RubyVMの設計と実装

木山 真人*

1 はじめに

スクリプト言語は現在幅広く普及しており，その重要性が指摘されている [1, 2]．また，Python，Ruby[3] といったスクリプト言語はオブジェクト指向機能を有しているため，生産性だけではなく，保守性も高い．そのため，オブジェクト指向スクリプト言語は文字列処理など規模の小さなプログラムだけではなく，Web アプリケーションなど規模の大きなプログラムに使用され始めている．このことから，オブジェクト指向スクリプト言語は今後ますます普及し，様々な用途で使用されると考える．

しかし，オブジェクト指向スクリプト言語には幅広い用途での使用を妨げる問題がある．それは，オブジェクト指向スクリプト言語の処理速度である．オブジェクト指向スクリプト言語の多くはインタプリタ方式で実装されているため処理速度が遅く，高速な処理を必要とするアプリケーション開発には向いていない．処理速度の問題を解決し，オブジェクト指向スクリプト言語の幅広い用途での使用を促進させるため，オブジェクト指向スクリプト言語の高速化が重要となる．

高速化の対象として，オブジェクト指向スクリプト言語の 1 つである Ruby を選択する．Ruby には，変数や式に型がない，整数などの基本的なデータ型をはじめとしてすべての値がオブジェクトなどの特徴がある．

現在，Ruby はプログラムを構文木に変換して実行している．プログラムを実行するたびに構文木を構築し，構文木をたどりながら解釈実行を行う．構文木の解釈は、解釈関数を再帰的に呼び出すことで実現されている．そのため，この再帰的に関数を呼び出すことがボトルネックになると考えられる．

そこで，Ruby を高速化するため，プログラムをバイトコードに変換して実行する方式にすることを考える．本論文では，Ruby をバイトコードで実行する場合に必要となる仮想マシン（Virtual Machine；VM）の設計と実装について述べる．

2 VMの利点

プログラムを計算機上で実行する方式は，大きく分けて 2 つある．コンパイル方式とインタプリタ方式である．コンパイル方式は，プログラムを機械語に変換して実行する方法である．インタプリタ方式は，プログラムを中間コードに変換して実行する方法である．中間コードは，構文木と VM のバイトコードの 2 つがある．

Ruby はプログラムをインタプリタ方式で実行する．現在の Ruby は，中間コードとして構文木を採用している．中間コードとして構文木を採用しているため，実行速度が遅くなると考えられ

*広島市立大学情報科学研究科

る．そこで，この欠点を解消するため，Ruby の中間コードを VM のバイトコードに置き換える (図 1)．

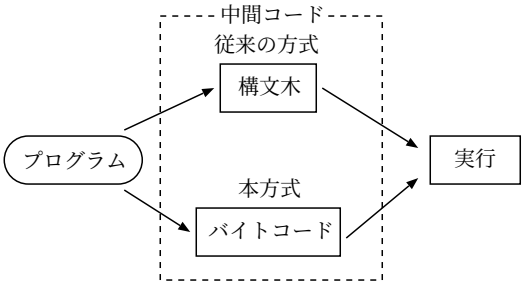

図 1: 中間コードの違いによる実行方法の変化

中間コードを構文木から VM のバイトコードに変更する利点を以下に挙げる．

- 構文木をたどって命令を解釈すると，関数の再帰的な呼び出しが行われる．しかし，バイトコードの命令は逐次に命令を実行するため，構文木よりも高速に命令を解釈することができると考えられる．
- 構文木を構成するノードよりもバイトコードのほうがメモリ使用量が少なくなる．
- リバースエンジニアリングを防ぐことが可能となる．

3 設計

3.1 VM の構造

VM の構造を図 2 に示す．VM の構造の各部分は以下のような目的で利用される．

クラス定義 (Class Definition)
: class 命令によって読み込んだクラスの定義を保持する．

ヒープ (Heap)
: newobj 命令で生成されたオブジェクトを保持する．ここにあるオブジェクトがガベージコレクションの対象となる．

スレッド (Thread)
: プログラムの実行単位である．プログラムの実行に必要な情報を保持する．スレッドは，実行状態に応じたフレームを保持する．フレームがすべてなくなると，スレッドは終了する．

フレーム (Frame)
: メソッドの実行コンテキストである．メソッド呼び出しごとに生成され，メソッドの終了後に破棄される．これは，オペランドスタックとローカル変数を保持する．

オペランドスタック (operand stack)
: 命令の引数や返り値を格納する領域である．スタックの大きさはコンパイル時に計算される．

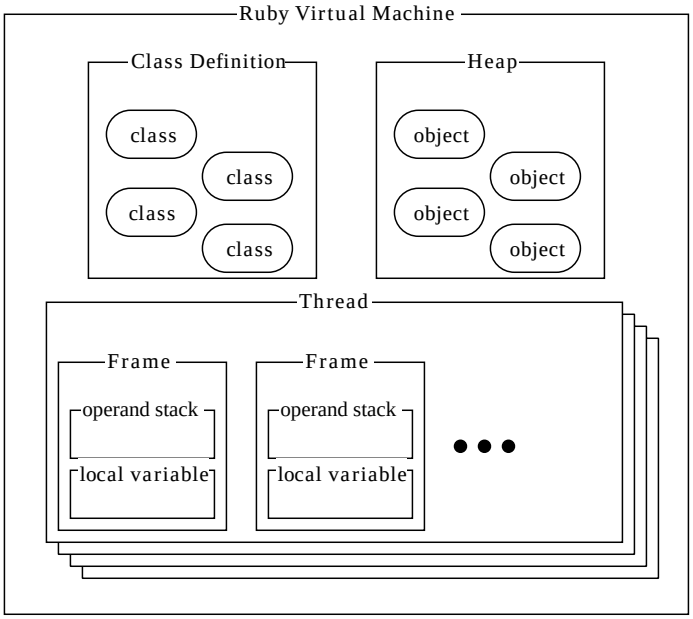

図 2: VM の内部構造

ローカル変数 (local variable)
: メソッドのローカル変数を格納するための領域である．ローカル変数の個数はコンパイル時に計算される．

3.2 命令

VM の命令の引数は，コードの中，もしくは，オペランドスタックにある．例として，push 命令と add 命令を考える．push 命令は，引数をオペランドスタックに入れる命令である．push 命令の引数は，命令の直後，すなわち，コードの中にある．add 命令は，引数を加算する命令である．add 命令の引数は，オペランドスタックにあり，オペランドスタックから 2 つの値をポップし，それらの値を加算した結果をオペランドスタックにプッシュする．

クラス名を示す文字列などの複雑な引数をとる命令がある．このような命令の引数は，コードの中に直接埋め込むことはせずに，コンスタントプールと呼ばれる定数を格納する場所に埋め込む．そして，引数はコンスタントプールの参照場所を指定するようにする．例えば，あるクラスのオブジェクトを生成する newobj 命令は引数にクラス名 (文字列) を指定する必要がある．このクラス名は，コンスタントプールに格納され，newobj の引数ではクラス名の情報が格納されているコンスタントプールの場所を指定する．

以下で，VM の命令をいくつか説明する．

class クラス名
: クラス名が参照しているクラスを定義する．

newobj “クラス名”
: “クラス名” で指定されるクラスのオブジェクトを生成する．

push 数値
スタックに引数で指定される定数をプッシュする．

push_const_n
スタックに定数 n をプッシュする．n の値は -1 〜 4 の範囲である．ただし -1 の場合は n は m1 と記述する．

load_n
n 番目のローカル変数をスタックにプッシュする．n の値は 0 〜 4 の範囲である．

store_n
スタックからオブジェクトをポップし，それを n 番目のローカル変数に格納する．n の値は 0 〜 4 の範囲である．

setpool
プログラム中で使用する定数を格納するためのコンスタントプールを，現在のコードで使用されるものに設定する．これは，コンスタントプールはコンパイル単位に一つずつ存在するため，この命令以降のコンスタントプールの参照を正しいコンスタントプールに結びつけるために必要である．

ifeq ラベル
スタックのトップの値が 0 と等しければラベルにジャンプする．

invokevirtual “メソッド名”[引数の数]
スタックのトップにあるオブジェクトのクラスのメソッドを呼び出す．

return
スタックのトップの値を戻り値にしてメソッドを終了する．

3.3 実行

簡単な例を用いて，Ruby のコードがどのようなアセンブリコードに変更されるかを示す．以下のことを行うプログラムを例として説明する．

1. 最初に，メンバ変数 @local を設定する set メソッドを持つクラス A を定義する
2. クラス A のオブジェクトを生成する．
3. 生成したオブジェクトの set メソッドを呼び出す．

図 3 が Ruby のコードと変換後のアセンブリコードである．

クラス定義 この部分は，クラスを定義するだけである．定義されたクラスの情報は，仮想マシンの Class Definition に保持される．

オブジェクトの生成 ここで Ruby のコードには `a` というローカル変数が現れているが，アセンブラではこれを仮想マシンの Frame 内の local variable の 0 番目に割り当てている．そのため `newobj` で生成されたオブジェクトは，`store_0` で 0 番目に格納されている．

メソッドの呼び出し 仮想マシンレベルでメソッドを呼び出すには `invokevirtual` 命令を実行する．このメソッドを呼び出すには，引数，オブジェクトの順番でオペランドスタックに積み，`invokevirtual` を実行する．

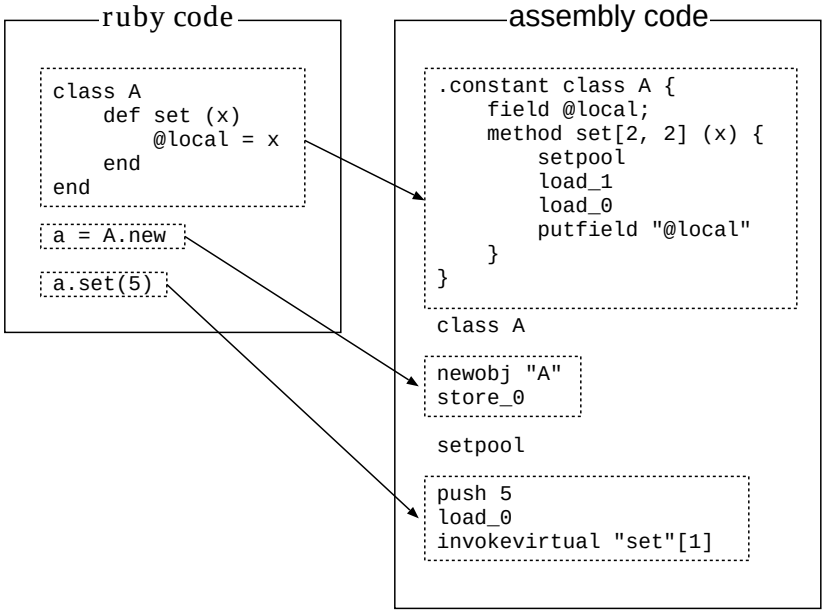

図 3: コードの例

4 性能評価

表 1: 評価環境

CPU	PentiumII 350MHz	
キャッシュ	1 次キャッシュ	データ: 16 KB 命令: 16 KB
	2 次キャッシュ	512KB
主記憶	128 MB	
OS	Linux 2.2.16	
コンパイラ	gcc 2.95.2	

インタプリタ方式の従来手法（ruby）と VM 上で実行する本手法（vm）でテストプログラムを実行し，結果を比較することで評価する．また，C で実行した場合とも比較する．評価環境を表 1 に示す．

現在，VM には数値クラスしか実装していないため，テストプログラムとして，30 番目のフィボナッチ数を求めるプログラムと竹内関数 tak(18, 12, 6) を 20 回計算するプログラムを使用する．それぞれの Ruby のコードとアセンブリコードを図 4 と図 5 に示す．

表 2: 実行時間

	fib	tak
ruby	14.36	6.27
vm	6.78	2.93
C	0.14	0.07

単位はすべて sec

テストプログラムの実行時間を表 2 に示す．表 2 から，従来の方式と比べ，VM のほうが 2.1 倍速いことが分かる．また，C で実行するほうが，VM で実行するよりも 42〜48 倍速いことが分かる．

5 結論および今後の展望

本論文では，オブジェクト指向スクリプト言語 Ruby のための VM の設計について述べた．また，性能評価を行い，VM 上で実行することで Ruby のプログラムが高速に実行されることを確認した．

現在，VM の一部と数値クラスの実装が完了している．今後は，以下の実装を行う予定である．

- 数値クラス以外のクラスの実装.
- ガベージコレクションの実装.
- スレッドへの対応.
- バイトコードコンパイラの実装.

現在の VM は不完全なうえ，Just In Time コンパイラなどを用いて最適化されていない．そのため，様々な高速化手法を用いれば，C での実行に近づくと考えている．

また，VM 上でプログラムを実行するようになると，拡張ライブラリをどのように構築するかという問題がある．互換性の問題もあるため，これについては現在未定である．C だけでなく，他のプログラミング言語で作られたプログラムを Ruby が容易に用いることができるようにする機構が必要であると考えている．

参考文献

[1] Ousterhout, J.: Scripting: Higher level programming for the 21st century, *IEEE Computer*, Vol. 31, No. 3, pp. 23–30 (1998).

[2] Prechelt, L.: An Empirical Comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a Search/String-Processing Program, Technical Report 2000-5, Fakultät für Informatik, Universität Karlsruhe, Germany (2000).

[3] まつもとゆきひろ, 石塚圭樹: オブジェクト指向スクリプト言語 Ruby, アスキー出版局 (1999).

A テストプログラムのコード

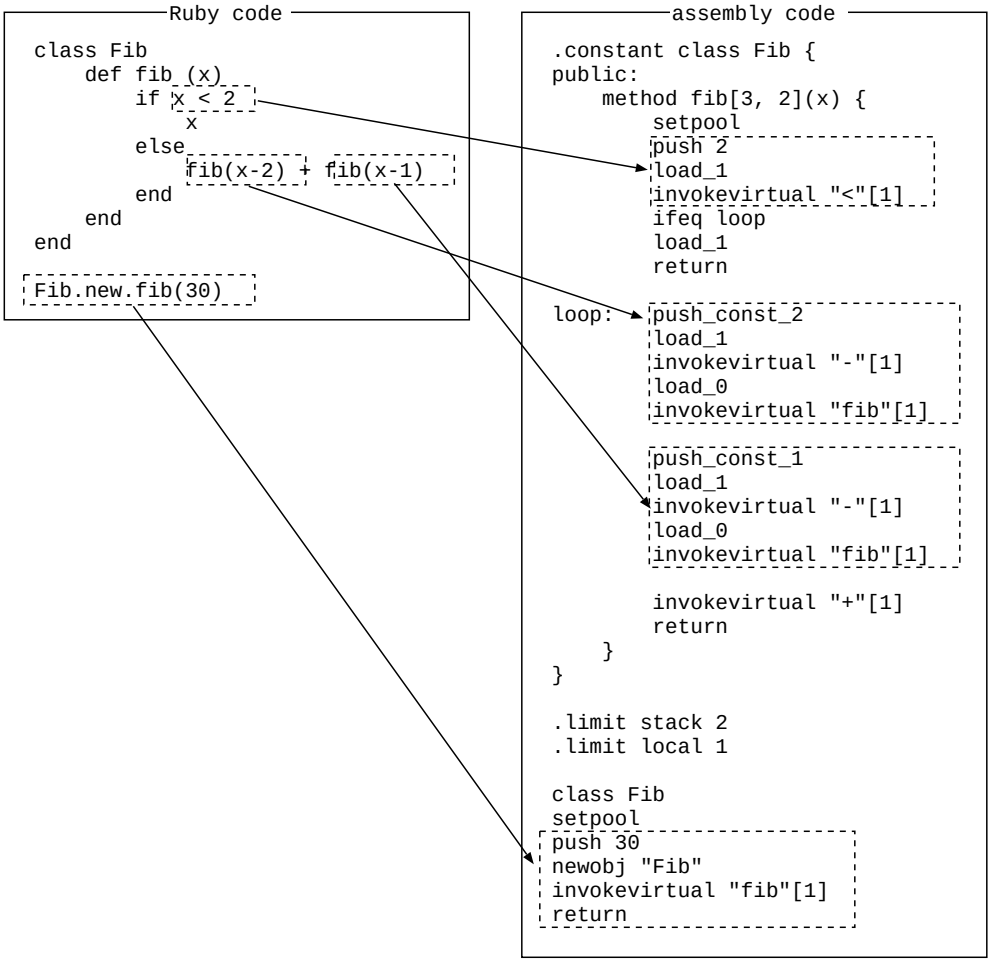

図 4: フィボナッチ数のプログラム

Ruby code

```
class Tak
  def tak (x, y, z)
    if y >= x
      z
    else
      tak(tak(x - 1, y, z),
          tak(y - 1, z, x),
          tak(z - 1, x, y))
    end
  end
end

a = Tak.new
a.tak(18, 12, 6)
    .
    .    20 times
    .
a.tak(18, 12, 6)
```

assmebly code

```
.constant class Tak {
public:
    method tak[6, 4](x, y, z) {
        setpool
        load_1
        load_2
        invokevirtual ">="[1]
        ifeq loop
        load_3
        return

loop:   load_2
        load_1
        push_const_1
        load_3
        invokevirtual "-"[1]
        load_0
        invokevirtual "tak"[3]

        load_1
        load_3
        push_const_1
        load_2
        invokevirtual "-"[1]
        load_0
        invokevirtual "tak"[3]

        load_3
        load_2
        push_const_1
        load_1
        invokevirtual "-"[1]
        load_0
        invokevirtual "tak"[3]

        load_0
        invokevirtual "tak"[3]
        return
    }
}

.limit stack 4
.limit local 3

class Tak
newobj "Tak"
store_0

setpool
jsr task
    .
    .    20 times
    .
jsr task
load_2
return

task:   store_1
        push 6
        push 12
        push 18
        load_0
        invokevirtual "tak"[3]
        store_2
        ret 1
```

図 5: 竹内関数のプログラム