         virtual void visitNil( SchemeNil *host ) = 0;
4         virtual void visitSymbol( SchemeSymbol *host ) = 0;
5         virtual void visitCons( SchemeCons *host ) = 0;
6         /* 他のプリミティブ型についても同様 */
7       };
```

図 3.10. SchemeObjectVisitor の定義

束縛されているか調べ，その値を返す．Cons の評価は手続きの呼び出しを意味し，評価される Cons オブジェクトの car が表す手続きに cdr が表す引数リストが渡される．それぞれのオブジェクトは自身が呼び出された時に実行する手続きを SchemeObject#apply として記憶しているので，評価器は呼び出される手続きを表すオブジェクトの apply を呼び出し，結果を返す．Quote の評価は quote されていたオブジェクトを返すので，SchemeQuote が内部に持っている，quote されていたオブジェクトを取り出し，返す．

3.1.3　GC

本実装ではメモリをコピー GC によって管理する．コピー GC はヒープを二つの領域に分割し，普段は一方のみを使用し，GC が必要になると生存している全てのオブジェクトをもう一方の領域にコピーし，二領域の役割を入れ替えることで GC とコンパクションを同時に行う GC である．本実装では SchemeHeap クラスがメモリアロケーションや GC などのメモリ管理を担当し，SchemeHeap の派生クラスであり，コピー GC を実装した CopyHeap が実際に使用される．CopyHeap はコピー GC によってヒープを管理するため，内部に new と old の二つの領域を持つ．これらの領域は Region クラスによって表現される．Region クラスは実際にオブジェクトが配置されるメモリチャンクを指す chunk，メモリチャンクの最大サイズを表す capacity，メモリチャンクの消費バイト数を表す size の三つをフィールドに持つ．Region クラスの定義を図 3.12 に示す．メモリリークを避けるため，chunk は scoped_ptr<void >型とした．Region クラスはメソッドとしてメモリアロケーションを行う allocate の他に，サイズを 0 に戻す clear，メモリチャンク上の pos バイト目のアドレスを返す at，ポインタが chunk メモリチャンク上を指しているかどうかを判定する is_in を持たせた．Region#allocate メソッドは要求されたバイト数だけ size を増やすが，この時，$-size\&7$ を加えることで，次回のアロケーションが 8 バイト境界上に行われるよう調整する．

SchemeHeap クラスは VM から見たヒープのインターフェースを定めるクラスである．SchemeHeap クラスの定義を図 3.13 に示す．SchemeHeap クラスが提供するインターフェースは allocate と observe の二つだけである．SchemeHeap#allocate メソッドはヒープから指定されたサイズ以上の領域を確保し，その先頭アドレスを返す．SchemeHeap#observe メソッドは各手続きがヒープ内を指すポインタを作成した時に作成されたポインタを引数に呼び出される．通常 SchemeHeap#observe メソッドは，引数に受け取ったポインタをルー

```
1  class EvalVisitor : public SchemeObjectVisitor {
2  public:
3    VM *vm;
4    SchemeObject *result;
5    SchemeObject *env;
6    ObjectObserver envObserver;
7  public:
8    EvalVisitor( VM *vm_, SchemeObject *env_ ):vm( vm_ ), env( env_ ){
9      envObeserver = vm->observe( env );
10   }
11 public:
12   void visitSymbol( SchemeSymbol *host ){
13     /* 変数参照  現在の環境に問い合わせる */
14     result = asEnv( env )->getVariable( host );
15   }
16   void visitCons( SchemeCons *host ){
17     /* 関数呼び出し 先頭要素の SchemeObject#apply メソッドを呼び出す */
18     SchemeObject *args = host->cdr;
19     ObjectObserver argsObserver = vm->observe( args );
20     SchemeObject *proc = eval( host->car, vm, env );
21     result = proc->apply( args, vm, env )
22   }
23   void visitQuote( SchemeQuote *host ){
24     /* Quote されているオブジェクトを取り出す */
25     result = host->quoted;
26   }
27   void visitNil( SchemeNil *host ){
28     /* Cons, Quote 以外は Nil と同様に引数の host を result に代入する */
29     result = host;
30   }
31   ...
32 };
33
34 SchemeObject *eval( SchemeObject *expr, VM *vm, SchemeObject *env ){
35   EvalVisitor visitor( vm, env );
36   expr->accept( visitor );
37   return visitor.result;
38 }
```

図 3.11. 評価器の定義

トポインタとして内部のデータベースに登録し，登録を解除するようなデストラクタを持つ ObjectObserver オブジェクトを返す．各手続きは allocate や observe を VM に依頼するが，VM 自身は何もせず，SchemeHeap オブジェクトに委譲するような設計になっている．

CopyHeap はヒープをコピー GC で管理する SchemeHeap クラスの派生クラスである．CopyHeap クラスの定義を図 3.14 に示す．CopyHeap#do_gc にはコピー GC のアルゴリズムを記述した．CopyHeap#allocate は Region#allocate に委譲する前に oldRegion の空き容量を調べ，少なかった場合，do_gc メソッドを呼び，GC を行う．CopyHeap#do_gc は new

領域の拡張，ルートのコピー，old 領域の幅優先探索，old 領域と new 領域の入れ替え，new 領域の後始末，new 領域拡張の予約の 6 つのステップから成る．

第一のステップは new 領域の拡張である．前回の GC で拡張が予約されていた場合，new 領域を old 領域の 2 倍の大きさを持つ Region で置き換える．また，この時点で new 領域がヌルである場合があるので，その場合は old 領域と等しい大きさの Region で初期化する．

第二のステップはルートのコピーである．ルートセットに登録されているオブジェクトを順次コピーし，同時にルートポインタを書き換える．copyObject 関数は第一引数の SchemeObject を第二引数の Region にコピーし，コピー後のオブジェクトを指すポインタを返す関数である．copyObject 関数の定義を図 3.16 に示す．copyObject はヌルポインタを渡された場合，何もせずにヌルポインタを返す．また，forwardPointer が設定されている SchemeObject はコピー済みであるため，コピーはせずに forwardPointer を返す．そのどちらでもない場合はコピーを行う．schemeObjectSize 関数はオブジェクトが占めるヒープのバイト数を返す関数である．copyObject 関数は schemeObjectSize 関数が計算したサイズのメモリブロックを dst Region にアロケートし，mempcpy で内容をコピーすることでオブジェクトをコピーする．また，オブジェクトヘッダに forwardPointer を書き込む操作も copyObject 関数が行う．

第三のステップは old 領域の幅優先探索である．都合よく new 領域に探索されるオブジェクトが幅優先探索順で並ぶので，これを走査し，コピーするべきオブジェクトをコピーしていく．copyReferencedObjects 関数は第一引数の SchemeObject から参照される SchemeObject の内，第二引数の src Region に配置されているものを第三引数の dst Region にコピーする関数である．ヒープ上で，連続で並んだ前のオブジェクトから次のオブジェクトを引くためには，前のオブジェクトのアドレスに前のオブジェクトのサイズを加えれば良い．この時，8 バイト境界を守るよう注意する必要があるが，schemeObjectSize 関数は 8 バイト境界も考慮したオブジェクトのサイズを返すので，CopyHeap#do_gc メソッド中では特別な操作を必要としない．

第四のステップは old 領域と new 領域の入れ替えである．old 領域と new 領域はそれぞれ std::unique_ptr に管理される Region を指すポインタなので，std::unique_ptr の内容を swap することで入れ替えを実現した．

第五のステップは new 領域の後始末である．次回の GC のため，new 領域をクリアしておく必要がある．また，第一のステップで拡張された領域がこの時点での old 領域である可能性がある．old 領域より小さい new 領域には生存している全てのオブジェクトが入り切らない可能性がありるため，new 領域を old 領域以上の大きさに拡張する必要がある．本実装では， new 領域を表す Region オブジェクトを std::unique_ptr#release メソッドで解放することで，次回の GC の第一ステップで new 領域が初期化される設計を採用した．

第六のステップは拡張の予約である．この時点で old 領域の使用率が高いということはヒープの大部分が生存しているオブジェクトに占められており，この後のプログラムの実行で GC が頻発し，性能低下を招くことが予想される．そのような状況を避けるため，この時点で old 領域の使用率が高ければ，new 領域の拡張を予約する．ここで拡張が予約された場合，次回の

```
1  class Region {
2  public:
3    size_t capacity;
4    size_t size;
5    std::unique_ptr< void > chunk;
6    Region( size_t capacity_ ):capacity( capacity_ ), size( 0 ){
7      chunk.reset( allocate( capacity ) );
8    }
9    void clear(){
10     size = 0;
11   }
12   void *at( size_t pos ){
13     return (void *)( (char *)chunk.get() + pos );
14   }
15   void *allocate( size_t amount ){
16     void *result = at( size );
17     size += amount;
18     size += -size & 7;  // keep 8 byte offset
19   }
20   void is_in( void *ptr ){
21     return at( 0 ) <= ptr && ptr < at( capacity );
22   }
23 };
```

図 3.12. Region の定義

```
1  class SchemeHeap {
2  public:
3    virutal void *allocate( size_t size ) = 0;
4    virtual ObjectObserver observe( SchemeObject *&root ) = 0;
5  };
```

図 3.13. SchemeHeap の定義

GC の第一ステップで new 領域を old 領域の二倍のサイズの領域で置き換える．

3.2 Buffered Copying GC

Buffered GC はコピー GC のように振る舞う通常モードと，たとえ通常モードの GC の途中であったとしてもアロケーションや GC が可能であるバッファリングモードの二つのモードを持つ GC アルゴリズムである．Buffered GC はヒープを三領域に分割して管理する．Buffered GC はコピー GC をベースとした GC アルゴリズムであり，三分割した領域の内，二つの領域をコピー GC で管理する．この二つの領域はコピー GC にならい，Buffered GC の文脈下でもそれぞれ old 領域，new 領域と呼ぶ．第三の領域は buffer 領域と言い，バッファリングモードである間 nursery 領域のように振る舞い，一度も GC を経験していないような生まれたばかりのオブジェクトが配置される．通常モードの Buffered GC は buffer 領

```
1  class CopyHeap : SchemeHeap {
2  public:
3    static constexpr size_t HEAP_SIZE = ...;
4  public:
5    VM *vm;
6    std::unique_ptr< Region > oldRegion;
7    std::unique_ptr< Region > newRegion;
8    bool expandFlag;
9    std::unordered_set< SchemeObject ** > rootset;
10 public:
11   CopyHeap( VM *vm_ )
12     :vm( vm_ ), oldRegion( new Region( HEAP_SIZE ) ),
13      newRegion(), expandFlag( false ){}
14 public:
15   void *allocate( size_t size ){
16     if( /* oldRegion の空きが残り少ない */ ){
17       do_gc();
18     }
19     return oldRegion->allocate( size );
20   }
21   ObjectObserver observe( SObject *&root ){
22     rootset.insert( &root );
23     return ObjectObserver( [this, &root](){
24       rootset.erase( &root );
25     } );
26   }
27   void do_gc();
28 };
```

図 3.14. CopyHeap の定義

域が占有している分だけメモリ効率の悪いコピー GC のように振る舞う．すなわち，メモリアロケーションは old 領域に行われ，old 領域から new 領域へコピーすることで GC とコンパクションを同時に行う．バッファリングモードの Buffered GC はメモリアロケーションを buffer 領域に行い，buffer 領域が不足すると buffer 領域中に存在するオブジェクトを old 領域に書き出すことでスペースを確保する．このオブジェクトの書き出しをフラッシュと呼ぶ．フラッシュの途中で書き出し先を使い切ってしまい，コピーできない状況に陥ってしまった場合，Buffered GC は異常終了し，回復することはできない．

Buffered GC は通常モード GC の途中でもバッファリングモードに入り，アロケーションやフラッシュを行うことができる．通常モード GC の途中でフラッシュした場合，buffer 領域のオブジェクトは new 領域に書き出される．この機能を利用し，GC の途中でバッファリングモードに切り替えることでコピー時リフレクションを通常のユーザープログラムと同等のセマンティクスで呼び出すことができる．

Buffered GC はバッファリングモードから通常モードに戻る時，buffer 領域をフラッシュし，空にする．そのため，Buffered GC が通常モードであり，かつ buffer 領域が空ではない

```
1  void CopyHeap::do_gc(){
2    if( expandFlag ){
3      newRegion.reset( new Region( oldRegion->capacity * 2 ) );
4    }else if( !newRegion ){
5      newRegion.reset( new Region( oldRegion->capacity ) );
6    }
7    for( SObject **root : rootset ){
8      *root = copyObject( *root, newRegion.get() );
9    }
10   size_t i = 0;
11   while( i < newRegion->size ){
12     SObject *cur = newRegion->at( i );
13     copyReferencedObjects( cur, oldRegion.get(), newRegion.get() );
14     i += objectSize( cur );
15   }
16   oldRegion.swap( newRegion );
17   if( newRegion->capacity < oldRegion->capacity ){
18     newRegion.release();
19   }else{
20     newRegion->clear();
21   }
22   if( /* まだ oldRegion の空きが少ない */ ){
23     expandFlag = true;
24   }
25 }
```

図 3.15. CopyHeap::do_gc の定義

```
1  SchemeObject *copyObject( SchemeObject *obj, Region *dst ){
2    if( !obj ){ return nullptr; }
3    else if( obj->forwardPointer ){
4      return obj->forwardPointer;
5    }else{
6      size_t size = schemeObjectSize( obj );
7      SchemeObject *result = dst->allocate( size );
8      memcpy( result, obj, size );
9      obj->forwardPointer = result;
10     return result;
11   }
12 }
```

図 3.16. copyObject の定義

状態にはなりえない．その結果通常モードでは，まるで old 領域，new 領域の二つしか領域がないかのように管理することが可能であり，リメンバードセットのような追加のデータを必要としない．実験のために開発した Buffered GC を実現した処理系では，Copy GC によって管理されるヒープを表す CopyHeap クラスを拡張した ReworkableCopyHeap クラスを利用しているが，この拡張はバッファリングモード中に参照される状態を管理したり，バッファリングモードでユーザープログラムを実行した時に起きる不整合を追跡したりするものであり，通常モード下でのユーザープログラムの実行にはほとんど影響を及ぼさない．

3.2.1　Buffered GC によって管理されるヒープの実装

実験用に開発した Scheme 処理系では，GC によって管理されるヒープは SchemeHeap クラスを継承して実現する．Buffered GC によってメモリを管理するヒープ BufferedGCHeap の定義を図 3.17 に示す．Buffered GC には通常モードとバッファリングモードの二つのモードがあるため，BufferedGCHeap は現時点でどちらのモードなのかを表す bufferingMode をフィールドに持ち，BufferingGCHeap#switchMode メソッドによって切り替える．また，old 領域と new 領域が GC 中であるかどうかによってフラッシュの動作が異なるため，現在 old 領域と new 領域が GC 中であるかどうかを表すフラグ inNormalGC も持つ．BufferedGCHeap は，old 領域と new 領域の管理のために ReworkableCopyHeap クラスを利用する．ReworkableCopyHeap はバッファリングモードのための追加のフィールドを持つ CopyHeap の派生クラスである．ReworkableCopyHeap クラスについて，詳しくは後ほど触れる．また，BufferedGCHeap は buffer 領域を指すポインタを格納する std::unique_ptr<Region >型の bufferRegion 及び，buffer 領域を管理するための bufferRootset と rememberedset を持つ．

BufferedGCHeap#switchMode メソッドは BufferedGCHeap がバッファリングモードであるかどうかを表すフラグである，bufferingMode を反転する．また，フラグを反転する前に現在バッファリングモードであるか調べ，もしバッファリングモードだった場合は buffer 領域をフラッシュする．

BufferedGCHeap#allocate メソッドは現時点のモードがバッファリングモードならば bufferRegion からアロケートした領域を指すポインタを返し，通常モードならば copyHeap に処理を委譲する．また，アロケートされる領域の空きが残り少なかった場合，明示的に GC を呼び出す．BufferedGCHeap#allocate の外側で GC を呼び出すことで GC の開始と終了をフックし，現在コピー GC 中であるかどうかの状態を記憶する inNormalGC フラグを管理する．

BufferedGCHeap#observe はヒープ外からヒープ内を参照するルートポインタの監視を開始するメソッドであり，デストラクタで監視を解除する ObjectObserver オブジェクトを返す．現時点のモードがバッファリングモードだった場合，フラッシュによってルートポインタが移動する可能性を考え，bufferRootset と copyHeap.rootset の両方を調べてルートポインタを削除する ObjectObserver を返す．

ReworkableCopyHeap は Buffering GC のバッファリングモードが引き起こす不整合を記

```
1  class BufferedGCHeap : public SchemeHeap {
2  public:
3    bool                          bufferingMode;
4    ReworkableCopyHeap            copyHeap;
5    std::unique_ptr< Region >     bufferRegion;
6    std::set< SchemeObject ** > bufferRootset;
7    std::set< SchemeObject ** > rememberedset;
8  public:
9    void switchMode(){
10     if( bufferingMode ){
11       flush( false );
12     }
13     bufferingMode = !bufferingMode;
14   }
15   void *allocate( size_t size ){
16     if( bufferingMode ){
17       if( /* bufferRegion の空きが足りない */ ){
18         flush( true );
19       }
20       return bufferRegion->allocate( size );
21     }else{
22       return copyHeap.allocate( size );
23     }
24   }
25   void flush( bool expand );
26   ObjectObserver observe( SchemeObject *&root ){
27     if( bufferingMode ){
28       bufferRootset.insert( &root );
29       return ObjectObserver( [this, &root](){
30         if( bufferRootset.find( &root ) ){
31           bufferRootset.erase( &root );
32         }else{
33           copyHeap.rootset.erase( &root );
34         }
35       } );
36     }else{
37       return copyHeap.observe( root );
38     }
39   }
40 };
```

図 3.17. BufferedGCHeap の定義

憶し，通常モードに戻った後に追跡，修正するような CopyHeap の派生クラスである．ReworkableCopyHeap クラスの定義を図 3.18 に示す．ReworkableCopyHeap はフィールドに，不整合を記憶するための reworkset と，フラッシュのために，現時点が GC 途中の状態であるかどうかを表す inGC フラグを持つ．ReworkableCopyHeap#allocate は CopyHeap#allocate と全く同じ定義である．再定義の意図は do_gc メソッドの呼び出し先を ReworkableCopyHeap#do_gc で置き換えることにある．CopyHeap#do_gc メソッドが virtual メソッドだったならば allocate メソッドを再定義する必要はなかった．ReworkableCopyHeap#do_gc メソッドは CopyHeap#do_gc メソッドと比べて二つの追加機能を持つ．一つ目は inGC フラグの管理である．ReworkableCopyHeap#do_gc メソッドはメソッドの先頭で inGC フラグを立て，メソッドの末尾で inGC フラグを折る．そのため inGC フラグは ReworkableCopyHeap オブジェクトが GC の最中であるかどうかを正しく表し，inGC フラグを参照することで buffer 領域のフラッシュは適切な書き出し先を選択することができる．二つ目はバッファリングモードによって引き起こされる不整合を追跡，修正する機能である．バッファリングモードを利用して GC 中にユーザープログラムを実行した時，ユーザープログラムはオブジェクトの中にある参照を書き換えることができる．この参照が書き換えられたオブジェクトは，もしかすると GC の中ではすでに内部の参照全てを調査し終わったと考えられているかもしれず，もしそうだった場合，新しく書き込まれた参照は改めて調査する必要があるにも関わらず，調査されることはない．その結果，生存しているはずのオブジェクトが回収されてしまい，壊れたオブジェクトが未定義動作を引き起こすことになる．ReworkableCopyHeap では，newRegion 内のオブジェクトに oldRegion を指すような参照が書き込まれた時，書き込まれた参照を reworkset に追加することで記憶しておき，GC の最後のフェイズで，空になるまで reworkset に記憶されている参照を調査することで調査されない参照が生まれることを防ぐ．参照の書き込みは後述するライトバリアによって検出する．

3.2.2　buffer 領域のフラッシュの実装

バッファリングモードでは buffer 領域が不足した時，buffer 領域中のオブジェクトを別の領域に書き出すことで buffer 領域の空き容量を増やす．この書き出し操作をフラッシュと呼ぶ．フラッシュは old 領域と new 領域の間でコピー GC している間にも実行可能であり，その場合，オブジェクトの書き出し先は new 領域となる．buffer 領域のフラッシュはコピー GC と似たアルゴリズムで buffer 領域中のオブジェクトを書き出す．コピー GC と比較すると，rootset だけではなく，rememberedset も移動先を追跡する必要がある点と，コピーした root をコピー先領域のルートセットに加える点が大きく異なる．フラッシュの定義を図 3.19 に示す．フラッシュは書き出し先の選択，書き出し先領域サイズの記録，ルートセットのコピー，リメンバードセットのコピー，buffer 領域オブジェクトの幅優先探索，buffer 領域の後始末の 6 ステップで実現される．

第一のステップは書き出し先領域の選択である．old，new 領域間のコピー GC の途中であれば書き出し先は new 領域に設定され，そうでなければ old 領域に設定される．

```
1  class ReworkableCopyHeap : public CopyHeap {
2  public:
3    std::set< SchemeObject ** > reworkset;
4    bool                        inGC;
5    ReworkableCopyHeap():inGC( false ){}
6    void *allocate( size_t size ){
7      if( /* old 領域の空き容量が残り少ない */ ){
8        do_gc();
9      }
10     return oldRegion->allocate( size );
11   }
12   void do_gc(){
13     inGC = true;
14     CopyHeap::do_gc();
15     while( !reworkset.empty() ){
16       std::vector< SchemeObject ** > reworksetPool;
17       reworksetPool.assign( reworkset.begin(), reworkset.end() );
18       reworkset.clear();
19       for( SchemeObject **rework : reworksetPool ){
20         *rework = copyObject( *rework );
21       }
22     }
23     inGC = false;
24   }
25 };
```

図 3.18. ReworkableCopyHeap の定義

第二のステップは書き出し先領域サイズの記録である．第五のステップで buffer 領域オブジェクトを幅優先探索するため，書き出し先領域を LILO のように扱う．この時点の書き出し先領域のサイズは，書き出し先領域を LILO とみなした時，先頭アドレスを意味する．書き出し先領域のサイズは第三，第四のステップで増えるため，この時点で記録しておく必要がある．

第三のステップはルートセットのコピーである．ルートオブジェクトを書き出し先領域にコピーし，ルートポインタの修正を行う．ルートセットには buffer 領域内のオブジェクトを指すポインタだけではなく，バッファリングモード中に作成されたルートポインタが無差別に入っているため，コピーの前に buffer 領域内のオブジェクトかどうかを判定し，buffer 領域内のオブジェクトだけをコピーする．また，コピーによってルートオブジェクトが領域を移動するため，ルートポインタも移動させる必要がある．ルートポインタの削除は全てのルートオブジェクトの移動が終わってから一括で行うと効率的である．

第四のステップはリメンバードセットのコピーである．リメンバードオブジェクトを書き出し先領域にコピーし，リメンバードポインタの修正を行う．一度でも buffer 領域内を指した old，new 領域内のポインタはリメンバードセットに追加され，フラッシュされるまで削除されることはない．そのため，一度 buffer 領域を指した参照が再度変更された時，リメンバー

```
1  void BufferedGCHeap::flush( bool expand ){
2    Region *dst;
3    if( inNormalGC ){
4      dst = copyHeap.newRegion.get();
5    }else{
6      dst = copyHeap.oldRegion.get();
7    }
8    size_t i = dst->size;
9    for( SchemeObject **root : bufferRootset ){
10     if( bufferRegion->is_in( *root ) ){
11       *root = copyObject( *root, dst );
12     }
13     copyHeap.rootset.insert( root );
14   }
15   bufferRootset.clear();
16   for( SchemeObject **remembered : rememberedset ){
17     if( bufferRegion->is_in( *root ) ){
18       *remembered = copyObject( *rememberd );
19     }
20   }
21   rememberedset.clear();
22   while( i < dst->size ){
23     SchemeObject *cur = dst->at( i );
24     copyReferencedObjects( cur, bufferRegion.get(), dst );
25     i += schemeObjectSize( cur );
26   }
27   if( expand ){
28     bufferRegion.reset( new Region( bufferRegion->capacity * 2 ) );
29   }else{
30     bufferRegion->clear();
31   }
32 }
```

図 3.19. BufferedGCHeap#flush の定義

ドセットには buffer 以外の領域を指すポインタが格納される．リメンバードセットのコピーでもルートセットのコピーと同様に，コピーの前に buffer 領域内のオブジェクトを指しているかどうかを検査する必要がある．リメンバードポインタはルートポインタと異なり，別領域のリメンバードセットに移ることなく削除される．リメンバードポインタは new，old 領域から buffer 領域を指す領域をまたぐ参照であるが，コピー後は new，old 領域内の閉じた参照となるため，リメンバードポインタではなくなるためである．

第五のステップは buffer 領域オブジェクトの幅優先探索である．このステップはコピー GC のものと同様であり，コピーされたオブジェクトが幅優先探索順に並ぶ書き出し先領域を走査しつつコピーが必要なオブジェクトを書き出す．

第六のステップは buffer 領域の後始末である．buffer 領域の拡張が要求されている場合は倍の大きさの領域で buffer 領域を置き換え，そうでなければ size を 0 に戻す．

```
1  void BufferedGCHeap::writeBarrier( SchemeObject *owner,
2                                     SchemeObject **ptr,
3                                     SchemeObject *value ){
4    if( bufferingMode ){
5      if( !bufferRegion->is_in( owner ) && \
6           bufferRegion->is_in( value ) ){
7        rememberedset.insert( ptr );
8      }
9      if( copyHeap.inGC && \
10         copyHeap.newRegion->is_in( owner ) && \
11         copyHeap.oldRegion->is_in( value ) ){
12       copyHeap.reworkset.insert( ptr );
13     }
14   }
15   *ptr = value;
16 }
```

図 3.20. BufferedGCHeap#writeBarrier の定義

3.2.3 Buffered GC が要求するバリア

Buffered GC によってヒープが正しく管理されるためにはオブジェクトへの書き込みと読み出しをフックし，追加の処理を行う必要がある．書き込みのフックと読み出しのフックをそれぞれライトバリア，リードバリアと呼ぶ．バリアなしでは，バッファリングモード中に作成されたオブジェクトや参照を正しく追跡することができず，生存しているオブジェクトが回収されてしまう場合がある．

BufferedGC が要求するライトバリアの定義を図 3.20 に示す．ユーザープログラムや処理系がオブジェクト内の参照を書き換える時，代入文を記述する代わりに writeBarrier 関数を呼び出す．wirteBarrier 関数が受け取る引数はそれぞれ，書き換えられる参照の持ち主であるオブジェクトを指す owner，書き換えられる参照を指す ptr，書き換え後の値を表す value である．writeBarrier 関数は通常モードでは何もせず，ただ value を *ptr に代入する．バッファリングモードでの writeBarrier は owner が buffer 領域外に存在し，かつ value が buffer 領域内に存在する時，ptr を buffer 領域のリメンバードセットに追加する．また，copyHeap が GC の途中であり，owner も value も buffer 領域外だった場合，GC の中では全ての参照の更新が終わったと考えられているオブジェクトが，しかし書き換えられたために再度参照を更新する必要が発生することがある．このような参照を copyHeap.reworkset に記憶させることで，ReworkableCopyHeap#do_gc メソッドの最後で参照が更新させることができる．

バッファリングモードを利用して old 領域の GC 中にユーザープログラムを動かす時，すでにコピーされたオブジェクトを操作したために，オブジェクトへの変更が GC の後で無効化されてしまう事がある．オブジェクトの読み出し時に forward ポインタを調べ，コピー済みのオブジェクトであるかどうかを判定することでこの問題を回避することができる．

```
1  SchemeObject *BufferedGCHeap::readBarrier( SchemeObject *ptr ){
2    if( copyHeap.inGC ){
3      return (ptr->forwardPointer ? ptr->forwardPointer : ptr);
4    }else{
5      return ptr;
6    }
7  }
```

図 3.21. BufferedGCHeap#readBarrier の定義

Buffered GC が要求するリードバリアを図 3.21 に示す．ユーザープログラムや処理系がオブジェクトを読み出す時，読み出したいオブジェクトを指すポインタを readBarrier 関数に渡すと，真に参照されるべきオブジェクトを指すポインタが返ってくるので，ユーザープログラムや処理系は帰ってきたオブジェクトを使用して計算を進める．readBuffer は copyHeap が GC 中でなければ何もせず，渡されたポインタをそのまま返す．もし GC 中だった場合は forward ポインタを調べることでオブジェクトがコピー済みであるかどうかを調べ，コピー済みである場合は forward 先オブジェクトを指すポインタを返す．

3.3　Buffered GC を利用したコピー時リフレクションの実現

Buffered GC は通常モードの GC の途中でもバッファリングモードに切り替えることでメモリアロケーションや GC をすることが可能であり，この機能を利用することで，コピー時リフレクションを実現することができる．コピー時リフレクションを実現するためには，ReworkableCopyHeap 内で呼び出される copyObject 関数を reflectiveCopyObject 関数で置き換える．reflectiveCopyObject 関数は引数に受け取ったオブジェクトを引数に登録されているコピー時メタプログラムを呼び出し，メタプログラムが返した結果をコピー先領域にコピーし，そのコピー先を引数に受け取ったオブジェクトのコピー先として forward ポインタに登録する．メタプログラムはバッファリングモード内でなければ呼び出せないため，メタプログラムを呼び出す前にバッファリングモードに切り替え，コピーが完了した後に通常モードに戻す．また，メタプログラムを呼び出すためにはメタプログラムの引数を格納するリストとメタプログラムのローカル環境が必要である．これらが buffer 領域に配置されるよう，バッファリングモードに切り替えてから作成する．

reflectiveCopyObject 関数の定義を図 3.22 に示す．ヌルポインタや，forward ポインタが設定されているコピー済みのオブジェクトはコピーする必要が無いため，コピーせずに結果を返す．それ以外のオブジェクトはコピーする．オブジェクトをコピーするためにはメタプログラムを起動して変形するかどうか問い合わせる必要があり，メタプログラムを起動するためにはバッファリングモードに切り替える必要がある．VM の switchMode を呼び出すことで Buffered GC をバッファリングモードに切り替える．次に，引数リストが必要である．引数リストは Scheme リストであり，SchemeCons::newInstance を呼び出すことで作成される．

```
1  SchemeObject *reflectiveCopyObject( SchemeObject *obj,
2                                      Region       *dst,
3                                      VM           *vm,
4                                      SchemeObject *metaprogram ){
5    if( obj ){ return obj; }
6    if( obj->forwardPointer ){ return obj->forwardPointer; }
7    vm->switchMode();
8    SchemeObject *nil = SchemeNil::getInstance();
9    SchemeObject *quoted = SchemeQuote::newInstance( vm, obj );
10   SchemeObject *args = SchemeCons::newInstance( vm, quoted, nil );
11   SchemeObject *evolved = metaprogram->apply( args, vm, vm->topLevel() );
12   {
13     ObjectObserver evolvedObserver = vm->observe( evolved );
14     vm->switchMode();
15   }
16   SchemeObject *result = copyObject( evolved );
17   obj->forwardPointer = result;
18 }
```

図 3.22. reflectiveCopyObject の定義

引数の中身はコピーされるオブジェクトをクォートした Quote オブジェクトである．関数オブジェクトは引数を評価してから本体を実行するため，Quote しなければこの評価によってオブジェクトが変形してからユーザープログラムに入ってしまう．メタプログラムが終了し，コピーされるべきオブジェクトが返ってきたならば，次は Buffered GC を通常モードに切り替える．この時 buffer 領域がフラッシュされるため，コピーされるオブジェクトが回収されてしまう可能性がある．そのため，通常モードに復帰する前にコピーされるオブジェクトをルートセットに追加する．Buffered GC が通常モードに復帰したならば，次にコピーを実行する．このコピーにはリフレクティブではない版の copyObject 関数を利用する．そして最後に，引数に受け取ったオブジェクトの forward ポインタにコピーの結果を登録する．

第 4 章

実験

Buffered GC を利用して開発されたがリフレクティブ GC が，コピー時リフレクションを利用したプログラムを実行ためにどのくらいの大きさのヒープを最低限必要とするかを調べる実験を行った．ヒープ効率を計測する方法として，最大ヒープサイズを定め，アプリケーションを実行できる最小の最大ヒープサイズを計測する方法が一般的である．Buffered GC では，old 領域と new 領域は同じ大きさを持つが，buffer 領域は独自の大きさを持つ．old 領域，new 領域のサイズと buffer 領域のサイズの関連は分かっていないため，本実験ではそれぞれを変化させながら実行可能かどうかを調べた．実行可能かどうかは実行時例外が発生したかどうかで判定しており，実行時例外の原因がヒープ不足なのか，処理系にバグがあるのか，あるいはそれ以外の原因なのか調査できていない．実行するアプリケーションとして，コピー時リフレクションを利用して開発した String Deduplication の元で同一内容の文字列を多数生成するプログラムを作成した．アプリケーションについては後節で詳しく触れる．実験環境は Ubuntu16.04 LTS，Intel(R) Core i7-4770S CPU @3.10 GHz × 8，メモリは 16G バイトである．また比較対象として，リフレクションが利用する分のヒープを残した状態でコピー GC を開始し，GC が終わるまで GC を行わないリフレクティブ GC を実現した処理系でも同様の実験を行った．

4.1　実験に使用した Scheme アプリケーション

実験に使うアプリケーションとして，リフレクティブ GC を利用して開発した Stinrg Deduplication の下で，文字列を二文字ずつに切り分け出現頻度を数えるプログラムを使用した．実験用に開発した String Deduplication は String によるメモリ消費を削減するが，しかしすでに intern された文字列も GC のたびに intern されてしまうため，実行時間に多大なオーバーヘッドを発生させる．リフレクティブ GC を用いた String Deduplication の性能を向上するためには，GC のより多くの部分をリフレクティブに開発する必要がある．

アプリケーションプログラムのソースコードを図 4.1 に示す．最初の二つの define は intern 関数の定義である．intern 関数は引数に受け取った文字列が intern された文字列を格納するハッシュテーブル intern-table に登録されているか調べ，登録されていればそれを返し，登

```
 1 (define intern-table (make-hash))
 2 (define intern (lambda (str)
 3   (if (hash-has-key? intern-table str)
 4     (hash-ref intern-table str)
 5     (begin (hash-set intern-table str str)
 6       (str)))))
 7 (define evolve (obj)
 8   (if (string? obj) (intern obj) obj))
 9 (reflect-copy-procedure evolve)
10 (define string "dcdbddadcaabcdbadcdbab...")
11 (define bigrams
12   (letrec ((loop (lambda (i)
13              (if (< (+ i 1) (string-length string)) ()
14                (cons (substring i (+ i 2) string)
15                      (loop (+ i 1)))))))
16     (loop 0)))
17 (define bigram-table (make-hash))
18 (letrec
19   ((loop (lambda (xs)
20      (let* ((x  (car xs))
21             (ys (cdr xs))
22             (n  (if (not (hash-has-key? bigram-table x)) 0
23                   (hash-ref bigram-table x))))
24        (begin (hash-set! bigram-table x (+ n 1))
25               (loop ys))))))
26   (loop bigrams))
27 (print bigram-table)
```

図 4.1. 実験で使用した Scheme プログラム

録されていなければ登録して返す．次の define と reflect-copy-procedure はコピー時メタプログラムの定義である．evolve 関数は引数が文字列かどうかを調べ，文字列ならば intern する．この関数をコピー時メタプログラムとして登録することで，GC のたびに全ての String インスタンスが intern され，同一内容の String インスタンスが同一のインスタンスで置き換えられる．次の define は入力データの定義である．入力データ string には 'a'，'b'，'c'，'d' の四文字からなる長さ 1024 のランダムな文字列を使用する．次の define は入力データの分割である．入力データ中の全ての連続した二文字を含むリストが名前 bigrams に束縛される．分割は再帰関数 loop によって行われる．loop は string 上のインデックスを引数にとり，それが有効なインデックスならばそのインデックスから始まる，string 上の連続した二文字を substring 関数で切り出し，次のインデックスで loop を再帰的に呼び出して得られたリストと cons する，letrec 内で定義されるローカル関数である．次の define とその次の letrec は連続した二文字の出現パターンを数える．この数え上げは loop 関数によって行われる．loop は引数に受け取ったリストの要素がそれぞれ何回出現したかを bigram-table に格納する，letrec 内で定義されるローカル関数である．最後に print 関数で結果が格納されている bigram-table を出力する．

4.2　実験結果

実験結果を図 4.2，4.3 に示す．図 4.2 は比較対象である，メタプログラム用のヒープを残してコピー GC を開始する処理系での実験結果である．それぞれのヒープサイズで実行にかかった時間を表している．また，横線は実行時エラーが発生し，実行が完了しなかったことを表している．結果としては，3.00 MB 未満のヒープでは実行時エラーが発生し，3.00 MB 以上のヒープがあれば正しく計算することが可能であり，その計算時間は 3.00 MB 以上ではほとんど変化しない．図 4.3 は提案手法である，Buffered GC を実装した処理系での実験結果を表している．各列が old，new 領域の合計サイズを表し，各行が buffer 領域のサイズを表している．処理系が使用するヒープ領域は二つの値の合計である．MB 単位である old，new 領域に対して，buffer 領域は KB 単位で十分機能することが見て取れる．old，new 領域のサイズと buffer 領域のサイズ両方が実行可能性に影響することも見て取れる．buffer 領域のサイズを同じのままに，old，new 領域のサイズを増やした時に実行できなくなる場合がある．この原因は調査できていないが，死亡したオブジェクトの少ない GC とハッシュテーブルのリハッシュなどの原因でメタプログラムが大量にオブジェクトを生産したタイミングが重なってしまったなど，処理系のバグ以外の原因も考えられる．実行時間に与える影響は old，new 領域のサイズが支配的であり，比較対象処理系でも実行可能な 3.0 MB 以上であれば実行時間は大きく変化しない．

実装した Buffered GC にはバグが存在していることが判明しており，また，実行時例外の原因がヒープ不足によるものなのか，バグによる未定義動作なのか調査できていない．Buffered GC のバグはフラッシュが頻発する少バッファ環境下で起こりやすいと考えている．バグの原因究明が進むことで，追加の実行可能なパラメタ対が発見される可能性がある．

実験結果からは，比較対象処理系では実行に 3.00 MB 必要だったヒープが，Buffered GC では約 2.00 MB のヒープで実行可能だが，しかし実行時間にオーバーヘッドを発生させることが読み取れる．また，両方の手法で実行可能なヒープサイズであればその性能差は小さく，より多くの選択肢を提供する Buffered GC は比較対象手法を改良した手法であると考えられる．

ヒープサイズ	1.50 MB	1.75 MB	2.00 MB	2.25 MB	2.50 MB	2.75 MB	3.00 MB
実行時間 (s)	—	—	—	—	—	—	0.326

図 4.2. 比較対象処理系の実験結果

		ヒープサイズ						
		1.50 MB	1.75 MB	2.00 MB	2.25 MB	2.50 MB	2.75 MB	3.00 MB
バッファサイズ	1.0 KB	—	—	—	—	—	—	0.322
	1.5 KB	—	—	—	—	0.471	0.472	0.335
	2.0 KB	—	—	0.438	0.464	0.474	0.468	0.328
	2.5 KB	—	—	—	0.446	0.461	0.461	0.337

図 4.3. Buffered GC を実装した処理系の実験結果

第 5 章

まとめと今後の課題

5.1 まとめ

本研究では現実的なメモリ効率でリフレクティブ GC を実現できる GC アルゴリズム Buffered GC を提案し，実験によってメモリ効率が向上することを示した．GC の途中でユーザープログラムを呼び出すリフレクティブ GC では，GC 中のメモリアロケーションや GC 中に消費したメモリを管理する管理機構を必要とする．GC 中のメモリアロケーションのために十分な領域を用意し，GC 中に消費したメモリを GC 中には回収しない方法も考えられるが，近年のプログラミング言語では高速なメモリアロケーションと GC を背景に大量のオブジェクトが生産，回収されるため，GC 中にメタプログラムが消費するメモリは大きく，メモリ効率が非常に悪いことが予測される．

Buffered GC を利用することで GC 中に作成されたオブジェクトはバッファ領域で一時的に管理され，一定期間を生き延びたオブジェクトだけが通常の領域に書き出されるため，ヒープ効率が向上する．

Buffered GC を実装した処理系を開発し，その上で GC リフレクションを利用するアプリケーションを実行する実験によって Buffered GC がメモリ性能を向上させることを確認した．

5.2 今後の課題

GC リフレクションの研究が発展するためには適用可能なアプリケーションを増やし，その有用性を示す必要がある．現実的な問題に GC リフレクションを適用するために，より詳細な GC の操作をサポートする方針と，より多くの GC アルゴリズム，あるいはより多くのプログラミング言語処理系をサポートする方針とが考えられる．前者の研究が進むことでユーザーはより多彩な GC の改造を実現できるようになり，また GC リフレクションを利用する事によるオーバーヘッドを削減する機会を得る．後者の研究が進むことでより多くの現実のプロジェクトで GC の改造によって機能を追加する機会が生まれる．しかし，より詳細な制御を行うためには GC アルゴリズムが限定される必要があるため，この両方針の両立は難しい．両立のためには様々な GC アルゴリズムの実装をを抽象インターフェースの影に隠し，かつ

GC リフレクションとして十分有益であるような GC を表現するフレームワークを開発し，このフレームワーク上のインターフェースををリフレクションとして公開するアプローチが有効だろう．

また，Buffered GC を利用した GC リフレクションでは大きな実行時間オーバーヘッドが発生する．ユーザープログラムをなるべくブロックせずに GC を行う，ノンブロッキング GC で使われる技術を応用することでオーバーヘッドを削減できる可能性がある．

発表文献と研究活動

(1) 山崎 徹郎, 佐藤 芳樹, 千葉 滋. 大規模コードクローン検出のための包含文字列を対象とした省メモリ化プログラム変換の予備的な性能評価. 情報処理学会 第 78 回全国大会 (IPSJ 2016), 慶應義塾大学 矢上キャンパス, 2016 年 3 月.

(2) 山崎 徹郎, 佐藤 芳樹, 千葉 滋. 包含文字列の内部 char 配列を共有化する拡張文字列を部分的に利用した時の性能を調べる実験. PPL 2016, 2016.3.7-9 (ポスター)

参考文献

[1] Michihiro Horie, Kazunori Ogata, Kiyokuni Kawachiya, and Tamiya Onodera. String deduplication for java-based middleware in virtualized environments. *SIGPLAN Not.*, Vol. 49, No. 7, pp. 177–188, March 2014.

[2] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic code evolution for java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pp. 10–19, New York, NY, USA, 2010. ACM.

[3] Tal Cohen and Joseph (Yossi) Gil. Three approaches to object evolution. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pp. 57–66, New York, NY, USA, 2009. ACM.

[4] Richard L. Hudson, J. E Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. Technical report, Amherst, MA, USA, 1991.

[5] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? high performance garbage collection in java with mmtk. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pp. 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

[6] Brian Cantwell Smith. Reflection and semantics in lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pp. 23–35, New York, NY, USA, 1984. ACM.

[7] Charlotte Herzeel, Pascal Costanza, and Theo D'Hondt. Self-sustaining systems. chapter Reflection for the Masses, pp. 87–122. Springer-Verlag, Berlin, Heidelberg, 2008.

[8] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, Vol. 18, No. 9, pp. 807–820, September 1988.

[9] Robert R. Fenichel and Jerome C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Commun. ACM*, Vol. 12, No. 11, pp. 611–612, November 1969.

[10] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, Vol. 13, No. 11, pp. 677–678, November 1970.

[11] George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, Vol. 3, No. 12, pp. 655–657, December 1960.

[12] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, Vol. 26, No. 6, pp. 419–429, June 1983.

[13] 肥後芳樹, 楠本真二. プログラム依存グラフを用いたコードクローン検出法の改善と評価. 情報処理学会論文誌ジャーナル, Vol. 51, No. 12, pp. 2149–2168, 2010.

[14] Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the specjvm98 java benchmark. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, pp. 92–115, London, UK, UK, 1999. Springer-Verlag.

謝辞

本研究を進めるにあたり，支えてくださった皆様に感謝いたします．特に，指導教員である千葉教授にはいくつもの重要なご指導，ご助言を頂きました．大変感謝しております．佐藤芳樹氏にもまた研究の方針や論文の執筆について多くご指導頂きました．厚くお礼を申し上げます．市川 和央氏には研究に関すること，関しないこと，様々な相談に乗って頂きました．大変感謝しております．折笠 雄太郎氏には実装が行き詰まっていた時，問題の原因であるバグを発見していただきました．大変感謝しております．また，友人の助力のお陰で私の研究生活は非常に楽しいものとなりました．常々感謝しております．