情報処理学会論文誌　プログラミング　Vol.13　No.4　1–20 (Oct. 2020)

実行時リージョン解析による動的言語Rubyのメモリ割付け最適化

齋地 崇大[1,a)]　前田 敦司[2,b)]

受付日 2020年4月10日, 採録日 2020年7月22日

概要：リージョンによるメモリ管理は，メモリ空間をリージョンと呼ばれる単位で分割し，リージョン単位でLIFO順にメモリの割付けと解放を行う．プログラム中のメモリを必要とするオブジェクトは，それぞれの生存期間に対応したリージョンにメモリが割り付けられる．オブジェクトとリージョンの対応付けは，プログラムの構造を静的に解析することによって決定できる．実行の前にオブジェクトの生存期間を決定するため，ガベージコレクションによる実行時オーバヘッドの削減が期待できる．Ruby，Python，そしてJavaScriptといった動的言語は，実行時まで得られない動的な情報がプログラムに含まれるため，不要となったオブジェクトの割り付けられたリージョンを速やかに解放できる精度の高いオブジェクトとリージョンの対応付けを静的な解析で決定するのは困難である．本研究では，精度の高いリージョンによるメモリ管理を動的言語へ適用する手法として，実行時リージョン解析を提案する．実行時まで決定しない情報を用いてリージョンを解析するため，静的な解析と比較して精度の高いリージョン対応付けを決定できる．提案手法の性能を計測するため，プログラミング言語Rubyの処理系へ実行時リージョン解析機構を実装し評価を行った．いくつかのケースでは，ガベージコレクションの頻度や停止時間が削減され，実行時間の短縮が確認された．

キーワード：動的言語，メモリ管理，リージョン解析，Ruby，メモリ割付け最適化

Optimizing Memory Allocation of Ruby Implementation by Dynamic Region Analysis

Takamasa Saichi[1,a)]　Atusi Maeda[2,b)]

Received: April 10, 2020, Accepted: July 22, 2020

Abstract: Region-based memory management divides the memory space by the unit called the region and allocates and releases the memory by the region in LIFO order. Objects requiring memory in the program are allocated to regions corresponding to their lifetimes. The lifetime of an object is determined before execution, so it is expected that the runtime overhead by garbage collection is reduced. Dynamic languages such as Ruby, Python, and JavaScript contain dynamic information that cannot be obtained until runtime. Therefore it is difficult to determine the highly accurate object and region mappings which can quickly release allocated region of objects that are no longer needed by static analysis. In this paper, we propose dynamic region analysis as a method to apply region-based memory management to dynamic languages. By analyzing at runtime, regions can be analyzed using information which is not determined until runtime, so that accurate region mapping can be determined higher than the static analysis. In order to measure the performance of our method, the dynamic region analysis mechanism was introduced into the Ruby programming language, and the evaluation was carried out. In some evaluations, the frequency and pause times of garbage collection itself are reduced and have seen a reduction in execution time.

Keywords: dynamic language, memory management, region analysis, Ruby, optimizing memory allocation

[1] クックパッド株式会社
Cookpad Inc., Shibuya, Tokyo 150–6012, Japan
[2] 筑波大学システム情報系
Faculty of Engineering, Information and Systems, University of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan

[a)] saichi@ialab.cs.tsukuba.ac.jp
[b)] maeda@cs.tsukuba.ac.jp

© 2020 Information Processing Society of Japan

1. はじめに

Tofte らによって提案されたリージョンによるメモリ管理 [1], [2] は，プログラムの実行に必要なメモリ空間をリージョンと呼ばれる単位で管理する方法である．リージョンはスタック構造のように LIFO（Last In First Out）に従って追加と削除がされる．プログラム中に出現するオブジェクトは，自身よりも生存期間が長いリージョンへメモリが割り付けられる．対して，メモリの解放はリージョン単位で行われるため，個々のオブジェクトのゴミ判定のためのガベージコレクションは不要である．メモリ割付け後に間もなくゴミと判定されるオブジェクトが多いようなプログラムにおいては，ガベージコレクションと比較して実行時オーバヘッドの削減とパフォーマンスの向上が期待できる．ただし，パフォーマンスを向上させるためには，オブジェクトの生存期間に最も近い，つまり，不要となったオブジェクトのメモリを速やかに解放可能な精度の高いオブジェクトとリージョンの対応付けを決定する必要がある．そうしたリージョンを決定するアルゴリズムはリージョン推論 [3]，もしくはリージョン解析 [4] と呼ばれる．前者は既存の型システム上の型推論アルゴリズムを拡張することで実現され，後者はポインタ解析といった静的解析に基づいてオブジェクトとリージョンの対応付けを決定する．

Ruby [5]，Python [6]，そして JavaScript [7] といった，実行時まで決定することのない情報を多く含む動的なプログラム言語が広く使用されている．これらの言語はメタプログラミングや，型による制約を受けないプログラムを柔軟に記述できる．その反面，実行や入力に依存する動的な情報が含まれるため，型検査や精度の高い解析の適用が困難である．加えて，柔軟性を保ちながらの実行が要求されるため，プログラムのメモリ管理はガベージコレクションなどの，実行時の状態を動的に解釈してオブジェクトの生存期間を判定する手法が採用されていることが多い．しかし，プログラムによっては，LIFO に従った割付けを適用できるオブジェクトを含む可能性がある．その場合，動的なメモリ管理機構での生存期間の判定処理はオーバヘッドとなり得る．仮に，こうしたプログラムのメモリ管理方法としてリージョンによるメモリ管理を適用することができれば，実行時メモリ管理機構の処理を削減可能なため，パフォーマンスの向上につなげることができる．しかし，前述した型検査や精度の高い解析の適用が困難であるという性質から，リージョン推論やリージョン解析といった手法を適用した場合でも，精度の高い対応付けの決定は難しい．

本研究では，リージョンによるメモリ管理を動的言語へ適用する手法として，実行時リージョン解析を提案する．実行時に解析することにより，動的に定義されるメソッドといった，実行時まで決定しない情報を用いることが可能であるため，静的な解析と比較して精度の高いオブジェクトとリージョンの対応付けを決定できる．また，解析で得られた結果を用いて実行時コンパイルやコード変換を実現することで，動的言語であったとしてもリージョンによるメモリ管理が適用可能になる．ただし，精度の高い解析が実現できたとしても，動的言語へのリージョンによるメモリ管理の適用にはいくつかの課題が残る．まず，既存のメモリ管理機構との互換性を保たなければならない．ガベージコレクションの処理中に，リージョンに用いられるメモリ領域を考慮しなければならないため，言語処理系の実装によってはリードバリアの挿入やアルゴリズムの変更が必要になる．また，リージョンの仮定する参照制約を破る可能性がある言語機能を考慮しなければならない．リージョンの生存期間を過ぎた場合でも，リージョン内のオブジェクトを参照できる機能が動的言語によって提供されている場合，それらの機能を使用することによって，不正なメモリアクセスを招いてしまう可能性があるため，解析時に検出して最適化そのものを中止するか，実行中に該当する機能が使用されたことを検知して，リージョン内のすべてのオブジェクトの管理を既存のメモリ管理機構へ移譲しなければならない．これらの課題を含めて，実行時リージョン解析の適用可能性について議論するため，本研究ではプログラミング言語 Ruby の処理系へ実行時リージョン解析と実行時コンパイル機能を実装し，いくつかのケースで評価を行った．マイクロベンチマークにおいては，実行時リージョン解析はガベージコレクションの頻度を削減し，有効にしない場合と比較してパフォーマンスの向上を確認した．また，CRuby の提供するベンチマークでは，保守的な解析や最適化を解除する仕組みに課題が残ることも確認した．

2 章では，背景となるリージョンによるメモリ管理やその関連研究について述べる．3 章では，実装の対象としたプログラム言語 Ruby とその処理系について，提案手法の実装に関連する機能とその性質について説明する．4 章で実行時リージョン解析の概要と Ruby 処理系への実装について述べた後，5 章で実装の評価結果を元に実行時リージョン解析の性能を議論し，6 章と 7 章で関連する研究を取り上げつつ課題や考察について述べる．

2. リージョンによるメモリ管理

本論文では，実行時リージョン解析の戦略と実装について扱う．この章では，研究の背景として，リージョンによるメモリ管理の概要について述べる．また，リージョンを静的に決定する手法であるリージョン推論とリージョン解析について既存研究を例に解説し，最後にリージョンによるメモリ管理と他のメモリ管理手法を比較してそれぞれの特徴について述べる．

2.1 メモリ領域の分割と管理

リージョンによるメモリ管理は，プログラムの実行にお

© 2020 Information Processing Society of Japan

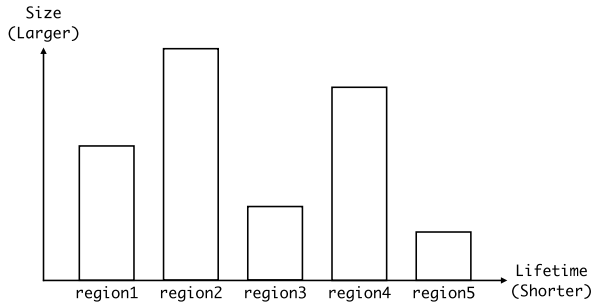


図 1　リージョンのメモリ領域
Fig. 1　Memory area of the region.

けるメモリ管理手法の 1 つであり，メモリ空間を**リージョン（region）**と呼ばれる単位で管理する．リージョンはスタック構造のように LIFO（Last In First Out）に従って追加と削除が行われる．スコープやコードブロックに対応して追加と削除がされるなど，実装によってタイミングは様々である．

プログラム中に出現するオブジェクトのメモリ割付けは，自身よりも生存期間が長い特定のリージョンに対して行われる．メモリの解放はリージョン単位で行われるため，個々のオブジェクトのゴミ判定のためのガベージコレクションは不要である．メモリ割付け後に間もなくゴミと判定されるオブジェクトのメモリが多く割り付けられるようなプログラムでは，ガベージコレクションと比較して実行時オーバヘッドの削減とパフォーマンスの向上が期待できる．ただしパフォーマンスを向上させるためには，オブジェクトの生存期間に最も近い，つまり，不要となったオブジェクトのメモリを速やかに解放可能な精度の高いリージョンを決定する必要がある．

図 1 はリージョン間のサイズと生存期間の関係を表している．リージョンに新しくオブジェクトが割り付けられると，そのリージョンはより大きなサイズに成長する．リージョンが削除されるまでメモリは解放されないため，サイズが小さくなることはない．図の状態に新たなリージョンを追加する場合は，`region6` が `region5` の右側，つまり，より生存期間の短いリージョンとして追加される．図の状態からリージョンの削除を行う場合は，LIFO に従い `region5` から順に削除される．

リージョンに割り付けられたオブジェクトの参照には制約が生じる．`region1` が `region2` より生存期間が長いとする．このとき，`region1` 内に割り付けられたオブジェクトから `region2` 内に割り付けられたオブジェクトへの参照は，`region2` が削除されるまでの間でのみ有効である．`region2` が削除されてもなお `region1` 内に参照が残っている場合，その参照はすでに解放されている領域を指してしまうため，ぶら下がりポインタが発生する．ぶら下がりポインタを許容できない処理系であれば，生存期間の長いリージョンから短いリージョンへの参照を防ぐ必要がある．本提案において，この参照の方向に関する制約を**リージョンの参照制約**として定義する．同様に，ヒープ領域内のオブジェクトといった，生存期間がリージョンとは別で管理されている箇所からの参照は最も生存期間が長いリージョンとして考える必要がある．

2.2　スタック割付け・ヒープ割付けとの比較

この節では，スタック領域へオブジェクトのメモリを割り付ける方法，およびヒープ領域へオブジェクトのメモリを割り付ける方法とリージョンによるメモリ管理を比較する．スタック領域へオブジェクトのメモリを割り付ける方法を**スタック割付け（stack allocation）**と表記し，その意味は，局所変数の管理に使用されるような，スコープに対応して確保されるメモリ領域へオブジェクトのメモリを割り付ける方法である．ヒープ領域へオブジェクトのメモリを割り付ける方法を**ヒープ割付け（heap allocation）**と表記し，ガベージコレクションによって管理されるような，ヒープへのメモリ割付けを意味する．

スタック割付けは，最も上のスタックフレームに対してのみメモリ割付けが可能であるのに対し，リージョン割付けは生存している任意のリージョンフレームにメモリ割付けが可能である．これは，スタック割付けが持つ LIFO の順序がスタックフレームの割付けと解放に適用されるのに対して，リージョン割付けが持つ LIFO の順序はリージョンの生存期間のみに適用されるためである．そのため，リージョン割付けはスタック割付けよりもメモリ割付けの点において制約が緩い．

スタック割付けとリージョン割付けは，どちらも割付けられたオブジェクト間に参照の制約が発生する．生存期間の短いフレーム内に割り付けられたオブジェクトから生存期間の長いフレーム内への参照は許容されるが，その逆の参照はぶら下がりポインタの発生を促してしまうため，そもそも参照の存在を許さないか，フレームのメモリを解放するごとにぶら下がりポインタの発生を検出して修正必要がある．

ヒープ割付けは割付け順序や生存期間に関しての制約がない．しかしながら，割り付けられたメモリの解放を行うための管理が必要である．多くのプログラム言語処理系は，ヒープメモリの管理機構として，ガベージコレクションの手法を採用している．これは，ヒープ割付けが制約を持たないという点を利用しつつ，オブジェクトの生存期間を管理するために有用な戦略である．ただし，実行時に生存期間を計算するオーバヘッドが発生するため，いかなるプログラムにも効率の良い方法であるとはいえない．

2.3　リージョン推論

静的にオブジェクトとリージョンの対応付けを決定する手法として，型システム上の型推論を拡張することで実現

© 2020 Information Processing Society of Japan

するものをリージョン推論（region inference）と呼ばれる．リージョン推論は Tofte らによって提案され，SML 処理系である MLKit などの言語処理系に採用されている．

2.3.1　ML に対するリージョン推論アルゴリズム

Tofte の提案した ML に対するリージョン推論は，リージョンが明示されていない ML プログラムを入力として，リージョン推論の結果を用いて，ML プログラムをリージョン割付けへ適応した形式へ変換する．たとえば，以下の ML プログラム

```
let x = 2 + 3 in
  (lambda y.(x, y))
end
5
```

は次のように変換される．

```
letregion r5 in
  letregion r1, r2 in
    let x = (2 at r1 + 3 at r2) at r3 in
      (lambda y. (x, y) at r4) at r5
    end
  end
  5 at r6
end
```

ここで，rN はリージョンを識別するリージョン変数（region variable）と呼ばれる．たとえば，`letregion r1 in` *exp* `end` は，式 *exp* の評価において `r1` という変数に新たなリージョンを束縛するという意味を持ち，*exp* `at r1` は，*exp* の評価結果に必要なメモリをリージョン変数 `r1` の指し示すリージョンに割り付けるという意味を持つ．

関数は，引数のオブジェクトが割り付けられているリージョン変数と，返り値のメモリを割り付けるリージョン変数を暗黙的に受け取る．実行時にわたされるそれらのリージョン変数を使用して，メモリを割り付けるリージョンの決定をパラメトリックに行うことができる．これをリージョン多相（region polymorphism）といい，この仕組みを使って複数の手続きにわたって生存するオブジェクトのメモリを管理する．リージョン多相については 2.5 節でさらに具体例を述べる．

2.3.2　Scheme に対するリージョン推論アルゴリズム

Nagata らは動的型付け言語のためのリージョン推論に基づくメモリ管理を提案した [8]．プログラミング言語 Scheme に対して，Cartwright らの Soft typing [9] と，Tofte らのリージョン推論を組み合わせてリージョンによるメモリ管理を実現をしている．しかしながら，動的に決定する情報は最も保守的に推論されるため，プログラムによっては精度の高いリージョンを決定することが困難であるほか，Scheme の言語機能が限定的であるため，実在する動的言語への適用という観点では課題が残る．

2.4　リージョン解析

型推論を拡張するリージョン推論に対して，プログラムの構造を解析することでオブジェクトとリージョンの対応付けを決定する方法をリージョン解析（region analysis）という．リージョン解析は，ポインタ解析やデータフロー解析によって，オブジェクトの生存期間を計算することで実現される．

2.4.1　Java に対するリージョン解析アルゴリズム

Cherem らは Java に対してリージョンを解析するアルゴリズムを提案した [4]．解析では，各メソッドについて，region points-to graph と呼ばれる参照関係のグラフを計算する．region points-to graph は，メソッド内のオブジェクトがどのリージョン変数に紐づくかを決定する．その後，メソッドの呼び出し関係間で，リージョン変数の単一化を行い，リージョン変数ごとに寿命を判定する．

解析結果から Java のプログラムをリージョン割付けに対応した形に変換する．変換の例を以下に示す．

```
public void valueAt(Complex x) {
  create r7, r8;
  List rev = coeffs.reverse<r7, r8>();

  create r9;
  Complex sum = new Complex(0, 0) in r9;

  create r10;
  Iterator it = rev.iterator<r10>();
  while (! it.empty ()) {
    Complex coeff = (Complex)it.next();
    create r11;
    Complex tmp = sum.mul<r11>(x);
    remove r9;
    create r9;
    sum = tmp.plus <r9>(coeff);
    remove r11;
  }
  remove r7, r8, r10;
  System.out.println (sum.re);
  remove r9;
}
```

`create r1` の記述で `r1` というリージョン変数に新たなリージョンを束縛する．オブジェクトの生成時に付与される `in r1` の記述でオブジェクトのメモリを `r1` に割り付ける．リージョンは `remove r1` で削除される．例のように，Cherem らの提案では，ループの反復ごとにリージョンの追加と削除が可能である．

© 2020 Information Processing Society of Japan

Java プログラム中には動的なクラスローディングといった，実行時の情報に依存する仕様が存在する．Cherem らは，これらの仕様に対して保守的な解析を行うことにより，リージョンの精度を下げることで互換性を保っている．

2.5　リージョン多相

リージョンによるメモリ管理の特性として，オブジェクトのメモリ割付け先のリージョンは LIFO 順で最も新しいリージョンに限らず，削除されていない任意のリージョンに割付け可能であることを 2.2 節で述べた．この特性はリージョン多相を実現するためにも利用される．この節では，ML のプログラム例を用いてリージョン多相の振舞いについて述べる．

リージョンによるメモリ管理が適用された関数を考える：

```
(lambda x y. (x :: y))
```

この関数のリージョン注釈付きの型は

$$((int\ \texttt{at rA}) \times (int\ list\ \texttt{at rB})) \to (int\ list\ \texttt{at rB})$$

であり，これは「rA リージョン上に割り付けられた整数と rB リージョン上に割り付けられた整数のリストを受け取り，rB リージョン上に割り付けられた整数のリストを返す」ことを意味する．このとき，rA と rB はリージョン多相パラメータであり，実行時に仮引数に代入される値のリージョンに置き換えられて解釈される．

リージョン多相パラメータは暗黙的に生存期間の大小関係が保証されていることが求められる．上記の例では，rB は rA への参照を持ちうるため，前後のプログラムによっては rB の生存期間は rA よりも長くなければならない．

3.　動的言語 Ruby

本研究では，実行時リージョン解析の実装対象として，プログラム言語の Ruby の標準実装である CRuby を選択した．提案内容の説明にあたって，この章ではその予備知識となる Ruby の特性や CRuby の実装について述べる．

3.1　CRuby の概要

CRuby のプログラムは抽象構文木をそのまま実行するのではなく，YARV と呼ばれる VM 上で動作するバイトコードにコンパイルされてから実行される [10]．

VM の実行に使用されるスタックマシンのコンテキストは，Ruby 上のスレッドごとに用意される．スレッドは **GIL（Global Interpreter Lock）**によって同時にたかだか 1 スレッドしか同時に実行されない．

3.2　静的解析の困難性

Ruby はメタプログラミングのしやすさや柔軟な記述を可能にするため，動的に様々な操作を実現できる機能を標準ライブラリとして提供している．この Ruby の強力さの反面，以下のように静的な解析を困難にする場合がある．

3.2.1　メソッドの動的なディスパッチ

Ruby のメソッドは呼び出された時点で，レシーバオブジェクトや実行環境から呼び出されるメソッドが動的に決定される．実行時に呼び出されるメソッドをプログラムの記述から静的に決定することは一般に困難である．

3.2.2　Module#define_method

Module#define_method[*1] メソッドを実行することでクラスやモジュールにメソッドを追加することができる．その引数はメソッド名とブロックであり，他のブロックを引数にとるようなメソッドと同じように呼び出すことができる．たとえば，以下のようなクラス Greeter を考える．

```
1  class Greeter
2    def initialize(name)
3      @name = name
4    end
5
6    define_method("hello") do
7      "Hello #{@name}!"
8    end
9  end
10
11 g = Greeter.new("ruby")
12 g.hello #=> "Hello ruby!"
```

12 行目で呼び出す Greeter#hello メソッドは Greeter クラスのロード時に Module#define_method メソッドを呼び出して定義されたメソッドである．このように，Module#define_method を介して定義されたメソッドは実行中に定義が追加され，通常のメソッドと同じように振る舞う．Module#define_method はキーワードではなく一般的なメソッドと同様の扱いであるため，Ruby のプログラムを静的に解析して，定義されているメソッドの集合を特定することは困難である．

3.2.3　Kernel.#eval

Kernel.#eval メソッドは，引数として受け取った文字列を Ruby プログラムとして解釈し，現在の実行環境でプログラムを実行する．たとえば，以下のようなプログラムを考える．

```
1 unused = nil
2 eval("unused = 100")
3 unused # => 100
```

[*1] Ruby 公式のリファレンスマニュアル（https://docs.ruby-lang.org/）の記法に従い，クラス Klass のクラスメソッド method は Klass.method，クラス Klass のインスタンスメソッド method は Klass#method モジュール Mod のモジュール関数 func は Mod.#func と表記する．

© 2020 Information Processing Society of Japan

1 行目で unused 変数に nil が代入されているが，直後の Kernel.#eval メソッドを呼び出した後に評価すると，値が 100 に書き換えられている．もし，Kernel.#eval メソッドの実行を静的に解析するとなれば，引数として与えられている文字列のパースを行い，妥当な Ruby プログラムだと判定したうえでそのプログラムを解析しなければならない．

Kernel.#eval メソッドも他のメソッドと同様の扱いであるため，Ruby のプログラムから静的に変数やメソッドの定義を決定することを困難にする．

このほかにも動的に決定する情報を持つプログラムの例はあげられるが，ここでは，提案手法の解説において言及が必要な以上 2 つの機能について述べた．

3.3　メモリ管理

3.3.1　ガベージコレクション

CRuby のオブジェクトに割り当てられるメモリは，ガベージコレクションによって管理されている．CRuby のガベージコレクションは以下のような特徴を持つ：

- ガベージコレクションの実装はマークアンドスイープ
- マークアンドスイープはインクリメンタルに実行
- 保守的なガベージコレクション
- 世代別ガベージコレクション

CRuby の世代別ガベージコレクションの実装は eden と tomb と呼ばれる 2 つの世代を持つ．eden は若い世代のヒープ領域を管理し，対して tomb は古い世代を管理する．

3.3.2　オブジェクトのメモリとヒープ領域

オブジェクトのメモリはガベージコレクションによって管理されるフリーリストから割り当てられ，RVALUE と呼ばれる構造体にマッピングされる．RVALUE はオブジェクトの型や変数テーブル，そしてオブジェクト自体の値表現を含む合計 40 バイトの固定長のサイズで表現される．もしオブジェクトが不要と判断された場合は RVALUE 分のメモリ領域をガベージコレクションによって管理されるフリーリストへ返却する．

RVALUE が配置されるヒープ領域は**ヒープページ（heap page）**というチャンク単位で管理されている．1 つのページには複数の RVALUE が含まれており，ヒープページのヘッダには，それらを管理するためのフラグやフリーリストの先頭アドレス，また，ガベージコレクションのマークに使用されるビットマップが用意されている．

ヒープの構造を**図 2** に示す．ガベージコレクションの世代ごとに rb_heap_t 構造体が用意され，この構造体がヒープページのリストを管理する．RVALUE のアドレスからビット演算でヒープページのヘッダを参照することができるよう，ヒープページは 16 KB でアラインメントされている．

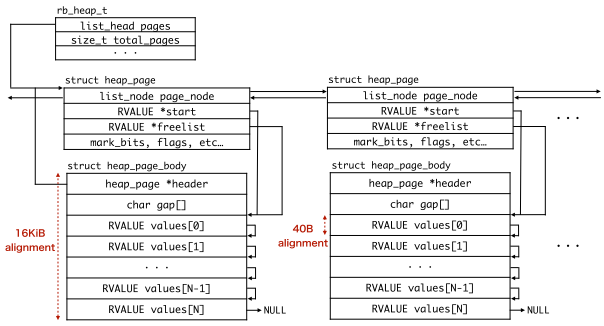


図 2　ヒープページの構造
Fig. 2　structure of the heap page.

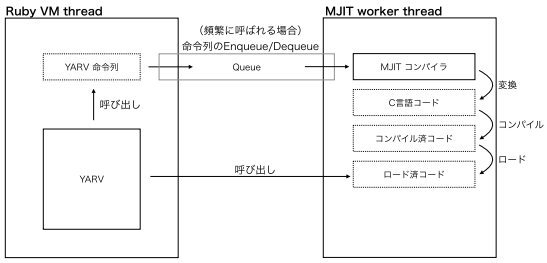


図 3　MJIT の構造
Fig. 3　structure of the MJIT.

3.4　実行時コンパイル機能 MJIT

CRuby は実行時コンパイルによって，YARV の命令列をネイティブコードへコンパイルする MJIT という JIT コンパイラが実装されている．MJIT は，VM の実行中，頻繁に実行されるメソッドを検出し，その命令列のコンパイルをスケジュールする．コンパイルが成功し，メソッドの内容をメモリにロードできた場合，次回以降のそのメソッドの実行は JIT コンパイルされたネイティブコードが呼び出される．

MJIT のコンパイルは VM のスレッドとは別スレッドで並行に行われる．コンパイルをスケジュールするタイミング以外で VM のスレッドが停止することはない．

MJIT でネイティブコードにコンパイルされたメソッドがいくつかの条件にあてはまる場合，スタックポインタやプログラムカウンタをそのままに命令列の実行を VM のインタプリタへ戻す可能性がある．この最適化状態から非最適化状態への切り替えは**最適化解除（deoptimization）**と呼ばれる [11]．たとえば，JIT コンパイラが前提としてしている事前条件の検証に失敗した場合や，Kernel.#set_trace_func が有効になったときである．事前条件の検証失敗は，組み込み最適化命令の実行前などで起こりうる．

4.　実行時リージョン解析

動的言語において，型推論を提供している処理系は多くない．もし提供されていたとしても，プログラマによる型注釈を必要としたり，具体的な型が求まらない場合は最

© 2020 Information Processing Society of Japan

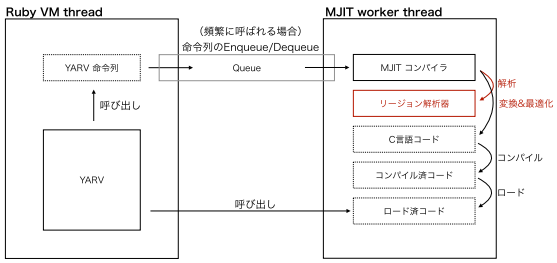


図 4　MJIT を利用した実行時リージョン解析の構造
Fig. 4　The structure of dynamic region analysis using MJIT.

も一般的な型を示す場合があるため，精度の高い結果は得られないことが多い．静的解析に関しても同様である．3.2 節で説明したような動的な情報の収集を静的解析で実現するためには，プログラムの実行と同程度の手順を踏む必要がある．こういった静的なアプローチを基礎としてリージョンの決定を試みようとしても，ぶら下がりポインタの発生を防ぐために，精度の低い保守的なリージョンとオブジェクトの対応付けをせざるを得ない．保守的なリージョンへのメモリ割付けは，不要になってから解放するまで期間が長くなるため，空間計算量の悪化を招く可能性がある．

この章では，前章までに説明した背景をもとに，本研究で提案する実行時リージョン解析について述べる．リージョンの解析に必要な情報が揃う実行時までリージョン解析を遅延させることで，静的なリージョン解析と比べて精度の高いリージョンを決定する．本提案の実現可能性について議論するため，Ruby の特性との互換性や，CRuby 処理系への実装についても述べる．

4.1　実行時コンパイルを契機としたリージョン解析

リージョン解析を実行時に行うといっても，一度にプログラム全体を解析するわけではない．動的言語は実行が進むごとにプログラムの構造が変化する可能性がある．その度にプログラム全体を解析していては，実行時のパフォーマンスに影響を及ぼしてしまう．特定のコードブロックに対して部分的にリージョン解析が行えればよいため，そうした目的で使用される実行時コンパイラの機能に着目した．

CRuby 処理系に実装されている実行時コンパイル機構である MJIT を拡張してリージョン解析を行うことを考える．MJIT は 3.4 節で述べたように，メソッド単位でコンパイルを行う．コンパイルがスケジュールされたとき，取得できる情報にはメソッドの命令列が含まれており，これは `Module#define_method` や `Kernel.#eval` を使った場合でも同様である．つまり，実行前に定義されているいないにかかわらず，YARV 上で動作する命令列を得ることができる．

MJIT のコンパイル手順にリージョン解析を組み込んだ場合の解析機構の全体図を**図 4** に示す．MJIT にコンパイルがスケジュールされるタイミングは，YARV がメソッド呼び出しを行う直前である．本提案で使用するバージョンに実装されている MJIT は，メソッドの中身を解析せずに直接命令列を C 言語のソースコードに変換する．本提案では，変換の直前にリージョン解析の手順を挿入し，C 言語への変換時にリージョンの情報を用いて割付け部分の差し替えを行う．

4.1.1　実行時コンパイル対象外なメソッドとの共存

MJIT では，C 言語で実装されている Ruby のメソッドはコンパイルされない．また，頻繁に実行されないようなメソッドや，MJIT が対応できないメソッドに関しては，コンパイル対象とならない．そのため，リージョン割付けを行うメソッドと，通常のヒープ割付けを行うメソッドが同時に存在することになる．一度にプログラム全体を解析することができる場合は，リージョン割付けに対応できないメソッドの存在を検出し，それらのメソッド間で行われるオブジェクトのやり取りでは，不正なメモリアクセスが生じないようにリージョン割付けを実現できる．本提案では，部分的にリージョン解析を行うため，どのオブジェクトがどのメソッドにわたされるかを厳密に検査することはできない．その場合，すでに解放されたリージョンに割付けられていたオブジェクトへの参照を持つぶら下がりポインタの発生を許してしまう可能性がある．本提案の実行時リージョン解析は，そのようなぶら下がりポインタが生じうると判断した場合，保守的な解析や，4.5.1 項で述べる最適化解除の仕組みによって，実行時エラーの発生を未然に防いでいる．

4.2　リージョン解析アルゴリズム

本提案のリージョン解析は，メソッドごとの YARV の命令列に対して行われる．プログラム全体を一度に解析するわけではないため，メソッド呼び出しや組み込み最適化命令の解析は保守的に行われる．本節では，YARV のバイトコードに対するリージョン解析アルゴリズムと，その解析結果を用いたメモリ割付け最適化の手順について述べる．

4.2.1　抽象解釈によるアプローチ

本提案のリージョン解析は YARV の命令列に対する**抽象解釈** [12] に基づく．抽象解釈はプログラムに対して静的な解析を行う手法の 1 つである．解釈の対象となる命令やプログラムに対して，何らか意味のうえで解釈を行うことによって，その意味に関する情報を得ることを目的とした解析手法として知られている．

本提案のリージョン解析では，MJIT のコンパイルの対象となるメソッド単位で抽象解釈を実行する．オブジェクトの生存期間に関する情報として，メソッド中に出現するオブジェクトにリージョン変数を割り当てる．VM スタックや変数束縛環境の操作によるリージョン変数への影響を各命令の規則として与え，メソッド中に発生しうるすべて

© 2020 Information Processing Society of Japan

のオブジェクトについてそのリージョンを決定する．

YARV の命令列は抽象解釈可能な形式ではあるが，Ruby プログラムからのコンパイル方法や，各命令の操作が形式的に定義されていないため，本提案では解析の正しさの証明は行わない．代わりに，CRuby 処理系の提案するテストやベンチマークとの互換性を保つことで，抽象解釈による解析の実装に問題がないことを確認する．

アルゴリズムの説明にあたって，抽象解釈に必要な構造を定義する．リージョン変数は `r` の接頭辞が付く連続した番号で表される．ヒープ領域は最も生存期間の長いリージョンとして扱い，$\mathtt{r}_{heap}$ と表記する．また，解析の都合上，解析中のメソッドのレシーバが割り付けられているリージョン変数を他のリージョン変数と区別し，$\mathtt{r}_{self}$ と表記する．リージョン変数全体の集合 $RegVar$ は次のように定義される．

$$RegVar = \{\mathtt{r}_{heap},\ \mathtt{r}_{self},\ \mathtt{r}_1,\ \mathtt{r}_2,\ \ldots\}$$

VM スタック S は，オブジェクトを LIFO で管理する VM スタックを表現する．環境 Γ は変数束縛環境を表し，次に示す種類の変数ごとに定義が存在する．

- ローカル変数束縛環境 Γ_{local}
- グローバル変数束縛環境 Γ_{global}
- クラス変数束縛環境 Γ_{class}
- インスタンス変数束縛環境 $\Gamma_{instance}$
- スペシャル変数束縛環境 $\Gamma_{special}$
- 定数変数束縛環境 Γ_{const}
- ブロックパラメータ変数束縛環境 $\Gamma_{blockparam}$

解析中，異なるリージョン変数が同一であることを**単一化（unification）**によって表現する．単一化はその結果としてリージョン変数間の**代入（substitution）**の集合を得る．代入は $RegVar \Rightarrow RegVar$ の形式をしており，リージョン変数の置き換えを意味する．単一化のアルゴリズムを Algorithm 1 に，リージョン変数への代入適用アルゴリズムを Algorithm 2 に示す．

解析は先頭の番地である `pc = 0` に配置されている命令から開始される．キューに追加された番地とスタックおよび変数束縛環境 3 つ組を取り出し，幅優先で評価される．評価後，`pc` をインクリメントして次の命令の番地に更新し，更新されたスタックと変数束縛環境とともにキューに追加する．ジャンプ命令の場合は，`pc` をジャンプ先の番地に置き換えてキューに追加する．分岐命令の場合は，その条件が成立するかが決定できないため，両方の分岐先の番地をキューに追加する．`leave` もしくは `throw` 命令に達した場合，命令列の実行はそこで終了する．

命令列の構造によってはループを含む可能性がある．ループによって同じ番地の命令が再び評価されるとき，評価後のスタックおよび変数束縛環境が前回の評価時から変化がなければ，その命令で解析は終了する．もし変化が生じる場合，スタックの各要素と変数束縛環境の各束縛について，前回の評価後のオブジェクトと今回の評価後のオブジェクトで単一化を行う．

Algorithm 1 unification

```
function UNIFY(r_N, r_M)
    if N = M then
        return ∅
    else if N = heap then
        return {r_M ⇒ r_heap}
    else if M = heap then
        return {r_N ⇒ r_heap}
    else if N = self then
        return {r_M ⇒ r_self}
    else if M = self then
        return {r_N ⇒ r_self}
    else if M < N then
        return {r_N ⇒ r_M}
    else
        return {r_M ⇒ r_N}
    end if
end function
```

Algorithm 2 substituion

```
function APPLYSUBSTSET(r_N, SubstSet)
    do
        applied ← false
        if SubstSet includes r_N ⇒ r_K then
            applied ← true
            N ← K
        end if
    while applied = true
end function
```

変数束縛環境は，メソッドを呼び出すごとに束縛関係が異なる可能性がある．レシーバや引数，ローカル変数，ブロックパラメータ変数を除いた，スコープ外で束縛関係を更新可能な環境から参照されるオブジェクトは，すべてヒープ領域に割付けされていると見なす．たとえば，評価中に `getglobal` 命令で取得されるオブジェクトは，グローバル変数束縛環境の状態にかかわらず，$\mathtt{r}_{heap}$ のリージョンに対応すると評価される．

代表的な命令の評価規則を**図 5** に示す．説明のため，命令のオペランドは抽象実行に必要な項目を簡略化し，規則によって影響のともなわない変数環境は省略している．それぞれの命令は，抽象解釈の各ステップにおいて，VM スタック S と環境 Γ，そして代入 $Subst$ を更新する．なお，$\mathtt{r}_{new}$ はその命令で生成されるオブジェクトに割り当てられる新しいリージョン変数を表す．

キュー内のすべての命令の評価が終了した後[*2]，評価中に生成されたリージョン変数の生存期間を計算する．代入によって置き換えたリージョン変数が，次のいずれの条件

[*2] 命令列の構造によっては評価が停止しない可能性も考えられるため，実装では，3,000 回の命令の評価で終了しない解析はその時点で中止とする．

© 2020 Information Processing Society of Japan

getlocal var_name
$\Gamma_{local} \cup \{\text{var\_name} \Rightarrow \mathtt{r}_N\} \,.\, S \,.\, Subst \Longrightarrow \Gamma_{local} \cup \{\text{var\_name} \Rightarrow \mathtt{r}_N\} \,.\, S :: [\mathtt{r}_N] \,.\, Subst$

setlocal var_name
$\Gamma_{local} \,.\, S :: [\mathtt{r}_N] \,.\, Subst \Longrightarrow (\Gamma_{local} - \{\text{var\_name}\} \cup \{\text{var\_name} \Rightarrow \mathtt{r}_N\} \,.\, S \,.\, Subst$

getglobal var_name
$\Gamma_{global} \cup \{\text{var\_name} \Rightarrow \mathtt{r}_N\} \,.\, S \,.\, Subst \Longrightarrow \Gamma_{global} \cup \{\text{var\_name} \Rightarrow \mathtt{r}_N\} \,.\, S :: [\mathtt{r}_{heap}] \,.\, Subst$

setglobal var_name
$\Gamma_{global} \,.\, S :: [\mathtt{r}_N] \,.\, Subst \Longrightarrow \Gamma_{global} - \{\text{var\_name}\} \cup \{\text{var\_name} \Rightarrow \mathtt{r}_N\} \,.\, S \,.\, Subst \cup UNIFY(\mathtt{r}_{heap}, \mathtt{r}_N)$

getinstancevariable var_name
$\Gamma_{instance} \cup \{\text{var\_name} \Rightarrow \mathtt{r}_N\} \,.\, S \,.\, Subst \Longrightarrow \Gamma_{instance} \cup \{\text{var\_name} \Rightarrow \mathtt{r}_N\} \,.\, S :: [\mathtt{r}_{heap}] \,.\, Subst$

setinstancevariable var_name
$\Gamma_{instance} \,.\, S :: [\mathtt{r}_N] \,.\, Subst \Longrightarrow \Gamma_{instance} - \{\text{var\_name}\} \cup \{\text{var\_name} \Rightarrow \mathtt{r}_N\} \,.\, S \,.\, Subst \cup UNIFY(\mathtt{r}_{self}, \mathtt{r}_N)$

putself
$\Gamma \,.\, S \Longrightarrow \Gamma \,.\, S :: [\mathtt{r}_{self}] \,.\, Subst$

putstring ”str”
$\Gamma \,.\, S \,.\, Subst \Longrightarrow \Gamma \,.\, S :: [\mathtt{r}_{new}] \,.\, Subst$

newarray N
$\Gamma \,.\, S :: [\mathtt{r}_{N:1} \ldots \mathtt{r}_{N:N}] \,.\, Subst$
$\Longrightarrow \Gamma \,.\, S :: [\mathtt{r}_{new}] \,.\, Subst \cup UNIFY(\mathtt{r}_{N:1}, \mathtt{r}_{new}) \cup \ldots \cup UNIFY(\mathtt{r}_{N:N}, \mathtt{r}_{new})$

send :method, ARGV
$\Gamma \,.\, S :: [\mathtt{r}_{receiver},\ \mathtt{r}_{arg:1},\ \mathtt{r}_{arg:ARGV}] \,.\, Subst$
$\Longrightarrow \Gamma \,.\, S :: [\mathtt{r}_{heap}] \,.\, Subst \cup UNIFY(\mathtt{r}_{arg:1}, \mathtt{r}_{receiver}) \cup \ldots \cup UNIFY(\mathtt{r}_{arg:ARGV}, \mathtt{r}_{receiver})$

図 5　抽象解釈における命令の規則例
Fig. 5　Example of rules for instruction in abstract interpretation.

表 1　抽象解釈結果の例
Table 1　Examples of abstract interpretation results.

pc	instruction	operands	stack	vaariable environments
0000	pustring	”foo”	$S = [\mathtt{r2}]$	$\Gamma_{local} = \emptyset$
0002	setlocal_WC_0	a@0	$S = []$	$\Gamma_{local} = \{a \Rightarrow \mathtt{r2}\}$
0004	getlocal_WC_0	a@0	$S = [\mathtt{r2}]$	$\Gamma_{local} = \{a \Rightarrow \mathtt{r2}\}$
0006	setlocal_WC_0	c@1	$S = []$	$\Gamma_{local} = \{a \Rightarrow \mathtt{r2}, c \Rightarrow \mathtt{r2}\}$
0008	putstring	”bar”	$S = [\mathtt{r6}]$	$\Gamma_{local} = \{a \Rightarrow \mathtt{r2}, c \Rightarrow \mathtt{r2}\}$
0010	setlocal_WC_0	c@1	$S = []$	$\Gamma_{local} = \{a \Rightarrow \mathtt{r2}, c \Rightarrow \mathtt{r6}\}$
0012	getlocal_WC_0	c@1	$S = [\mathtt{r6}]$	$\Gamma_{local} = \{a \Rightarrow \mathtt{r2}, c \Rightarrow \mathtt{r6}\}$
0014	leave		$S = [\mathtt{r6}]$	$\Gamma_{local} = \{a \Rightarrow \mathtt{r2}, c \Rightarrow \mathtt{r6}\}$

にも当てはまらない場合，そのリージョン変数は解析したメソッドに対応するリージョンに割付け可能である．

(1) $\mathtt{r}_{heap}$ である
(2) $\mathtt{r}_{self}$ である
(3) 引数として渡されたオブジェクトのリージョン変数と同じである
(4) `leave` または `throw` 命令の評価時にスタックのトップに存在していた

ただし，(2) と (3) の条件にあてはまる場合はリージョン多相によるリージョン割付けが適用できる可能性がある．そのため，解析結果にはリージョン多相の適用可能性に関する情報も付与する．リージョン多相の実現と，解析によって付与される情報については，4.4 節で詳しく述べる．

抽象解釈による解析の例を示す．次のメソッドを抽象解釈によって解析する．

```
1 def func
2   a = "foo"
3   c = a
4   c = "bar"
5   return c
6 end
```

このメソッド `func` は次のような命令列にコンパイルされる．

```
0000 putstring          "foo"
0002 setlocal_WC_0      a@0
0004 getlocal_WC_0      a@0
0006 setlocal_WC_0      c@1
0008 putstring          "bar"
0010 setlocal_WC_0      c@1
0012 getlocal_WC_0      c@1
0014 leave
```

命令列の抽象解釈の結果を**表 1** に示す．簡単のため，ローカル変数環境以外の変数環境は使用していないため省略している．この結果から，メソッドの実行後に返り値が割付けられているリージョンを指すリージョン変数は `r6` と分かる．対して，リージョン変数 `r2` に紐づくオブジェクトはメソッド内で生存期間が完結しているため，このメソッドの呼び出しに対応するリージョンにメモリ割付け可能であると分かる．

© 2020 Information Processing Society of Japan

4.3　リージョン情報によるメモリ割付け最適化

抽象解釈の結果，各番地で発生するメモリの割付け先のリージョンが決定する．MJIT コンパイル時にその情報を用いて MJIT のコード生成時にメモリ割付けの最適化を行う．

リージョンの追加と削除

抽象解釈を行ったメソッドがリージョン割付け可能な番地の命令を保持している場合，そのメソッドの先頭で，新規にリージョンを生成する処理を挿入する．また，メソッドから返る直前には，生成したリージョンを削除する処理を挿入する．

リージョンは，メソッド呼び出しに対応して 1 つだけ生成される．削除されるリージョンもそのメソッドに対応するリージョン 1 つだけである．

リージョン割付けへの置き換え

ある番地の命令でリージョン割付けが可能だと判明している場合，MJIT のコード生成時にヒープ割付けの処理をリージョン割付けへと置き換える．置き換え対象の割付けがリージョン多相の対象とならない場合，メソッドに紐づくリージョンへと割り付けられる．リージョン多相の対象となる場合，実引数のオブジェクトからその割付け方法を判定し，リージョン割付けの場合は同じリージョンへオブジェクトを割り付ける．

YARV の命令列のうち，オブジェクトを割り付ける命令は，String や Array といった組み込みクラスに対してしか用意されていない．特殊な命令が用意されていないような標準ライブラリやユーザ定義のクラスは，すべて `Class#new` という C 言語で実装されたメソッドが呼ばれ，そのなかでメモリ割付けが行われる．この `Class#new` はユーザによってオーバライド可能なメソッドであるため，保守的な解析ではオブジェクトが生成されるかを判定することができない．よって，本提案では，リージョン割付け対象のオブジェクトを，命令から割付けを検知することのできる String，Array，Hash の 3 つのクラスに限定する．

4.4　リージョン多相による最適化

2.5 節で解説したようなリージョン多相の完全な実現には，呼び出す先のメソッドが要求するリージョン多相のパラメータを計算しなければならない．そのためには，メソッド呼び出し関係にあるメソッド間で追加の解析が必要になる．動的にメソッドがディスパッチされる Ruby では，既存手法と同等なリージョン多相の仕組みを提供することは難しい．本提案では，解析中に得られたリージョン多相に関する情報を用いて，実行時に特定の場合のみリージョン多相を適用する．

リージョン多相が適用できる可能性のあるオブジェクトは，抽象解釈による解析で決定できる．オブジェクトに対応するリージョン変数が，単一化後に，引数として渡されたオブジェクトのリージョン変数，もしくは返り値となるリージョン変数と同等な場合である．前者は引数と同じリージョンにメモリを割り付けることができ，後者は呼び出し側の指定したリージョンにメモリを割り付けることができる．それらに該当するリージョン変数の情報は，解析結果として付与される．

リージョン多相に使用されるリージョン変数が一般化されるのは，実際にメソッドが呼び出された時点である．本提案では，呼び出し先のメソッドが，渡された実引数のそれぞれについて，割り付けられているメモリがヒープであるか，リージョンであるか，またリージョンである場合はどのリージョンであるかを特定する．特定には，4.5.3 項で説明するリージョンのメモリレイアウトの特性から，メモリ番地を計算して求めることができる．なお，解析の時点でリージョン多相が適用できないと判明している場合は，これらの計算はスキップされる．呼び出し元のメソッドでリージョン変数に関するパラメータを付与する方法も考えられるが，呼び出す先のメソッドが最適化されていないが，そのメソッドを介して最適化されたメソッドを呼び出すことを考えると，すべてのメソッド呼び出しにおいて，リージョン変数に関するパラメータの計算が必要になる．これはパフォーマンスの低下を招く可能性があるため，本提案では呼び出し先でリージョンの特定を行っている．そのため，返り値のリージョンを指定する振舞いについても，提案および計測時点では実装されていない．

リージョン多相の適用可能性に関する情報が付与される抽象解釈の例を示す．

```
1 def poly_func(a)
2   c = "str"
3   a << c
4 end
```

このメソッド `poly_func` は次のような命令列にコンパイルされる．

```
0000 putstring              "str"
0002 setlocal_WC_0          c@1
0004 getlocal_WC_0          a@0
0006 getlocal_WC_0          c@1
0008 opt_ltlt               <<
0011 leave
```

命令列の抽象解釈の結果を**表 2** に示す．簡単のため，ローカル変数環境以外の変数環境は使用していないため省略している．

0008 番地の命令 opt_ltlt では，変数 `a` に束縛されているオブジェクトをレシーバとし，変数 `c` に束縛されているオブジェクトを引数として与え，何らかのメソッドを呼び出す．たとえば，`a` が `Array` クラスのインスタンスであ

© 2020 Information Processing Society of Japan

表 2　リージョン多相が有効な抽象解釈結果の例
Table 2　Abstract interpretation results that enable region polymorphism.

pc	instruction	operands	stack	vaariable environments
0000	pustring	”str”	$S = [\mathtt{r}_2]$	$\Gamma_{local} = \{a \Rightarrow \mathtt{r}_1\}$
0002	setlocal_WC_0	c@0	$S = []$	$\Gamma_{local} = \{a \Rightarrow \mathtt{r}_1, c \Rightarrow \mathtt{r}_2\}$
0004	getlocal_WC_0	a@0	$S = [\mathtt{r}_1]$	$\Gamma_{local} = \{a \Rightarrow \mathtt{r}_1, c \Rightarrow \mathtt{r}_2\}$
0006	getlocal_WC_0	c@1	$S = [\mathtt{r}_1, \mathtt{r}_2]$	$\Gamma_{local} = \{a \Rightarrow \mathtt{r}_1, c \Rightarrow \mathtt{r}_2\}$
0008	opt_ltlt	<<	$S = [\mathtt{r}_1]$	$\Gamma_{local} = \{a \Rightarrow \mathtt{r}_1, c \Rightarrow \mathtt{r}_1\}$
0011	leave		$S = [\mathtt{r}_1]$	$\Gamma_{local} = \{a \Rightarrow \mathtt{r}_1, c \Rightarrow \mathtt{r}_1\}$

る場合，それは配列への追加を表す．抽象解釈では，どのメソッドが呼び出されるか決定できず，レシーバと引数の生存期間に関する関係を導くことができないため，opt_ltlt が評価される時点で a と c に束縛されているオブジェクトのリージョンを単一化する．

抽象解釈によるメソッド poly_func の解析結果より，0000 番地の命令で生成される文字列オブジェクトの生存期間は，引数として与えられているオブジェクトの生存期間と同じであると見なすことができる．つまり，引数のオブジェクトがリージョン割付けされている場合，リージョン多相を適用して，文字列オブジェクトを同じリージョンに割付け可能であることを意味する．このリージョン多相に関する情報を解析結果として付与し，前述した呼び出し先メソッドによる割付けの判定を用いて，実行時にリージョン割付けの判定を実施する．

4.5　CRuby 処理系への実装

実現可能性について議論するため，上記のアルゴリズムと最適化を CRuby 処理系に実装した．CRuby 処理系のコンパイルおよびテストではエラーが見られないように実装を追加しているが，既存の拡張ライブラリや C 言語で実装された拡張との完全な互換性は達成していない．それらの互換性を担保する方針や課題については 6.3 節で考察する．

この節では，CRuby 処理系の実装と互換性を保つために追加した機構について説明する．

4.5.1　リージョンの最適化解除

Ruby の機能にはリージョンによるメモリ管理の前提としている参照制約を破る可能性がある機能が存在する．それらの機能が使用された場合，リージョン割付けされているオブジェクトの解放によって，予期しないぶら下がりポインタの発生してしまう．そのため，対象となる機能が使われた際には，オブジェクトのメモリ管理をガベージコレクションに移譲させる必要がある．本提案では，そうしたリージョン割付けの最適化解除が，CRuby の既存機能との互換性のために提供されている．

リージョン割付けの最適化解除は，次のような機能が使われた場合にスケジュールされる．最適化解除がスケジュールされた場合，対象となるオブジェクトがリージョンごと削除されるタイミングで，リージョン割付けの最適化解除，つまり，オブジェクトのメモリ管理がガベージコレクションへ移譲される．

まずは，リージョンによるメモリ管理の前提としてしる参照制約を破りうる言語機能が使われた場合である．ObjectSpace.#each_object メソッドは，呼び出すことでメモリ空間に存在するすべてのオブジェクトを取得できる．このメソッドが呼び出されたスコープから参照することのできないオブジェクトも取得できてしまう．Kernel.#binding メソッドは，現在の実行コンテキストで参照可能なローカル変数のテーブルをすべて取得する．ローカル変数は Kernel.#binding を実行したスコープよりも外側のスコープも取得できるため，それらに対して代入が行われる可能性を考慮すると，最適化解除をせざるを得ない．グローバル変数への代入も最適化解除が必要である．リージョン割付けによって最適化されたメソッドから呼び出されたメソッド中で，リージョン割付けされたオブジェクトがグローバル変数に束縛された場合，そのオブジェクトは，リージョンの生存期間が過ぎても参照できていなければならないためである．

次に，MJIT 自体の最適化が解除される場合である．これは MJIT が解除される条件にメソッドの再定義が含まれているため，実行時リージョン解析によって得た情報が信頼できなくなるためである．MJIT が解除されるケースは 3.4 節でも述べている．

4.5.2　ヒープ領域への書き戻し

リージョン割付けの最適化解除では，ぶら下がりポインタの発生を防ぐため，リージョン割付けされたオブジェクトをガベージコレクションの管理するヒープ領域へ移動しなければならない．しかし，単にヒープ割付けされた領域へ移動するだけでなく，そのオブジェクトを参照するポインタをすべて書き換えなければならない．そのポインタを発見するために，すべてのポインタの使用時において，リードバリアやライトバリアを用いる方法が考えられるが，CRuby 処理系の実行ではポインタの使用は頻繁に行われるため，パフォーマンスに悪影響を及ぼす可能性がある．本提案の実装では，4.5.3 項で説明するリージョンのメモリ構造を用いることで，リージョン単位で書き戻しを

© 2020 Information Processing Society of Japan

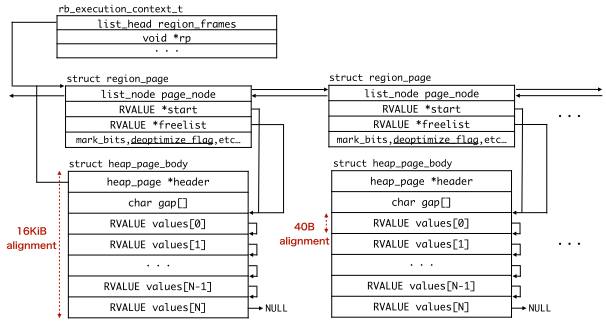


図 6　リージョンページの構造
Fig. 6　Structure of the region page.

実現してメモリの移動を避け，リードバリアやライトバリアの必要性を排除している．

4.5.3　リージョンのメモリ構造とヒープ領域

リージョンに使用されるメモリ領域は RVALUE のリストを持つ．リージョンへメモリの割付け要求がされた場合，ヒープ割付けと同様に RVALUE へのポインタを返す．リージョンの削除と同時に，割付けられているすべてのオブジェクトのメモリも解放される．

この項では，リージョンに使用されるメモリ領域の実装について述べる．ぶら下がりポインタの発生を避けてヒープ領域への書き戻しを実現するため，書き戻し時にオブジェクトのメモリを移動させない設計を採用した．

本提案の実装におけるリージョンは，そのメモリ領域として，ヒープページと同じサイズの**リージョンページ**（**region page**）を持つ．Ruby の実行コンテキストの構造体である rb_execution_context ごとにリージョンページのフリーリストが用意される．rb_execution_context とリージョンページの構造を**図 6** に示す．

リージョンを新規に追加時には，そのフリーリストから新規にリージョンページが割り当てられ，削除時にはフリーリストへと返される．これによってリージョンページの再利用が可能になり，リージョンの追加と削除ごとに malloc や free の操作が削減される．

最適化解除がスケジュールされた場合，個々のオブジェクトをヒープ領域へ書き戻すのではなく，書き戻し対象のオブジェクトが含まれるリージョンページを，ガベージコレクションの管理するヒープページのリストへ追加する．リージョンページはヒープページと同じサイズで確保されているので，フィールドの初期化を行うだけでヒープページとして見なせる．オブジェクトのメモリ番地も変わらないため，リードバリアやライトバリアなしに書き戻せる．

リージョンページ単位での書き戻しは，通常，連鎖的に発生する．書き戻す対象のリージョンページのうちから，さらに寿命の長いリージョンページへの参照がありうるため，最適化解除によってリージョンページの書き戻しが実行されるとき，起点とするリージョンページと，それよりも寿命の長いリージョンページをすべて書き戻すことになる．

5.　実装の評価

4 章で説明した実行時リージョン解析とその最適化を CRuby 処理系に実装した．この章では，いくつかのケースを用いてガベージコレクションやパフォーマンスの変化を比較し，その結果について考察する．ベンチマークで使用した環境は以下のとおりである．

- OS：Ubuntu 18.04.3 LTS（Linux 4.15.0-51-generic）
- CPU：Intel(R) Xeon(R) CPU E5-2440 v2 @ 1.90 GHz
- CPU Cache：20 MB
- Memory：98304 MB
- Compiler：clang version 6.0.0-1ubuntu2

5.1　特徴的なケースでの比較

実行時リージョン解析やリージョンによるメモリ管理の特性を実行例によって確認するため，特にその特性が顕著に出るケースを用いて計測と比較を行う．計測には benchmark_driver というライブラリを使用する[*3]．

計測

ベンチマーク対象のメソッドを実行することで計測する．計測は 10 回行い，その平均をグラフの値として，最大値・最小値をひげとして示す．計測する項目は以下の 3 つである．

- 1 秒あたりのメソッド実行回数 IPS
  （Iteration Per Second）
- 最大メモリ使用量 Max RSS
  （Resident Set Size）
- ガベージコレクションの回数 GC Count

比較対象

- CRuby 2.6.2：
  `2.6.2`
- MJIT を有効にした CRuby 2.6.2：
  `2.6.2(jit)`
- 提案手法を実装した CRuby 2.6.2：
  `2.6.2-region`
- 提案手法を実装して MJIT を有効にした CRuby 2.6.2：
  `2.6.2-region(jit)`
- 提案手法の実装と MJIT を有効にした CRuby 2.6.2：
  `2.6.2-region(jit,analysis)`

5.1.1　単純な文字列割付け

リージョンによるメモリ管理によってパフォーマンスが向上する例を見る．メソッド呼び出しごとにリージョン割付け可能な文字列を割り付ける次のようなメソッドを考える．

[*3] benchmark_driver は CRuby 処理系のリポジトリに配置されているベンチマークスクリプトの実行にも使用されている．ライブラリのソースコードは https://github.com/benchmark-driver/benchmark-driver で公開されている．

© 2020 Information Processing Society of Japan

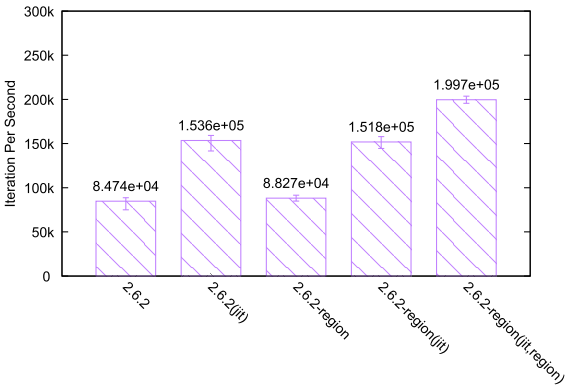


図 7　単純な文字列割付け：IPS
Fig. 7　Simple string allocation: IPS.

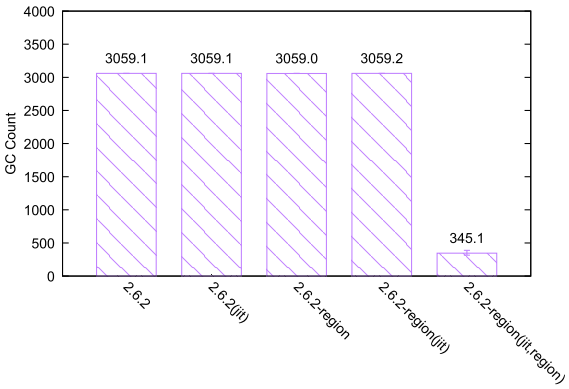


図 9　単純な文字列割付け：GC Count
Fig. 9　Simple string allocation: GC Count.

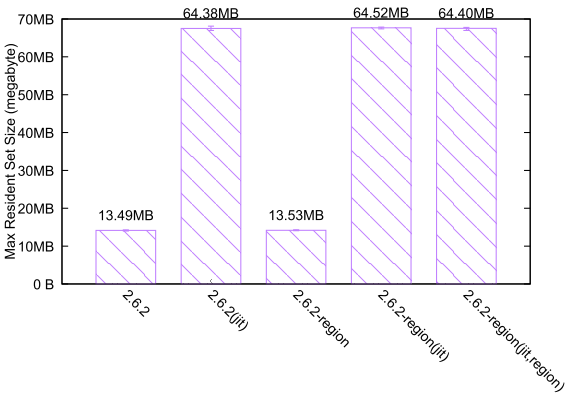


図 8　単純な文字列割付け：Max RSS
Fig. 8　Simple string allocation: Max RSS.

```
def string_region_alloc_100
  n = 100
  while n > 0
    n -= 1
    s = "string"
  end
  return nil
end
```

string_region_alloc_100 メソッドは参照がメソッドで閉じている 100 個の String オブジェクトを生成する．実行時リージョン解析によってメソッドが解析されると，String オブジェクトのメモリはすべてリージョンに割付け可能と判定される．

計測結果の IPS を図 7，Max RSS を図 8，GC Count を図 9 に示す．それぞれ，メソッドを 30 万回呼び出した結果の数値である．IPS の結果から，実行時リージョン解析を有効にした場合，MJIT の有効無効にかかわらず，パフォーマンスの改善を見ることができる．パフォーマンスが改善された理由として，GC Count の結果を参照することで，リージョン割付けによってガベージコレクションの回数が減ったからであると理由付けできる．Max RSS の結果は，MJIT の有無によって違いが生じている．これは，MJIT のコンパイル処理を行う別のスレッドのためにメモリを多く必要とすることや，MJIT でコンパイルしたメソッドをメモリに読み込む必要があるため，実行時にリージョン解析の実装とは関係なくメモリ使用量が増加していると考えられる．

5.1.2　再帰呼び出し

リージョン割付け可能なメソッドの再帰呼び出しが深くなるような例を見る．本提案のリージョンはメソッドの呼び出しと終了に対応してリージョンが追加・削除されるため，再帰の深くなるようなプログラムはメモリの解放ができない．

```
def region_rec_alloc_1(depth)
  n = 1
  while n > 0
    n -= 1
    s = "string"
  end

  region_rec_alloc_1(depth - 1) if depth >
      0
end
```

region_rec_alloc_1 メソッドは参照がメソッドで閉じている 1 個の String オブジェクトを生成する．実行時リージョン解析によってメソッドが解析されると，String オブジェクトのメモリはすべてリージョンに割付け可能と判定される．引数 depth が 0 より大きい場合，region_rec_alloc_1 は再帰的に呼び出される．このとき，5 行目でリージョン割付けをした String オブジェクトはすでに生存期間を過ぎているが，メソッドから返らないため，メモリを解放することができない．

計測結果の IPS を図 10，Max RSS を図 11，GC Count を図 12 に示す．depth の値は 30000 とし，それぞれ，メソッドを 10 回呼び出した結果の数値である．IPS の結果を見ると，一見パフォーマンスが改善しているように見える．しかしながら，Max RSS や GC Count の結果から，再

© 2020 Information Processing Society of Japan

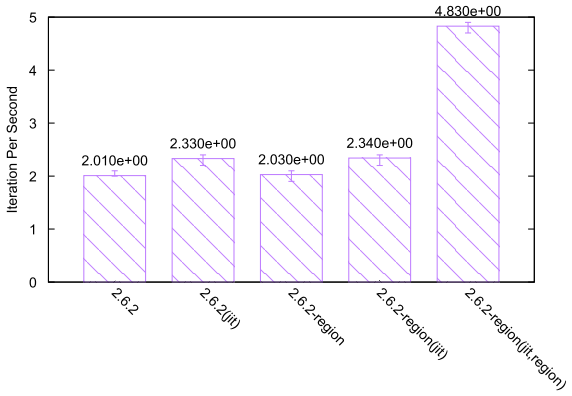


図 10　再帰呼び出し：IPS
Fig. 10　Recursive call: IPS.

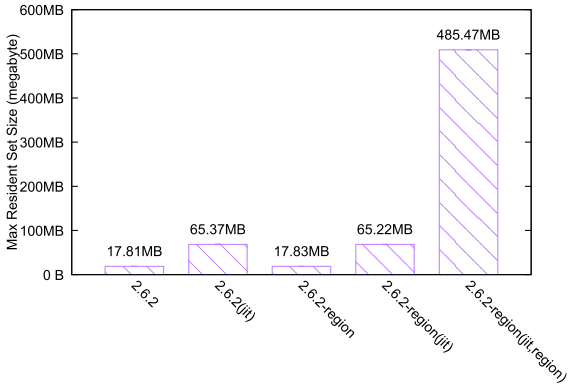


図 11　再帰呼び出し：Max RSS
Fig. 11　Recursive call: Max RSS.

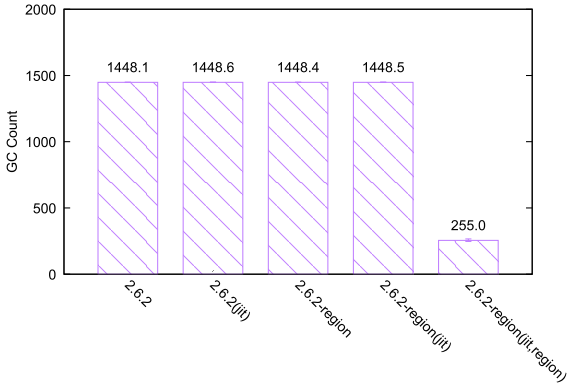


図 12　再帰呼び出し：GC Count
Fig. 12　Recursive call: GC Count.

帰呼び出し中に削除することができないリージョンにオブジェクトが割り付けられることによって，ガベージコレクション自体の機会が減少していることが原因であることが分かる．本提案の実装では，リージョン割付けを起因としたガベージコレクションの実行を予約していないため，ガベージコレクションの回数が減少している．しかしガベージコレクションの実行戦略によっては，メモリの使用量の増加などによって，反対にガベージコレクションの頻度が高くなる場合もありうる．事実，Max RSS の値は削除することのできないリージョンのメモリによって大きく増加

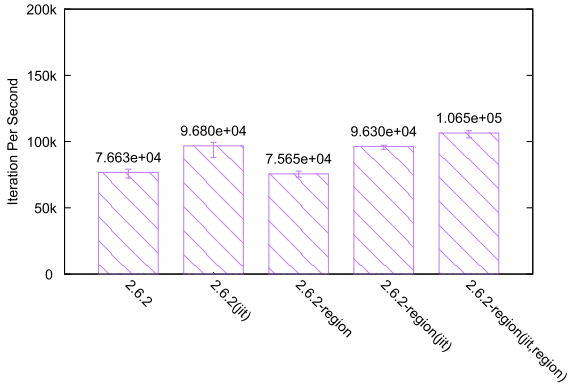


図 13　リージョン多相：IPS
Fig. 13　Region polymorphism: IPS.

しており，速度面でのパフォーマンスと引き換えに空間計算量が悪化していることが分かる．

5.1.3　リージョン多相

リージョン多相が有効になる例を見る．リージョン多相によって，参照が呼び出し元のメソッドへわたる場合においてもオブジェクトのリージョン割付けを適用することができる．

```
1  def region_poly_alloc_100
2    [1].each_with_object([]) do |_, a|
3      n = 100
4      while n > 0
5        n -= 1
6        a << "string"
7      end
8      nil
9    end
10   return nil
11 end
```

`region_poly_alloc_100` メソッドは呼び出しごとに Array オブジェクトを生成し，ブロックを呼び出す．ブロックは 100 個の String オブジェクトを生成し，先程生成した Array オブジェクトに追加する．実行時リージョン解析によってメソッドが解析されると，Array オブジェクトと String オブジェクトのメモリはすべて**同じ**リージョンに割付け可能と判定される．

保守的なリージョン解析においてもリージョン多相が適用されるようなメソッドを定義するため，`[1].each_with_object([])` のようにしてブロックを呼び出し実行している．レシーバオブジェクトを保守的な解析で決定することが困難であることから，工夫した形での呼び出しとなった．こうした保守的な解析の課題については 6.4 節でも詳しく述べる．

計測結果の IPS を**図 13**，Max RSS を**図 14**，GC Count を**図 15** に示す．それぞれ，メソッドを 30 万回呼び出した結果の数値である．単純な文字列割付けを適用した場合

© 2020 Information Processing Society of Japan

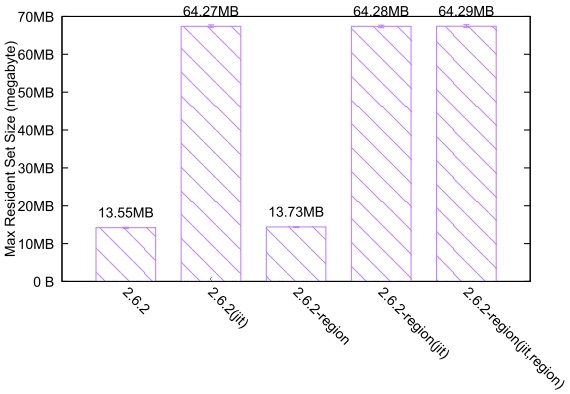


図 14　リージョン多相：Max RSS
Fig. 14　Region polymorphism: Max RSS.

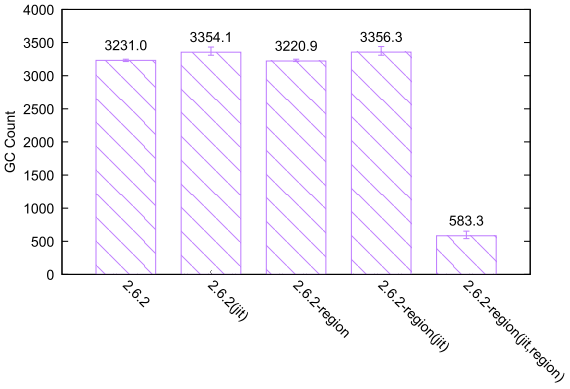


図 15　リージョン多相：GC Count
Fig. 15　Region polymorphism: GC Count.

と同様に，ガベージコレクションの回数が削減されたことによってパフォーマンスの向上が見られた．

5.2　rdoc ベンチマークでの比較

CRuby 処理系でガベージコレクションの性能評価に用いられる rdoc ベンチマークを用いて実行時リージョン解析の評価を行う．rdoc ベンチマークでは，リポジトリ内の Ruby プログラムファイル一覧を使ってドキュメント生成を行う．生成にかかる時間およびガベージコレクションの回数を計測することができる．

計測

ベンチマーク走行における実行時間とガベージコレクションの回数を計測する．計測は 10 回行い，その平均をグラフの値として，最大値・最小値をひげとして示す．

比較対象

- CRuby 2.6.2：
  `2.6.2`
- MJIT を有効にした CRuby 2.6.2：
  `2.6.2(jit)`
- 提案手法を実装した CRuby 2.6.2：
  `2.6.2-region`
- 提案手法を実装して MJIT を有効にした CRuby 2.6.2：
  `2.6.2-region(jit)`
- 提案手法の実装と MJIT を有効にした CRuby 2.6.2：
  `2.6.2-region(jit,analysis)`

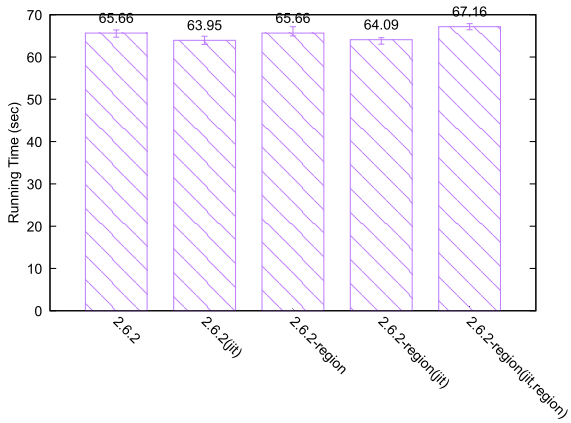


図 16　rdoc ベンチマーク：実行時間
Fig. 16　Benchmark of rdoc: Running Time.

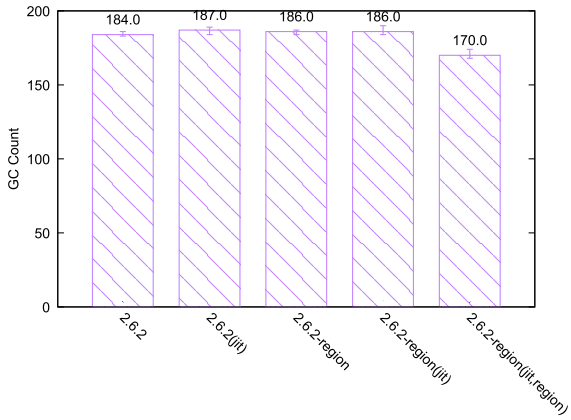


図 17　rdoc ベンチマーク：GC Count
Fig. 17　Benchmark of rdoc: GC Count.

計測結果の実行時間を**図 16** に，ガベージコレクションの回数を**図 17** に示す．実行時リージョン解析によってガベージコレクションの回数は削減しているものの，実行時間に優位な差は見られず，実行時リージョンを使用しない場合と比較してわずかにパフォーマンスが悪化している．

5.3　考察

上記のベンチマーク結果について，パフォーマンスが悪化した理由を調査した．

5.3.1　組み込み最適化命令による最適化解除

YARV は，コンパイル時に `#[]` や `#<<` などのメソッド呼び出しを特別扱いし，組み込み最適化命令に置き換える．組み込み最適化命令は，そのレシーバのクラスがビルトインであるとき，あらかじめ用意された関数を呼び出す．メソッドの動的ディスパッチによる探索時間を分岐 1 つで削減できるため，VM の実行パフォーマンスを向上させることができる．MJIT はその性質を利用して，組み込み最適化命令のレシーバがビルトインのクラスであることを前提

© 2020 Information Processing Society of Japan

にした最適化を適用する．もしビルトインのクラス以外のオブジェクトがレシーバとなった場合，MJIT は最適化解除を行う．これは，リージョン割付けされたオブジェクトのヒープ領域への書き戻しも発生する．rdoc ベンチマークおよびそのほかの Ruby プログラムでは，ビルトインのクラス以外で `#[]` や `#<<` などのメソッド使用例が多く存在するため，最適化解除によってパフォーマンスの向上が見られなかったと考える．組み込み最適化命令が使用されるような構文の記述を避けることは，最適化解除を避けるために有用であるが，メソッドの探索が必要になるため，可読性やパフォーマンスに影響が出る可能性がある．こうした Ruby のプログラミングスタイルとの兼ね合いに関する考察は 6.5 節でも述べる．

5.3.2　MJIT のコンパイル時間

MJIT は C 言語に変換したソースコードのコンパイルを gcc や clang を通して行う．これには最適化した命令列を実行するのに必要なヘッダファイルも含まれるため，コンパイル時間は短くない．1 つのメソッドをコンパイルするには，評価環境でおおよそ数百ミリ秒から数千ミリ秒必要である．

MJIT の対象となるメソッドは rdoc ベンチマークのプログラム内に 1,000 個程度存在する．対して，rdoc ベンチマークの実行時間は 60 秒から 70 秒程度で終了するため，MJIT のワーカーによってメソッドがコンパイルが終わる前にプログラムが終了していると考えられる．MJIT で同期的にコンパイルを行うオプションを有効にして実行時間を計測したところ，実行時間は約 500 秒程度増加したため，実行時間に対してコンパイルに時間がかかりすぎていることが確認できる．事前にコンパイルしている環境を再現するため，計測前にベンチマークと同じプログラムを複数回実行するケースでの計測も行ったが，組み込み最適化命令による最適解除の件数が増加し，パフォーマンスの改善は見られなかった．

6.　考察

6.1　既存のメモリ管理機構との共存

実行時リージョン解析では，部分的にメモリ管理をリージョンによるメモリ管理へ置き換える．ガベージコレクションが存在する場合は，そのアルゴリズムとの互換性を保持しなければならない．ガベージコレクションのアルゴリズムによっては，リージョンの削除後にヒープ領域内のオブジェクトからリージョンのメモリ領域への参照が残っている場合，ガベージコレクション時に不正な参照が発生してしまう可能性がある．たとえば，ガベージコレクションをプログラムの実行と並列に実行する処理系の場合，ガベージコレクションの最中にリージョンの削除が発生する可能性があるため，同時にリージョンの削除が発生しないよう遅延させるといった工夫が必要である．

本提案で実装の対象とした CRuby の処理系は，マークアンドスイープ方式でメモリ管理を行う．このアルゴリズムはルート集合から生存しているオブジェクトをたどるため，解析によって生存期間を過ぎたと判定されるオブジェクトは，ガベージコレクションにたどられるオブジェクトから参照されていることはない．ただし，リージョン内のメモリ領域を削除前に再利用するといった最適化を考える場合，ルート集合となる VM の実行コンテキストから参照が残る可能性があるため，ガベージコレクション中の参照にはリードバリアが必要である．メモリの再利用については次の 6.2 節で詳しく述べる．

6.2　リージョンサイズ解析と再利用

本提案のリージョンは，メソッドの呼び出しに対応して追加と削除がされる．この方針は，リージョン割付けが有効なメソッドを再帰的に呼び出す場合，リージョンが追加され続けるため，実行時のメモリ領域を圧迫する可能性がある．一部のメソッドは，末尾再帰による最適化を実現することで，再帰呼び出しごとにリージョンを部分的に再利用することが可能である．しかし，末尾再帰ではない，もしくは末尾再帰に変換することのできないメソッドには対応できない．

Tofte らはリージョン推論に加えて，上記のような問題の対策として，Storage Mode Analysis と Physical Size Inference によってメモリ領域の再利用性を静的に導く方法を提案している [13]．これらの解析や推論によって末尾再帰による最適化や，リージョンのサイズが有限である場合静的に決定することができる．動的言語においてこれらの手法が有効であるかの検討は今後の課題である．

メソッド内で実行される反復についても似たような問題がある．反復処理ごとに生存期間を過ぎるオブジェクトが存在した場合でも，リージョンが削除されるまでメモリを解放することができないため，無駄なメモリ領域が生じてしまう．反復処理に応じたリージョンによるメモリ管理の適用は関連研究で詳しく述べる．

6.3　標準ライブラリや拡張ライブラリとの互換性

本提案の実装は，CRuby 処理系のコンパイルおよびテストの要件と，ベンチマークで使用したようなプログラムとの互換性は満たしている．しかしながら，Ruby の標準ライブラリ全体や，CRuby 2.6.2 時点で動作するすべての Ruby プログラムで互換性を保つにはまだ考慮する点が残る．

CRuby の API を使用して C 言語で実装されているメソッドは，YARV の命令列にコンパイルされないため，実行時でもリージョン解析が不可能である．それだけでなく，C 言語の構造体にマッピングされるクラス定義やメモリ割付けが可能であるため，実装によってはリージョンが前提

© 2020 Information Processing Society of Japan

としている参照制約を破る可能性がある．それらの C 言語実装との互換性を保つためには，適切な位置に最適化解除を挿入しなければならない．しかしながら，後方互換性を担保することを考えると，最適化解除を必要な箇所にのみ追加することは難しい．保守的な解決案としては，C 言語で実装されたメソッドを呼び出す際に必ず最適化解除を行う方法があるが，CRuby の機能の多くが C 言語で実装されていることもあり，パフォーマンスの観点からは避けたい．現状の解決案は，リージョン割付けが存在する場合でも安全に呼び出せる，もしくは必要に応じて最適化解除を呼び出すように変更された C 言語実装のメソッドのホワイトリストを生成することである．本提案では，最適化解除が必要な C 言語実装のメソッド定義は事前に対応することで擬似的にホワイトリスト方式の状態を再現している．

CRuby の実行コンテキストは Ruby レベルのスレッド単位で生成される．そのため，リージョンのメモリ領域もスレッド単位で用意しなければならない．Ruby における `Thread` クラスの実装には，他のスレッドに対して割り込みを行い，指定したブロックの実行や例外の発生を可能にする機能がある．それらの機能により，マルチスレッドでのプログラミングに柔軟性が持たされているが，リージョンのようなスレッドローカルなメモリ領域とは相性が悪い．本提案の実装では，最も保守的に実行をする．複数のスレッドが協調して動作を行うような機能においては，`Thread` クラスのメソッドが呼び出された時点で最適化解除がスケジュールされる．そのため，マルチスレッドを多用するようなプログラムでは，本提案の実装による最適化は有効でない場合が多くなる．

6.4　保守的な解析の課題

実行時リージョン解析のアルゴリズムはぶら下がりポインタの発生を防ぐため，想定されるすべての状態を考慮して保守的に実装されている．そのため，プログラム全体で見るとリージョン割付け可能であったとしても，ヒープ割付けと判断される可能性がある．特定のベンチマークではパフォーマンスの向上が見られたものの，実用的なプログラムでのパフォーマンスはそこまで変化が見られなかった．この節では，よりリージョン割付けの機会を増やし，実用的なプログラムでもリージョン割付けの機会を増やす方法をいくつか議論する．

動的ディスパッチの情報を使って特化コードを生成する方法を考える．YARV は，特定の番地で一度呼び出したメソッドやブロックをキャッシュすることができる．キャッシュ情報はリージョン解析においても参照可能であるため，メソッドやブロック間のリージョン解析で利用可能である．しかし，キャッシュが無効になった場合，特化コードの仮定が信用できなくなるため，最適化解除が必要になる．プログラムの実装によってはキャッシュがつねに有効になるような記述も存在するため，実行のプロファイリングをすることで特化コードの生成が可能だと考えている．

リージョン割付け可能であることをプログラマに記述させる方法も考えられる．プログラマによって文脈を考慮した割付けのヒントを付与させ，解析時に参考にすることで，プログラマが保証した条件の元で楽観的にリージョン割付けが可能だと考える．

割付けの情報に限らず，型情報も解析に活かすことができる．引数やレシーバの型が記述されている場合，前述した特化コードの生成が可能になるためである．Ren らはメタプログラミングによってメソッドの定義がされるような動的言語に対して，実行時のメソッド定義時に型情報を収集し，実行時型検査を行う仕組みを提案している [14]．評価に使用されている Ruby では，Rails のような巨大なフレームワークに対しても型検査を可能にしている．型検査によってメソッドの引数となる型が決定的になることは，特化コードの実行において最適化解除の機会も減少することになる．

6.5　プログラミングスタイルとの兼ね合い

Ruby のプログラムの多くは，ガベージコレクションが前提に実装されていることが多い．たとえば，メソッドの最後の式が返り値になることを考慮しない実装は，リージョン割付けの機会を減少させる．Ruby プログラムからドキュメントを生成する rdoc ライブラリのソースコードを例にとる[*4]．

```
1  def update_output_dir(op_dir, time, last = {})
2    return if @options.dry_run or not @options.update_output_dir
3    unless ENV['SOURCE_DATE_EPOCH'].nil?
4      time = Time.at(ENV['SOURCE_DATE_EPOCH'].to_i).gmtime
5    end
6
7    File.open output_flag_file(op_dir), "w" do |f|
8      f.puts time.rfc2822
9      last.each do |n, t|
10       f.puts "#{n}\t#{t.rfc2822}"
11     end
12   end
13 end
```

このメソッドの 9 行目から 11 行目にかけて記述されているブロックは，ファイルへの書き込み操作を行う．10 行目

[*4] https://github.com/ruby/ruby/blob/d254d5563e0e599f25aca9507683ee0abb9e18de/lib/rdoc/rdoc.rb#L224-L239

© 2020 Information Processing Society of Japan

f.puts の引数として生成される文字列オブジェクトは，ブロック呼び出し後に不要となるが，Ruby のメソッドやブロックにおいて最後に書いた式がそのまま返り値となることを考えると，保守的なリージョン解析では不要であることを想定することができないため，このブロックに対応するリージョンにメモリ割付けができない．例のプログラムは，以下のように書き直すことでリージョン割付けを適用することができる．ただし，この書き換えはプログラムの想定する挙動を変えることはない．

```
def update_output_dir(op_dir, time, last = {})
  return if @options.dry_run or not @options.update_output_dir
  unless ENV['SOURCE_DATE_EPOCH'].nil?
    time = Time.at(ENV['SOURCE_DATE_EPOCH'].to_i).gmtime
  end

  File.open output_flag_file(op_dir), "w" do |f|
    f.puts time.rfc2822
    last.each do |n, t|
      f.puts "#{n}\t#{t.rfc2822}"
      nil # inserted this line
    end
  end
end
```

ガベージコレクションでメモリ管理がされていることを前提とすると，変換前のコードにおいても（rdoc では返り値は用いられず捨てられているため），次のガベージコレクションのタイミングで文字列オブジェクトが回収される点はおなじであり，変換によってパフォーマンスやメモリ効率に影響はない．

リージョンによるメモリ管理を適用することは，それまでのメモリ管理の都合上問題ないとされてきたプログラミングスタイルに影響を与える可能性がある．特に Ruby の場合は，明示的に返り値がないことを表現する方法も存在しないため，代わりに即値となる nil の値などを用いることになる．

7. 関連研究

7.1 空間計算量を削減した楽観的なスタック割付け

Corry は Java に似た言語に対して，既存のスタック割付けやリージョン割付けに見られるような，再帰や反復で空間計算量が悪化する課題を解決しつつスタック割付けを実現する方法を提案した [15]．提案におけるスタック割付けは楽観的に行われる．ヒープを最も生存期間の長いスタックフレームと仮定し，オブジェクト間の参照は，スタックの生存期間順と逆方向のみを許容する．生存期間順と順方向の参照が生じる場合，まず，参照先のオブジェクトをヒープ領域へ移動させ，参照元を移動先へ書き換える．次に，参照先のオブジェクトが含まれていたスタックフレームより生存期間の短いすべてのスタックフレームを走査し，参照先のオブジェクトを参照していた逆方向の正しい参照を移動先へ書き換える．参照もすべて移動させることによって，フォワーディングポインタが必要なくなるため，リードバリアを挿入することなくメモリ番地の移動を実現することができる．

Corry の提案では，スタックはコールスタックとオブジェクトスタックに分けられる．前者はメソッド呼び出しなどにともなって操作され，後者は前述したスタック割付けのために使われる．オブジェクトスタックにおけるフレームの追加と削除は反復処理の繰り返しごとに行われる．メソッドの呼び出しや終了時に操作は行われず，同じスタックフレームを共有する．各反復の先頭でリージョンを追加し，各反復の終了時にリージョンを削除するため，反復処理内に反復処理が存在する場合はスタックフレームが 2 度追加される．このスタック操作によって，再帰呼び出しによって無駄なスタック領域が生じてしまう問題や，反復処理を含むメソッド中でスタックのサイズが決定できない問題を解決する．また，スタック割付けの解析は反復処理ごとに行われるため，内部に反復処理を含まなくなり，各スタックフレームのサイズは決定的である．

本提案では，リージョンページをヒープページに変換することによって，オブジェクトのヒープ領域への移動を実現している．Corry の提案は，ヒープ領域への移動の際，スタックフレームの走査が必要になるため，スタックのサイズに応じたパフォーマンスへの影響が考えられる．特に本提案で扱った動的言語のように，頻繁に最適化解除が発生する可能性のある言語においては，メモリ番地の変更が必要ないページ単位での移動が有効だと考える．

本提案の実装は，Ruby のブロックごとにリージョンを生成するため，Enumerable モジュールを使用した反復処理ごとにリージョンが確保される．しかし，while や for といった組み込みの構文を使用して反復処理を記述した場合，YARV の命令列として反復が表現されるため，反復を含むようなメソッドはリージョンのサイズを決定することが困難である．命令列の構造から反復処理を検出し，反復処理に応じてリージョンの追加と削除を行う方法も検討する余地があると考える．

7.2 エスケープ解析

エスケープ解析は，メソッドや関数の処理中に外のスコープに参照がエスケープするかを解析によって求める方法である [16], [17]．エスケープ解析によって参照がそのスコープで閉じていると判定されたオブジェクトは，スタッ

© 2020 Information Processing Society of Japan

ク割付けやスカラ値に置換することが可能であるため，スタック割付けのための解析にも用いられる．

エスケープ解析によってオブジェクトがスタック割付けが可能と判断されるためには，他のスコープから参照可能であってはいけない．それと比較して，リージョンによるメモリ管理では，他のスコープへ参照がわたされる場合においてもリージョン割付けを可能とする．これは，リージョンによるメモリ管理がオブジェクトの生存期間を特定することを目的としていることに起因する．エスケープ解析は参照がエスケープするかどうかを判定する解析であるため，メソッドや関数で解析を閉じることができるが，リージョン推論や解析は，メソッドや関数の呼び出し関係まで検査して解析を行うため，より一般化した手法であるといえる．事実，リージョンによるメモリ管理では，リージョン多相などの仕組みによって，エスケープ解析によるスタック割付けよりも多くのオブジェクトにリージョン割付けを適用することができる．

7.2.1　部分エスケープ解析

GraalVM と呼ばれるフロントエンドを多言語に対応した JVM に，部分エスケープ解析（Partial Escape Analaysis）という実行時最適化手法が実装されている [18], [19]．プログラムの文脈を考慮したエスケープ解析を可能とし，たとえば，メソッド内の特定の分岐でのみエスケープしないようなプログラムを部分的に最適化することができる．また，最適化はしばしば投機的に行われるため，最適化解除の仕組みによってプログラムの実行状態を非最適化時のものに戻すこともできる．

GraalVM 上には TruffleRuby という Java で実装された Ruby 処理系が存在する．TruffleRuby の処理系は部分エスケープ解析が適用されることで，一部のオブジェクトがスタック割付け可能である．しかしながら，部分エスケープ解析は GraalVM の中間表現や GraalVM の機能を前提に実現されている機能であり，メモリ番地の移動が困難である CRuby 処理系に適用することは難しい．

8.　おわりに

リージョンによるメモリ管理を動的言語に適用する手法として，実行時リージョン解析による動的なメモリ割付け最適化を提案し，CRuby 処理系の MJIT を使用してその実装を行い，いくつかのベンチマークによって効果を検証した．マイクロベンチマークにおいては，オブジェクトの割付けガベージコレクションのいらないリージョン割付けへ置き換えられたことにより，ガベージコレクションの機会が削減され，実行時間が短縮されていることを確認した．しかしながら，CRuby 処理系によって提供されているベンチマークではパフォーマンスの向上は見られなかった．その考察として，保守的なリージョン解析ではリージョン割付けの機会が少ないことや，MJIT の他の最適化機構による最適化解除の影響により，限定的なプログラムでのみ有効であることを確認した．

実行時にリージョン解析は，解析結果におけるリージョンの精度を向上させるが，保守的なアプローチによって適用範囲が限定的になることが問題である．改善のためには，実行のプロファイリングやメソッド呼び出しのキャッシュを用いて，楽観的なリージョン割付けを実現する方法が考えられる．ただし，Ruby のプログラミングスタイルがガベージコレクションを前提にされていることもあるため，リージョンによるメモリ管理を Ruby に最適な形で提供するには，プログラムの書き換えがともなう可能性も指摘した．

CRuby 処理系に実装されている実行時コンパイラである MJIT は，2.6 から追加された機能であり，提供されている機能は非常に単純である．また，Rails や Sinatra といったウェブアプリケーションフレームワークのベンチマークでは，MJIT を使用しない場合と比較してパフォーマンスが低下する場合があることも確認されている．実行時リージョン解析は MJIT の仕組みを使用して実装されたため，MJIT の都合により最適化解除せざるを得なくなり，リージョン割付けを実現することが難しい場合も存在した．今後，MJIT の改善によって，それらの機能が改善されることは，リージョン割付けの機会を増加させることにもつながる．

リージョン解析アルゴリズムの正しさや CRuby 処理系およびライブラリとの完全な互換性については，簡単な議論しか行えなかった．YARV の命令セットの意味が形式的に定義されていないことや，公開されているライブラリすべてに対して互換性を検査することは現実的ではないためである．それらに関する具体的な議論や，解析や最適化を改善してリージョン割付けの機会を増加させることは本研究の課題である．

参考文献

[1] Tofte, M. and Talpin, J.-P.: Region-based memory management, *Information and Computation*, Vol.132, No.2, pp.109–176 (1997).

[2] Tofte, M. and Talpin, J.-P.: Implementation of the typed call-by-value λ-calculus using a stack of regions, *Proc. 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.188–201, ACM (1994).

[3] Tofte, M. and Birkedal, L.: A region inference algorithm, *ACM Trans. Programming Languages and Systems* (*TOPLAS*), Vol.20, No.4, pp.724–767 (1998).

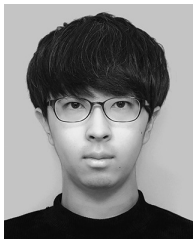
[4] Cherem, S. and Rugina, R.: Region analysis and transformation for java programs, *Proc. 4th International Symposium on Memory Management*, pp.85–96, ACM (2004).

[5] Ruby programming language, available from ⟨https://www.ruby-lang.org/⟩.

[6] Python programming language, available from

© 2020 Information Processing Society of Japan

⟨https://www.python.org/⟩.
[7] Ecmascript® 2017 language specification, available from ⟨http://www.ecma-international.org/publications/standards/Ecma-262.htm⟩.
[8] Nagata, A., Kobayashi, N. and Yonezawa, A.: Region-based memory management for a dynamically-typed language, *Asian Symposium on Programming Languages and Systems*, pp.229–245, Springer (2004).
[9] Cartwright, R. and Fagan, M.: Soft typing, *Proc. ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, PLDI '91*, pp.278–292, ACM (1991).
[10] 笹田耕一, 松本行弘, 前田敦司, 並木美太郎ほか：Ruby 用仮想マシン yarv の実装と評価，情報処理学会論文誌プログラミング（PRO），Vol.47, No.SIG2(PRO28), pp.57–73 (2006).
[11] Hölzle, U., Chambers, C. and Ungar, D.: Debugging optimized code with dynamic deoptimization, *ACM Sigplan Notices*, Vol.27, pp.32–43, ACM (1992).
[12] Cousot, P. and Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, *Proc. 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp.238–252, ACM (1977).
[13] Tofte, M.: A brief introduction to regions, *ACM SIGPLAN Notices*, Vol.34, No.3, pp.186–195 (1999).
[14] Ren, B.M. and Foster, J.S.: Just-in-time static type checking for dynamic languages, *Proc. 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '16*, pp.462–476, ACM (2016).
[15] Corry, E.: Optimistic stack allocation for java-like languages, *Proc. 5th International Symposium on Memory Management*, pp.162–173, ACM (2006).
[16] Blanchet, B.: Escape analysis: Correctness proof, implementation and experimental results, *Proc. 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.25–37, ACM (1998).
[17] Salagnac, G., Yovine, S. and Garbervetsky, D.: Fast escape analysis for region-based memory management, *Electronic Notes in Theoretical Computer Science*, Vol.131, pp.99–110 (2005).
[18] Stadler, L., Würthinger, T. and Mössenböck, H.: Partial escape analysis and scalar replacement for java, *Proc. Annual IEEE/ACM International Symposium on Code Generation and Optimization*, p.165, ACM (2014).
[19] Graalvm, available from ⟨https://www.graalvm.org/⟩.
[20] Hallenberg, N., Elsman, M. and Tofte, M.: Combining region inference and garbage collection, *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, PLDI '02*, pp.141–152, ACM (2002).
[21] Elsman, M. and Hallenberg, N.: Combining region inference and generational garbage collection, Technical Report, Technical Report 2019/01, Department of Computer Science, University of Copenhagen (2019).



齋地 崇大　（正会員）

2018 年筑波大学情報学群情報科学類卒業．2020 年同大学大学院システム情報工学研究科コンピュータサイエンス専攻修了．修士（工学）．2020 年クックパッド株式会社入社．

前田 敦司　（正会員）

慶應義塾大学大学院理工学研究科数理科学専攻単位取得退学．博士（工学）．電気通信大学大学院情報システム学研究科助手，筑波大学電子・情報工学系講師等を経て現在，筑波大学システム情報系准教授．研究分野はプログラミング言語処理系，構文解析アルゴリズム，ランタイムシステム，動的資源管理．ACM，日本ソフトウェア科学会各会員．

© 2020 Information Processing Society of Japan