Ruby プログラムを高速に実行するための処理系の開発

笹田　耕一 †,††

オブジェクト指向スクリプト言語 Ruby はその利用のしやすさから世界的に広く利用されている．しかし，現在の Ruby 処理系の実装は単純な構文木をたどるインタプリタであるため，その実行速度は遅い．これを解決するためにいくつかのバイトコード実行型仮想マシンが提案・開発されているが，Ruby のサブセットしか実行できない，実行速度が十分ではないなどの問題があった．この問題を解決するため，発表者は Ruby プログラムを高速に実行するための処理系である *YARV (Yet another Ruby VM)* を開発している．本処理系はスタックマシンとして実装し，効率よく実行させるための各種最適化手法を適用する．本稿では，プログラミング言語 Ruby の，処理系実装者から見た特徴を述べ，それを実装するための各種工夫，最適化手法の考察およびその実装方法について述べる．また，本処理系の性能についてのベンチマーク結果を示す．

Development of a High-speed Ruby interpreter

Koichi Sasada †,††

Ruby - an Object-Oriented scripting language - is used world-wide because of its ease to use. However, the current interpreter is slow. To solve this problem, some virtual machines were developed, but they didn't have enough performance or functionality. With this background, I have developed a ruby interpreter called *YARV (Yet Another Ruby VM)*. YARV is based on a stack machine architecture. YARV has some optimizations for high speed execution of ruby programs. In this paper, I describe the characteristics of ruby from the aspect of a ruby interpreter implementor and some methods of implementation and optimization. This paper also shows benchmark results.

1. はじめに

スクリプト言語 Ruby[1),16),17)] は、手軽にオブジェクト指向プログラミングを実現するための種々の機能を持つプログラム言語である．

Ruby の特長は次のような点である[1)]．

- シンプルな文法
- 普通のオブジェクト指向機能 (クラス，メソッドコールなど)
- 特殊なオブジェクト指向機能 (Mixin，特異メソッドなど)
- 演算子オーバーロード
- 例外処理機能
- ブロック付きメソッド呼び出しとクロージャ
- ガーベージコレクタ
- ダイナミックローディング (アーキテクチャによる)
- 移植性の高さ

† 東京農工大学大学院工学教育部
Graduate School Technology, Tokyo University of Agriculture and Technology
†† 日本 Ruby の会 (http://jp.rubyist.net/)
Nihon Ruby-no-Kai

ほかにも，多くの優れたライブラリが利用できることもあげられる．これらの特長により，簡単に，楽しくプログラミングを行うことができる．[20)] これらの特長をもつ Ruby は，世界中で広く利用されており，多くのユーザを抱えるプログラミング言語となっている．たとえば，英語のメーリングリストである ruby-talk ML は一日百通以上もの投稿もあり，活発である．日本でも，最近日本 Ruby の会[24)] が発足され，Rubyist Magazine[23)] を発刊するなど，利用者・開発者コミュニティとしての活動も盛んである．

しかし，現在の Ruby 処理系は，速度的な問題を抱えている[8)]．この原因の一つは，現在の Ruby 処理系は Ruby プログラムをパースした結果生成される構文木を実行時にたどり実行する方式にある．木のトラバースにはその木を評価する関数を再帰呼び出しすることで行っているが，次に示す理由で速度的な問題点あると考えられる．

- 構文木をトラバースするためのコストがかかる
- 知られている仮想機械向け最適化の適用が難しい
- 例外処理の実現にコストがかかる

そこで，筆者は構文木をたどるのではなく，構文木から命令列を生成してその命令列を解釈実行する処理

系 *YARV - Yet Another RubyVM*[13]（以下 YARV）を開発している．YARV は Ruby プログラムを高速に実行することを目的とした仮想機械で，新たに設計した命令セットへ Ruby プログラムをコンパイルし実行する．YARV は スタックマシンアーキテクチャで実装しており，既知の各種最適化を盛り込む．

本稿では，YARV について，その設計と現状の性能を述べる．まず，Ruby 処理系開発における課題を述べ，そのような背景のもとで YARV をどのように実装したかについて述べる．そして，各種最適化手法について述べ，実際にどの程度性能向上があったのか，ベンチマーク結果を示す．最後に関連研究を示し，まとめる．

2. Ruby 処理系開発における課題

本節では Ruby 処理系開発における課題について述べる．

2.1 パ ー サ

Ruby の文法は，利用者にとって自然に感じるようデザインされているが，その反面プログラムによりパースすることが難しい．たとえば，メソッド呼び出しには必要なければ括弧を省略することができるが，形式的な文法として，単純に BNF を記述できるものではない[21]．

2.2 実 行 系

Ruby プログラムを実行するにあたり，とくに処理速度を向上させるための課題になるものについて述べる．

2.2.1 オブジェクト指向機能

Ruby はクラスベースのオブジェクト指向プログラミングを実現する．たとえば，基礎となるメソッド呼び出しは，`recv.method(args)` のように記述されレシーバ `recv` のクラスに定義されている `method` というメソッドを，`args` という引数で評価する．

これを実行するためには，`recv` オブジェクトのクラスについて，`method` メソッドを表現する実体を検索する必要がある．Ruby プログラムの大部分はメソッド呼び出しを行うことによって実現するため，この検索を毎回行うのはオーバヘッドが大きいため最適化が必要である．

2.2.2 オブジェクトモデルとガーベージコレクション

Ruby のプログラムで用いられる値はすべてオブジェクトである．たとえば，プログラミング言語 Java[14] では，整数型のようなプリミティブ型を用意しているが，Ruby ではこのようなものをすべてオブジェクトとして表現される．そのため，整数型同士の加算 `x + y` なども `x.+(y)` というメソッド呼び出しと等価になり，再定義などが可能である．

また，Ruby はガーベージコレクションを標準として備えているため，このオブジェクト管理機能の性能は，処理速度に影響することになる．

```
# (1) Ruby: ブロックの利用例
def method() # メソッド定義
  yield      # ブロックを評価
end
method(){    # ブロック付きメソッド呼び出し
  # ...
}

# (2) Ruby: 繰り返しの例
5.times{|i|
  # ... この処理を 5 回繰り返す
}

; (3) (1) を Scheme ではこう書く
(define (method block)
  (block))
(method (lambda ()
          ; ...
        ))
```

図 1 Ruby におけるメソッド付きブロック呼び出しの利用例と Scheme プログラムとの対比

2.2.3 ブロックつきメソッド呼び出しとクロージャ

Ruby の特長のひとつとして，ブロックつきメソッド呼び出しが挙げられる．これは，メソッド呼び出しに手続きをブロックとして渡す機能である．プログラム言語 Scheme でいえば `lambda` 式によってクロージャを生成し，関数に引数として渡す処理にあたる．Ruby ではこれを文法レベルでサポートしていると言える．ブロックを受け取ったメソッドは，`yield` 式によりそのブロックを評価することが可能である（**図 1**）．

当初は繰り返しを実現するための文法要素として用意されているため，イテレータと呼ばれていたが，遅延評価などその他の用途にも利用できる．

また，クロージャを表現する Proc クラスのオブジェクトを明示的に生成することも可能であり Proc オブジェクトをブロックとしても利用可能である．

繰り返しの各イテレーションごとにこのブロックが評価されるため，ブロック（クロージャ）の起動はできるだけ速いことが望ましい．

2.2.4 動的な実行モデル

Ruby ではさまざまなことが動的に決定されるため Ruby プログラムの静的な解析は困難である．たとえば，クラスの定義，メソッドの定義なども実行時に行われ☆，実行時の再定義などが可能である（**図 2**）．

また，文字列を Ruby プログラムとして実行時に評価する `eval` メソッドがあるため，再定義されないという保証を得ることが難しい．このため，コンパイル時の解析が非常に困難になっている．たとえば，定数

☆ Ruby ではクラス定義文自体も実行文であり，クラス定義文中に任意の Ruby プログラムを記述することができる．

```
class C
  if cond1 then
    # m1 cond1 によっては定義されない
    def m1()
    end
  end
end

# ...

class C
  def m1() # m1 の再定義
  end
end
```

図 2 動的な評価が必要な例

```
begin
  begin
    raise "raise exception!"
  ensure
    ...# この節は例外があってもなくても
       # 必ず実行される（Java の finally）
  end
rescue
  ... # この節で例外をキャッチする
end
```

図 3 Ruby の例外処理機能を用いたプログラムの例

演算のための演算が再定義されないという保証がないため，定数畳み込みなどの処理はできない．

2.2.5 例外処理

Ruby は例外処理機能を持ち，たとえば**図 3** のように記述することができる．

現在の Ruby 処理系は構文木を評価する関数を再帰呼び出しすることによって Ruby プログラムを実行している関係上，その例外処理部分（図 3 における `begin`）で毎回 `setjmp` 関数によりコンテキストを保存し，例外の発生時（図 3 での `raise`）では `longjmp` を実行してマシンスタックの巻き戻しを行う手法により実現している☆．

この方式では，例外発生時のキャッチ部分へのジャンプは軽量に行うことができるが，例外処理部分に突入するごとに `setjmp` によるコンテキストの保存が必要になる．一般的に，例外を発生することは稀であるため，この手法は効率が悪い．例外処理部分への突入には余計なコストがかからないことが望ましい．

2.2.6 Ruby C API

Ruby の特長のひとつに，C 言語用 API が充実しており，C 言語などで作成する Ruby を拡張するためのライブラリ（拡張ライブラリ）を容易に記述が可能ということがある．たとえば Ruby で定義したメソッドを C 言語から呼び出すには，C 言語の関数呼び出しで自然に行えるインターフェースが望ましい．また，Ruby プログラムで記述する例外処理なども，C 言語のインターフェースで自然に表現できることが望ましい．現在の処理系では Ruby C API を用意してこれらの要件をサポートしている．これを利用した拡張ライブラリの書きやすさは定評がある．

これは，Ruby プログラムの評価自体が C の関数の再帰呼び出しとして実現しているため容易に実現できている．しかし，評価関数を再帰呼び出しとして実現すると，余計なコストがかかることがある．

2.2.7 スレッドの対応

Ruby はスレッドの生成・実行をサポートしている．スレッドを実現するにはいくつか方式があるが，たとえば複数 CPU 資源を利用する場合には OS の提供するスレッド機能を使う必要がある．現在の Ruby 処理系は独自でユーザレベルスレッドを提供している．

2.3 Ruby 処理系開発における課題

本節で述べた Ruby 処理系開発における課題，最適化を困難にする要素をまとめると次のようになる．

- パーサ作成が困難 (*)
- オブジェクト指向機能の実現
- プリミティブ型がない (*)
- ブロック・クロージャの実現 (*)
- 静的解析が困難 (*)
- 例外処理機能の実現
- Ruby C API のサポート (*)
- スレッドのサポート

とくに，(*) 印をつけたものは Ruby 特有の課題といえる．

3. YARV の設計と実装

前節で述べた課題を踏まえ，YARV はどのような設計にしており，どのように実装しているかについて本節でまとめる．

3.1 全体構成

YARV の全体像を**図 4** に示す．YARV の行う処理は主に (1) 構文木を YARV 命令セットにコンパイル (2) 命令列を実行 となっている．とくに，(2) についてはまさに処理速度に直結するため，この部分の最適化を多く行っている．また，実行系は命令列を直接実行する評価器と JIT/AOT コンパイルして実行する部分がある．

YARV の実行モデルはシンプルなスタックマシンとしている．YARV 自体は C 言語で実装し，一部後述する命令記述を Ruby で記述した前処理プログラムで処理している．

3.1.1 現在の Ruby 処理系との連携

YARV は，現在の Ruby 処理系の拡張ライブラリとして実現している．YARV により Ruby プログラムを実行する場合，現在の Ruby 処理系によって YARV モジュールをロードし，Ruby プログラムを YARV

☆ 例外処理以外にも，ブロックの実行を中断する break などもこれを利用して実現している．

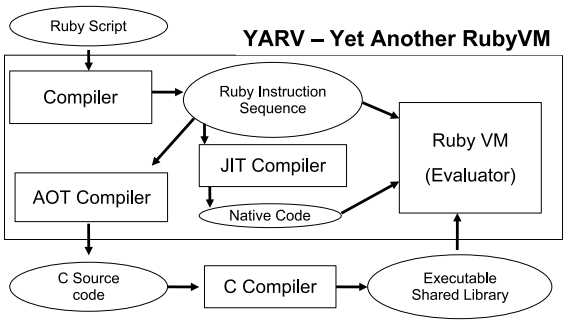

図 4 YARV の全体構成

モジュールに渡すことで行われる．

この手法では，現在の Ruby 処理系の機能を容易に利用できる．具体的には，Ruby プログラムのパーサやオブジェクトモデルの管理（とくにガーベージコレクタ），現在の Ruby 処理系用に開発されたライブラリなどをそのまま利用できる．また，現在の処理系の実行処理部分は利用できるため，実行時にどちらの処理系を利用するかを選択することができる．

これにより，YARV の開発を容易に行うことができる．とくに，評価とテストが容易である．また，Ruby が用意する拡張ライブラリ用 API が利用可能であるため，開発が容易になる．たとえば，YARV プログラム中で Ruby オブジェクトを生成した場合，適当なタイミングでガーベージコレクションを行ってくれる．これを利用して後述するコンパイラの開発が容易になった．

この方式では，YARV のみで実行する場合にくらべ現在の Ruby 処理系との連携のためのスタブコードが必要になり，これを実行するためのオーバヘッドが発生するが，将来的に YARV 実行系のみにすれば解決できる．

3.2 基本命令セット

YARV では，Ruby プログラムを正しく表現するための基本命令セット（**表 1**）を定義した．執筆時現在，基本命令は 43 命令である．また，このほかに最適化用命令も定義する．

たとえば Ruby プログラム `a=recv.method(b)` は，次のようにコンパイルされる．

```
getlocal 2       # ローカル変数 recv を push
getlocal 3       # ローカル変数 b を push
send :method, 1 # 引数 1 でメソッド呼び出し
setlocal 1       # ローカル変数 a に pop & set
```

Ruby にはプリミティブ型のようなものがないので，数値の演算命令などは用意しない☆．また，メソッド定義，クラス定義は実行時に行う必要があるので，命令として用意している．

☆ ただし，後述するように最適化関係の命令として簡単な数値演算用命令などを用意する．

表 1 命令セットのカテゴリ一覧

変数関係	ローカル変数などの値を取得・設定（`getlocal`, `setlocal` など）
値関係	self の値や文字列・配列などを生成（`putself`, `putstring` など）
スタック操作関係	スタック上の値操作（`pop`, `dup` など）
メソッド定義	メソッドを定義（`methoddef` など）
クラス定義関係	クラス・モジュール定義スコープに入る（`classdef` など）
メソッド呼び出し関係	メソッド呼び出しや yield など（`send`, `end` など）
例外関係	例外を実装するために利用（`throw`）
ジャンプ関係	ジャンプ・条件分岐（`jump`, `if`, `unless`）

```
DEFINE_INSN
getlocal
(ulong idx)  /* オペランド          */
()           /* スタックからポップ */
(VALUE val)  /* スタックへプッシュ */
{            /* 具体的な処理の記述 */
  val = *(GET_LFP() - idx);
}

DEFINE_INSN
setlocal
(ulong idx)  /* オペランド          */
(VALUE val)  /* スタックからポップ */
()           /* スタックへプッシュ */
{            /* 具体的な処理の記述 */
  (*(GET_LFP() - idx)) = val;
}
```

図 5 命令記述の例

3.2.1 命令記述

基本命令は命令記述フォーマットを定義して，それに基づいて記述した．具体的な処理の部分は C 言語で記述するが，利用するオペランド，スタックからポップする値，命令終了後スタックへプッシュする値を，それぞれ C 言語の変数のように宣言する（**図 5**）．

図 5 を見ると，ローカル変数をスタックへプッシュする命令 `getlocal` はオペランドとしてローカル変数インデックスを必要とするが，それを `idx` として宣言している．命令開始時，スタックからポップする値はないので，この部分は空となり，命令終了後スタックへローカル変数の値をプッシュするので，`val` として宣言している．スタックからポップした値をローカル変数へセットする命令 `setlocal` では逆にスタックからポップするものを `val` として宣言し，スタック上に何も乗せないためプッシュするものは空である．

処理記述部分では，ここで宣言した変数名を利用して C 言語で処理を記述する．

YARV をビルドするときには，この命令記述を専用のパーサを通すことで実行可能な C プログラムに変換する（**図 6**）．

このような記述を行うことで，各機能を自動生成す

```
INSN_ENTRY(getlocal){
{
  VALUE val;
  ulong idx = GET_OPERAND(1);
  ADD_PC(1+1); /* PC を 2 進める */
{
  val = *(GET_LFP() - idx);
  PUSH(val);
  END_INSN();
}}}
INSN_ENTRY(setlocal){
{
  ulong idx = GET_OPERAND(1);
  VALUE val = TOPN(0);
  ADD_PC(1+1); /* PC を 2 進める */
  POPN(1);
{
  (*(GET_LFP() - idx)) = val;
  END_INSN();
}}}
```

図 6 命令記述を変換した結果の例

ることができる．具体的には，コンパイラ，命令列変換による最適化器，逆アセンブラ，コードベリファイア，命令列評価器など（の一部）を自動生成することができる．また，後述する命令融合やスタックキャッシングを適用することが容易になる．

3.3 コード生成部分

Ruby プログラムのパースは前述したように，現在の Ruby 処理系のものを用いる．コンパイラは，パース結果として得られる構文木を，前項で述べた命令セットの命令列に変換する．変換には，命令記述を解析した結果を利用する．

ここでは変換のほかに簡単なピープホール最適化を行う．また，後述する命令融合などの命令（列）変換処理も行う．

コンパイルした結果は，1 命令（もしくは 1 オペランド）をそのマシンのポインタ型の大きさで表現したものの配列として返す．つまり，一般的にはバイトコードなどと呼ばれている仮想機械の命令は，YARV においてはワードコードといえる．

これは，1 バイトを単位として命令列を表現すると，マシンによってはアクセス速度が不利になる可能性があること（アラインメントの問題）と，オペランドに Ruby オブジェクトを埋め込むのが容易であるからである☆．

3.4 命令列評価器

命令列評価器はコンパイルされた命令列を実行する部分であり，YARV の中心的な機能である．

YARV はスタックマシンとして構成され，以下のレジスタを持つ仮想マシンである．

PC プログラムカウンタ
SP スタックポインタ
CFP フレーム制御ポインタ
LFP メソッドローカル環境ポインタ
DFP ブロックローカル環境ポインタ

PC は実行している命令の位置を示す．SP は，各スレッドがそれぞれひとつ持つスタックのトップを指す．CFP は，現在実行中の制御フレーム（フレームについては後述）を指す．

LFP，DFP は現在実行中の環境を指す．それぞれ，メソッドローカル変数を格納する環境，ブロックローカル変数を格納する環境へのポインタである．後者は，より上位の環境へのポインタをたどることができ（後述する PDFP），最終的にはメソッドローカル環境をたどることは可能である．そのため，Lisp などの処理系のように，環境へのポインタはひとつしか用意しない選択肢もあるが，Ruby のの特質☆☆からそれぞれ用意した．たとえば，ブロックの深いところでメソッドローカル変数を参照しようとしたとき，LFP による間接アクセスを行い定数時間でアクセスすることが可能である．

実行は，ある命令列について，PC の指す命令を取り出し，その命令を実行し，プログラムカウンタを進めるということの繰り返しになっている．このループをひとつの評価器関数内での繰り返しとして実現している．場合によっては，この関数を再帰呼び出しすることが可能である．

たとえば，Ruby のあるメソッド `m1_ruby` が C で記述されたメソッド `m2_c` を呼び出し，`m2_c` 中で Ruby で記述されたメソッド `m3_ruby` を呼ぶような例では，この評価器関数が再帰呼び出しされることになる．

執筆時現在，YARV 命令セットのほぼすべてを実行可能であり，Ruby プログラムの大部分を動作させることができる．

3.4.1 スタックフレーム

メソッド呼び出し（`send` 命令など），クラス・モジュール定義（`classdef` 命令など），ブロックの起動（`yield` 命令など）などは，それぞれ変数スコープを変える必要があるため，スコープを管理するためにそれぞれスタックフレームを作成する．メソッド呼び出しとクラス・モジュール定義用フレームはほぼ等しいので後者の説明は省略する．

メソッドフレーム（**図 7**）はメソッド呼び出し時に作成される．メソッドローカル変数領域をスタック上に確保し，必要なら初期化する．その上でフレーム制御情報（渡されたブロック，レシーバ（図中の `self`），実行する命令列の情報（図中の `iseq`），退避するレジスタ（継続情報））をスタックに積み，CFP はこれを指す．LFP，DFP はともにメソッドローカル変数領域を指す．

☆ Ruby オブジェクトの表現もポインタ型と同様のサイズである．

☆☆ とくに，次期メジャーバージョンアップである Ruby 2.0 では，ブロックローカル変数がさらに現れにくくなる．

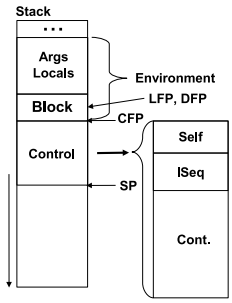

図 7 メソッドフレーム

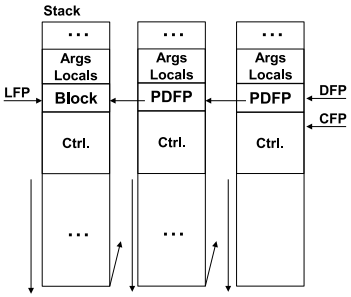

図 8 ブロックフレーム

ブロックフレーム（**図 8**）はブロックの起動時に作成される．ブロックローカル変数領域をスタック上に確保し，必要なら初期化する．ブロック情報として渡される，以前の DFP を PDFP として積む．そしてフレーム制御情報をスタックに積み，CFP はこれを指す．DFP は確保したブロックローカル変数領域を指し，LFP はブロック情報として渡される LFP を設定する．

クロージャ（Ruby では `Proc` オブジェクト）を作成するときには，LFP，DFP が指す環境をヒープに保存する．また，フレームを遡って以前の環境を指していたポインタをヒープ上の環境に張りなおす．

各フレームは，継続情報を見ることで呼び出し元フレームへとたどることができ，後述する例外処理やバックトレースの作成を行うことが可能になる．

Ruby C API で Ruby のメソッドを呼び出した場合，命令実行関数を再帰することになるが，継続情報を積むことで，呼び出し元より以前のスタックフレームをたどることができるようにしている．

3.4.2 例外処理

YARV では各スコープ（命令列）ごとに例外処理用テーブルをコンパイル時に用意することで実現した．この方式は Java 仮想マシン[18] などとほぼ同様で，例外が発生したとき，この表を参照することで例外処理をどのように行うべきかどうかを知ることができる．また，Ruby の大域ジャンプする式（`retry` など）もこの機構を利用する．

例外発生時，スタックを巻き戻し，キャッチする部分が見つかるまでスタックフレームごとにこの表を検査することを繰り返すため，現在の Ruby 処理系よりも不利になるが，例外が発生しない場合，実行コストがかからない．殆どの例外処理部分では例外は発生しないため，本方式のほうが有利である．

ただし，Ruby では例外処理構文なども式となり，値を持つことが可能な点や，エラーを受け取る方法の違いから，単純に Java 仮想マシンなどと同様の仕組みを利用することはできず，少々の工夫を要した．

また，Ruby C API での例外の生成などをサポートするため，評価器関数の最初で `setjmp` し，これに備える．Ruby C API により例外が発生したとき，このポイントで例外処理を行うことができる．その評価器関数内で，仮想機械スタックフレームをそれ以上遡れない場合，`longjmp` してマシンスタックの上位に例外発生を伝播する．

3.4.3 スレッドの対応

YARV ではスレッドについてのサポートは現在行っていないが，OS の提供するスレッドを適用することができるように設計を行っている．ただし，これについてはオブジェクトの管理部分，とくにガーベージコレクタなどの対応が不可欠であるため，即座に対応することはできない．

3.5 そ の 他

コンパイル済みの命令列を永続化する機能，またそれをロードし実行する機能を組み込む予定である．これにより，実行時にコンパイルのコストを削減することができる．たとえば，利用頻度の高いライブラリなどをあらかじめコンパイルしておくことで，起動時のオーバヘッドを削減可能である．

また，コードベリファイアも用意し，仮想機械にとって危険なコードを判別できるようにする予定である．たとえば，ネットワーク経由でコンパイル済みの Ruby プログラムをダウンロードして実行するときなど，この機能が必要になる．

4. YARV での最適化

本節では YARV における，実行最適化手法について述べる．本処理系開発において，一番楽しい部分であり，もっとも大きなモチベーションのひとつである．

現段階で，すでに実装したものと設計段階のものがある．

4.1 命令ディスパッチ

命令を実行するためのループをどのように実装するかはいくつか方法があるが，GCC[7] ではラベルを値として用いることができるため，これを利用した Direct Threaded Code[3] を用いオーバヘッドを削減する．また，間接ジャンプを行うマシン命令番地が異なる場所で分岐することになるため，分岐予測精度の向上が期

```
Ruby Program:
  Const # 定数 Const を参照

Compiled YARV Instruction sequence
l1:
  # インラインキャッシュを見て，ヒット
  # ならばその値を取り出し，l2 へジャンプ
  getinlinecache l2, vm_cnt, cached
  getconstant :Const       # 定数アクセス
  # l1 にある命令に値をキャッシュ
  setinlinecache l1
l2:
```

図 9 定数アクセスで利用するインラインキャッシュ命令

待できる[5]．

GCC 以外のコンパイラでは C 言語 switch 文による命令ディスパッチを行う．

4.2 インラインキャッシュ

メソッドディスパッチでは，レシーバオブジェクトのクラス，そしてその親クラスへとメソッドの実体が見つかるまで検索をする必要があるが，この検索をメソッドディスパッチのたびに行うのはコストが高すぎる．現在の Ruby 処理系では，グローバルメソッドキャッシュ[17),21)] を用いてこのコストを軽減している．

YARV では，従来のグローバルメソッドキャッシュ以外に，命令自体にキャッシュする値を埋め込むインラインメソッドキャッシュを用意する．

また，Ruby では定数の参照には，特定の検索パスにより検索しなければならない．これは，動的な文法で定数をレキシカルに参照するための言語的要求であるが，頻繁に変更することはないため，毎回検索を行うのは無駄である．そこで，一度アクセスした定数はインラインキャッシュとして保存しておき，可能であれば次にこの命令を実行したときにキャッシュした値を定数値として返す（**図 9**）．

インラインキャッシュの基本的なアイデアは，仮想マシンに状態カウンタを設けるという点にある．キャッシュしてある値が有効であるかは，キャッシュと一緒に格納している状態カウンタの値と，現在の仮想マシンの状態カウンタの値が一致していれば正しいものであるとわかる．

状態カウンタはメソッドの定義や定数の定義などが行われたときに 1 増加する．これらの定義の頻度は基本的にまれであるため，キャッシュには十分ヒットする．この方式を用いることで，再定義時に必要になるキャッシュのクリアについての問題が解決する．

インラインメソッドキャッシュにおいては，他にもキャッシュと一緒に格納しているクラス情報を確認し，これがレシーバのクラスと一致すれば，キャッシュヒットということになる．

4.3 特化命令

Ruby にはプリミティブ型が無いため，すべての演算はメソッド呼び出しと等価であるが，たとえば整数

```
opt_plus:
  if(a と b は整数)
    if(整数についてのメソッド + が
       再定義されていない)
      return a + b
  return 通常のメソッド呼び出し（a.+(b)）
```

図 10 最適化用命令 opt_plus の擬似コード

加算のためにメソッドフレームの構築などを行うのは無駄である．そのため，YARV では特定のセレクタ（メソッド名），特定の引数の数（二項演算子ならば引数の数は 1）の場合，特別な命令にコンパイル時に変更する．この特別な命令を特化命令という．

たとえば，Ruby の式 `a+b` は，`a.+(b)` と同様であるが，このとき通常のメソッド呼び出し命令ではなく `opt_plus` という命令にコンパイルする☆．

`opt_plus` の実行は，まずレシーバと引数（この場合は `a` と `b`）が整数であるかどうかを確認する．もしそうであれば今度は整数同士の加算のメソッドが再定義されていないか確認する．もしされていなければ，整数加算をした結果をスタックに積む．そうでない場合は，通常のメソッド起動処理に移行する（**図 10**）．

利用頻度の高い（そしてメソッド起動のコストが気になるような）特定のメソッドは，このような工夫をすることでメソッドとしての一般性を失うことなく高速に実行することが可能になる．

4.4 オペランドの融合と命令融合

ある命令において，ある特定のオペランドを頻繁に利用する場合，融合して新しい命令を作ることによってオペランドフェッチのコストを削減し，処理速度が向上する．たとえば，命令 `A` がオペランド `x` を頻繁に利用する場合，命令 `A` をオペランド `x` に固定した `A_x` という新しい命令を作る．

また，ある n 個の命令の並びが頻出する場合，それらを融合し新しい命令を作ることで，命令ディスパッチ，スタックの遷移やプログラムカウンタの操作のコストを削減することができる．たとえば，命令 `C` のあとで命令 `D` が現れるような場合が頻繁にある場合，`C` と `D` の機能を実行する `C_D` という命令を新しく作る[22)]．

YARV では，どのオペランドを融合するか，どの命令列をひとつの命令に融合するかを指定すれば，自動的に該当の命令を作成し，コンパイル時にこの命令に変換するための最適化器を生成する☆☆．これは，命令記述を解析することで実現する☆☆☆．

これら融合操作によるインタプリタの最適化は文献 22) でとくに述べられている最適化手法である．YARV

☆ このとき，a，b のクラスは実行時に判定するためコンパイル時に静的に解析する必要は無い．

☆☆ 本質的ではないが，命令のディスアセンブラもこの拡張命令に簡単に対応する．

☆☆☆ 執筆時点ではまだ命令融合には対応していない．

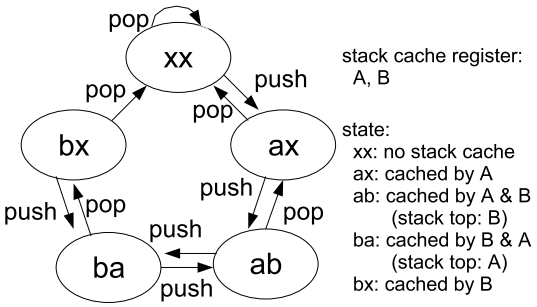

図 11 スタックキャッシングの状態遷移図

ではこの融合操作を自動的に行うことが可能であるため，容易に最適化を行っていくことができる．

将来的には，プロファイリング（利用統計と性能統計）と融合操作の繰り返しを自動で何度も行うことにより，あるプログラムセットにおける最適な命令セットの生成が可能ではないかと考えている☆．

4.5 スタックキャッシング

スタックマシンの最適化にスタックキャッシング[4]がある．スタックトップをいくつかキャッシュすることで，メモリへの書き込みやスタックポインタ操作などをある程度省略できる．本手法の詳細は文献 4) を参照されたい．

YARV では 2 個のキャッシュレジスタを利用して 5 状態で静的スタックキャッシングを行う．この条件での YARV で採用したスタックキャッシュの状態遷移図について**図 11** に示す．各命令ごとに，スタックキャッシュの各状態に応じた命令が生成される．YARV では 5 つの状態をもつため，命令数は基本命令数と最適化用命令数を足した数の 5 倍になる（現状では 300 命令ほど）．これらの各状態用の命令は，命令記述を解析することで基本命令から自動的に生成する．コンパイラ部分では，コンパイル終了後の命令列を対象として，命令記述解析結果を利用してスタックキャッシュを行う命令列に変換する．

キャッシュするための変数がマシンレジスタに割り付けられるかどうかは，マシンやコンパイラによるが，この変数がマシンレジスタに割り付けられれば，単純な命令列ではスタック操作がすべてキャッシュ上で行われるため，高速な動作が可能になる．

4.6 プロファイラ

上記の最適化を行うため，YARV 実行時の命令の利用頻度やオペランドの利用状況，レジスタへのアクセス頻度，命令の連結度などの情報を集計するプロファイラを用意した．

オペランドの利用状況を見ることでオペランド融合の有効性を判断できる．また，命令の連結度は，どの命令を連続して行ったかを見ることで，命令統合を行う基準を知ることができる．

このプロファイラ自身は実行時コストが高いものであるため，実行時にこのプロファイル結果を利用した最適化を行うのは現実的ではないが，将来的には集計する情報を選別し軽量化することで，リアルタイムに更新される統計情報を利用した実行時最適化を行いたいと考えている．

4.7 ネイティブコンパイラ

マシンコードに変換するネイティブコンパイラの実現手法にはいくつかあるが，JIT（Just-In-Time）コンパイラ，および AOT（Ahead-Of-Time）コンパイラに大別できる．YARV では両者実装する予定である．

JIT コンパイラは，すべての命令のアセンブル記述を用意するには実装コストが問題になるため，Dynamic Replication[5]，Selective Inlining[11] などの手法を利用するなどを考えている．

AOT コンパイラは，命令列を命令記述から C 言語のならびに変換し，これを拡張ライブラリとしてコンパイルすることで，Ruby 側から利用する（Ruby プログラム → YARV 命令列 → C プログラム → Ruby 拡張ライブラリ）．この方式で Ruby の機能を損なうことなく，C コンパイラの最適化機能により十分な性能をもつプログラム変換が可能になると考えている．

5. 性能評価

本節では執筆時現在の YARV の処理速度を現在の Ruby 処理系と比較し，YARV の性能を示す．

5.1 評価環境

評価は CPU：Intel Celeron 1.7GHz，搭載メモリ：512MB，OS：Microsoft Windows2000，Cygwin 1.5.12，コンパイラ：gcc (GCC) 3.3.3 を利用して行った．比較対象（および，YARV を拡張ライブラリとしてロードする）Ruby 処理系は ruby 1.9.0 (2004-11-30) [i386-cygwin] を利用した．

gcc の最適化オプションは `-O2` を指定した．また，命令列評価器をコンパイルするときには `-fno-crossjumping` オプションも追加した．このオプションは同一命令列をまとめ，コード量を節約する最適化を無効にするオプションである．YARV の評価器は，各命令ごとに多くの共通部分があるため，この最適化を有効にする（`-O2` ではデフォルトで有効になる）とコード量削減を行うことができるが，その反面余計なジャンプ命令が挿入され，YARV 1 命令を実行するために必要な機械語命令数が増加する．また，ジャンプ命令は最近のプロセッサにとってパイプライン擾乱を起こす原因にもなり，性能低下につながる．

☆ テストプログラムが簡単な例では基本ブロックをすべて融合するだけで終わるかもしれないが，テストで利用するプログラムのセットがある程度大きな量になれば，さまざまなトレードオフが生じるはずで，このトレードオフの不動点を発見し，理想的な命令セットが発見できるかもしれない．

表 2 ベンチマーク一覧

プログラム名	処理内容
whileloop	while による繰り返し
times	times によるブロックの繰り返し
const	定数アクセス
method	空のメソッド呼び出し
block	ブロック付きメソッド呼び出し
rescue	例外処理（例外は発生しない）
rescue2	例外処理（例外は発生を発生）
fib	フィボナッチ数の計算（再帰） fib(32)
tak	たらいまわし関数 tak(18, 9, 0)
tarai	たらいまわし関数 tarai(12, 6, 0)
so_matrix	行列の乗算
so_sieve	素数の計算
so_ackermann	アッカーマン関数 ack(3, 7)
so_count_words	ファイル中の文字列を数える

表 3 ベンチマークの実行結果

プログラム名	Ruby(sec)	YARV(sec)	Speedup
whileloop	7.66	0.69	11.10
times	6.47	3.12	2.07
const (*a)	1.90	0.22	8.48
method (*a)	7.25	0.74	9.83
block (*a)	13.76	1.60	8.58
rescue (*a)	3.10	0.02	131.87
rescue2 (*b)	5.32	3.17	1.69
fib	7.73	0.79	9.82
tak	25.07	3.42	7.33
tarai	20.26	4.13	4.91
so_matrix	6.37	2.62	2.43
so_sieve	3.56	0.73	4.88
so_ackermann	3.06	0.27	11.55
so_count_words	0.63	0.61	1.03

(*a) 10,000,000 回の繰り返し (*b) 100,000 回の繰り返し

5.2 ベンチマークプログラムとその実行結果

利用したベンチマークプログラムを**表 2** に示す．プログラム名の先頭に `so_` とあるものは，文献 8) からプログラムを拝借した．計算時間が短いものは，その処理を繰り返して計測した．

適用した最適化は Direct Threaded Code，特化命令（整数演算 `+`，`-`，`<`），オペランド融合である．

実行結果を **表 3** に示す．表中 (a)，(b) のプログラムは繰り返しのための実行時間は除いてある．

おおむね高速化されているが，とくに `rescue` は速度向上率が著しい．例外処理を表引きにし，例外処理部分に入るコストを無くしたためである．それに比べ，`so_count_words` や `rescue2` などはあまり高速化できていない．これは文字列操作など，ライブラリによる処理が計算時間を多く消費しているからである．

`times` による繰り返しが，`whileloop` と比較して非常に遅いが，これはブロックの起動のコストが大きいことを示している．この問題を解決するために，今後ブロックインライニングなどの最適化を検討していく．

表 4 スタックキャッシングを有効にした処理速度

	YARV(sec)	YARV SC(sec)	Speedup
whileloop	0.69	0.40	1.72
fib	0.79	0.56	1.41
so_sieve	0.73	0.61	1.19

5.3 スタックキャッシング

スタックキャッシングを有効にしたときのベンチマーク結果を**表 4** に示す．執筆時現在，まだ動かない機能があるため，プログラムを 3 つにしぼり，その結果を記載した．表中の Speedup とは，この最適化を適用前と比較して，適用後の速度向上率を示している．

`whileloop` は，特化命令により簡単な命令に置換されている．これがさらにスタックキャッシング最適化を適用することでかなりの速度向上を達成できた．

6. 関 連 研 究

Ruby 向け仮想マシンはいくつか提案されている[12]が，ほとんどのプロジェクトは Ruby 処理系としての機能が不十分，性能が現在の処理系よりも遅い，またはプロジェクト自体が消滅いているなどの状態である．

ruby2c[2] は Ruby プログラムをパースして S 式を生成し，それを C 言語プログラムへ変換する．YARV でも AOT コンパイラとして同様のことを実装する予定だが，コンパイル対象が YARV 命令セットを利用可能な点で異なる．

rubydium[9] は YARV と同様，速度向上を目指した Ruby 処理系である．最適化については，開発者である Kellett 氏はブロックを命令ディスパッチ時にインライン化して実行する方式を提案している．YARV でもブロックのインライン化は検討しているが，筆者は呼ばれた側がこれを行う方式を検討している．

vmgen[6] は YARV と同様，命令記述を特定のフォーマットで行い，これを利用して仮想機械の C プログラムを自動生成する，汎用的な仮想機械生成系である．また，複数の命令を組み合わせた命令を自動生成する機能も有し，どの命令を最適化するかを判断するためのプロファイラもある．しかし，vmgen ではオペランド融合やスタックキャッシング用の命令を自動生成する機能はない．

文献 22) では，Scheme インタプリタを例に，融合操作によって体系的に命令を追加し，仮想機械の最適化を行う方法を述べている．YARV ではこの手法を Ruby に適用している．また，YARV ではこれらの融合操作を自動化する仕組みを備えているため，この手法を適用が容易に適用可能である．

JavaVM[18] や .NET[10]，Parrot[15] などの既存の仮想機械を利用する方法もある[19]．この利点は，すでに実装された JIT コンパイラのような最適化器などをそのまま利用できることである．しかし，Ruby の

モデルと微妙に違いがあり，この差を埋めるためのコストが生じる．また，YARV では Ruby に特化した最適化が可能である．

7. まとめ

本稿では Ruby プログラムを高速に実装するための処理系である YARV について，その設計と実装方法，最適化手法の概要，そしてその性能について述べた．

現状での YARV の最適化でも，ベンチマーク結果により，現在の Ruby 処理系と比較して高速に実行できることを示した．

執筆時現在，まだ Ruby の機能を完全にはサポートしていないため，早急にこの対応を行う．また，いくつかの最適化手法の実装も未着手であるため，これらについて随時行っていく．とくに，JIT・AOT コンパイルは大きな性能向上が期待できるため積極的に実装していく予定である．

今後，本処理系の品質をさらに向上させることで，Ruby によるプログラミングをより快適なものにしていきたい．

謝　辞

YARV 開発にあたり，Ruby 開発者であるネットワーク応用通信研究所のまつもとゆきひろ氏をはじめ，YARV 開発用メーリングリストに参加されている方々には有益なアドバイスを頂いております．感謝いたします．また，いつもお世話になっている筆者の所属する研究室の担当教員である東京農工大学大学院の並木美太郎助教授に感謝します．

本処理系の開発プロジェクトは，IPA（情報処理推進機構）の公募事業 2004 年度未踏ソフトウェア創造事業「未踏ユース」（プロジェクト・マネージャ：筧 捷彦早稲田大学教授）に採択され，支援を受けています．

参考文献

1) : オブジェクト指向スクリプト言語 Ruby. http://www.ruby-lang.org/ja/.
2) Davis, R. and Hodel, E.: ruby2c Automatic translartion of ruby code to C. http://zenspider.com/ ryand/Ruby2C.pdf.
3) Ertl, A.: Threaded Code. http://www.complang.tuwien.ac.at /forth/threaded-code.html.
4) Ertl, M. A.: Stack caching for interpreters, *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, ACM Press, pp. 315–327 (1995).
5) Ertl, M. A. and Gregg, D.: Optimizing indirect branch prediction accuracy in virtual machine interpreters, *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, ACM Press, pp. 278–288 (2003).
6) Ertl, M. A., Gregg, D., Krall, A. and Paysan, B.: vmgen: a generator of efficient virtual machine interpreters, *Softw. Pract. Exper.*, Vol.32, No. 3, pp. 265–294 (2002).
7) Free Software Foundation (FSF): GCC Home Page. http://gcc.gnu.org/.
8) Fulgham, B.: The Computer Language Shootout Benchmarks. http://shootout. alioth. debian.org/.
9) Kellett, A.: rubydium. http://rubyforge.org/ projects/rubydium/.
10) Microsoft: Microsoft .NET Information. http:// www.microsoft.com/net/.
11) Piumarta, I. and Riccardi, F.: Optimizing direct threaded code by selective inlining, *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, ACM Press, pp. 291–300 (1998).
12) RubyGarden: RUBY: VirtualMachineOptions. http://www.rubygarden.org/ruby? VirtualMachineOptions.
13) Sasada, K.: YARV: Yet Another Ruby VM. http://www.atdot.net/yarv/.
14) Sun Microsystems: Java テクノロジ. http:// jp.sun.com/java/.
15) The Perl Foundation: Parrot - parrotcode. http://www.parrotcode.org/.
16) Thomas, D., Fowler, C. and Hunt, A.: *Programming Ruby*, The Pragmatic Programmers (2004).
17) まつもとゆきひろ, 石塚圭樹: オブジェクト指向スクリプト言語 Ruby, 株式会社アスキー (1999).
18) ティム・リンドホルム, フランク・イェリン: Java 仮想マシン仕様第 2 版, ピアソン・エデュケーション (2001).
19) 淺川浩紀: Ruby.NET コンパイラの開発. 2004 年度未踏ソフトウェア創造事業（未踏ユース）採択概要（http://www.ipa.go.jp/jinzai/esp/2004youth/ gaiyou/2-15-18.html）.
20) 松本行弘: Ruby の真実, 情報処理, Vol. 44, No. 5, pp. 515–521 (2003).
21) 青木峰郎: Ruby ソースコード完全解説, インプレス (2002).
22) 前田敦司, 山口喜教: Scheme インタプリタにおける仮想マシンアーキテクチャの最適化, 情報処理学会論文誌 (PRO), Vol. 44, No. SIG13, pp. 47–57 (2003).
23) 日本 Ruby の会: Rubyist Magazine. http:// jp.rubyist.net /magazine/.
24) 日本 Ruby の会: 日本 Ruby の会 Wiki. http://jp.rubyist.net/.