Ruby 処理系での軽量な浮動小数点数表現

笹田 耕一 †

オブジェクト指向スクリプト言語 Ruby は変数に型の無い動的なプログラミング言語であり，コンポーネントを組み合わせる糊付け言語のとしての特徴を有する．Ruby 処理系ではこの動的な型を実現するために，オブジェクトごとに型情報を含めた領域を割り当て，オブジェクトをその領域へのポインタ値で表現する．この方式では各オブジェクトごとに領域を割り当てることになりメモリ効率が悪い．とくに，現在の Ruby 処理系では浮動小数点数 1 つに 1 つの領域を割り当てるため効率が悪い．そこで，本稿では 64 ビット CPU を用いて浮動小数点数をポインタ型に埋め込む方式を提案する．本稿では現状の Ruby 処理系でのデータ構造，とくに浮動小数点数のデータ表現について説明する．そして，提案する浮動小数点数の表現方式について述べ，実装し評価した結果を述べる．

A Lightweight Representation of Floting-Point Numbers on Ruby Interpreter

Koichi Sasada †

Ruby is an object-oriented, dynamic, programming language. For example, all variables are not typed. This feature is useful for combining components as a glue language. The Ruby interpreter represents every object as a pointer to an entity in heap memory. An object entity includes the object type and other information. However, this representation can be inefficient because every object needs an entity in heap memory. Especially, the representation of floating-point (FP) is inefficient because numbers as a double precision value needs additional object header. To solve this issue, we propose an efficient representation of Ruby's FP numbers on 64bit CPUs. The idea is simple: represent the FP object as an encoded, immediate pointer value similar to Ruby's Fixnum representation. In this paper, we describe an object representation for the Ruby interpreter, showing the proposed method and the evaluation results.

1. はじめに

オブジェクト指向スクリプト言語 Ruby[15)14)]（以降 Ruby）は，その使いやすさから世界中で利用されているプログラミング言語である[16)]．Ruby の特徴の 1 つに動的な言語仕様がある．たとえば，変数に対して格納するオブジェクトのクラスを記述する必要がなく，どのようなクラスのオブジェクトも任意の変数に格納することができる．この機能により，Ruby を様々なコンポーネントを組み合わせて目的のソフトウェアを構築するために必要な糊付け言語として利用することができる．

Ruby では，基本的に各オブジェクトのアイデンティティをヒープ上に配置した領域へのポインタとして表現している．ポインタが指し示す領域に，それぞれオブジェクトごとにそのオブジェクトがどのような情報を保持しているのかを示す情報や，そのオブジェクトのクラスを示す情報などを記録するヘッダ部を格納している．このオブジェクトの表現手法は動的な型をもつプログラミング言語を実装する一般的な手法である．

Ruby では浮動小数点数の計算もサポートしている．Ruby で計算時間のかかる数値計算を行うことは，現在の処理系の速度的な制約からあまり行われることは無いが，C 言語などで記述された既存の数値計算ライブラリへのインターフェースとして利用したり，大規模な数値計算のプロトタイプとして記述力の高い Ruby を利用するなど，Ruby での浮動小数点数を利用した数値計算の需要がある．とくに，Ruby の記述力の高さから，より高速に実行できるのであれば Ruby で数値計算を行いたい，という声を聞く．

Ruby での浮動小数点数は，C 言語の double 型で表現される倍精度浮動小数点数であり，`Float` オブジェクトとして利用できる．現在の Ruby 処理系では，他のオブジェクトと同様に `Float` オブジェクトそれぞれにヒープ上の領域を確保することで実現している．これは，`Float` オブジェクトごとにヘッダ部を保持するためメモリ効率が悪い．とくに，近年広く利用されている 64 ビット CPU 上では，実際に表現したい double 型の値に比べ，ヘッダ部が大きく効率が悪い．

† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

また，浮動小数点数を利用した計算結果ごとに Float オブジェクトを生成するため，ガーベージコレクションを頻繁に起動することになり実行時間に大きな影響を与えており，Ruby で数値計算が大きな要因の 1 つとなっている．

そこで，本稿では 64 ビット CPU 上において，Float オブジェクトの表現をヒープ上の実体の割り当てをほぼ不要とする方式について提案する．基本的なアイデアは Ruby の固定範囲の整数値である Fixnum オブジェクトを表現する手法と同様に，ポインタを格納する領域にポインタではない，という印（タグ）を付けることで実現する（unbox 化する）．unbox 化にはタグを付加するための領域が必要となるが，現在の計算機環境での double 型の浮動小数点数が IEEE 754[1)5)] 規格で規定されている倍精度浮動小数点数でメモリ上に格納されることを利用して，64 ビットの精度を落とさずにタグを付加し，効率的な Float オブジェクトの表現手法を実現する．

本稿で提案する手法により，Ruby における浮動小数点数を利用した数値計算において，メモリ確保やガーベージコレクションのオーバヘッドを排除し，以前よりも高速な数値計算を可能とした．

なお，本稿で対象とする Ruby 処理系とは，C 言語で記述している仮想マシン YARV[17)] を導入した Ruby 1.9.0 を示す．

本稿では，2 章で Ruby でのオブジェクトの一般的な表現手法を説明し，Float オブジェクトの表現手法を述べ，現在の Ruby 処理系での表現手法の問題点を述べる．3 章で，64 ビット CPU 上での効率的な浮動小数点数表現の手法について提案する．4 章で提案手法を実装し，評価を行った結果を示し，5 章で関連研究を述べ，6 章でまとめる．

2. Ruby オブジェクトの表現

本章では現在の Ruby 処理系でのオブジェクトの表現手法と浮動小数点数の表現手法を述べる．そして，現在の浮動小数点数の表現が非効率であり，数値計算におけるボトルネックの一因であることを示す．

なお，本章で述べる内容は文献 18) に詳しいので詳細はそちらを参照されたい．

2.1 ポインタ参照による表現

C プログラム中で Ruby のオブジェクトを格納する型として VALUE 型が用意されている．VALUE 型はその実行環境でのポインタ型と同じサイズの型となっており，一般的に 32 ビット CPU では 4 バイト，64 ビット CPU では 8 バイトの大きさを持つ．なお，本稿ではポインタ型のサイズを 1 ワードと数える．

後述する VALUE 埋め込み表現を用いない，ほとんどの Ruby オブジェクトでは VALUE 型がそのオブジェクトの情報が格納されているヒープ上の領域へのポインタとなっている．その領域は RVALUE[☆] データ構造として規定されている．

```
struct RBasic {
 VALUE flags;
 VALUE klass;
};
```

図 1 ヘッダを表現する RBasic 構造体

```
struct RString {
 struct RBasic basic;
 long len;
 char *ptr;
 VALUE aux;
};
```

図 2 文字列表現

RVALUE データ構造は，2 ワードのヘッダ部と 3 ワードの本体部の合計 5 ワードのデータである．ヘッダ部を表現するのは図 1 で示す RBasic 構造体である．RBasic 構造体の flags メンバにこのデータ表現の型（データ型）や GC に必要となるデータ領域（1 bit），型ごとに必要となる情報が格納されており，klass メンバにはこのオブジェクトが属するクラスオブジェクトへの参照を保持している[☆☆]．

Ruby のメモリ管理はこの RVALUE の獲得，解放によって行われる．Ruby 処理系はヒープ上に RVALUE の集合を用意しておき，オブジェクト生成時にはこの集合から RVALUE 要素を 1 つ切り出す．取り出すことができる RVALUE が無くなったとき，もしくは malloc() によって獲得したメモリ使用量が閾値を超えたときにガーベージコレクション（GC）を行い，参照のない RVALUE を回収する．回収しても必要な RVALUE が確保できなかった場合，RVALUE の集合を新たに malloc() によって加える．

本体部の利用方法やデータ構造のレイアウトはデータ型ごとに異なっている．例として文字列データ型と配列データ型を示す．

文字列データ型（図 2）では，本体部の len メンバに文字列の長さ，ptr メンバに文字列の実体が格納されている．aux メンバは文字列の共有などに使う．配列データ型（図 3）では len メンバに配列の長さ，ptr に VALUE 型を格納するための配列の実体，aux に配列のキャパシティなどを記録してある．文字列，配列ともに，計算の対象とする領域は ptr メンバが指し示

☆ RVALUE データ構造は Ruby のソースコード gc.c 中で共用体として宣言されている．

☆☆ メンバ名が class でない理由は，テキストエディタなどで C++ のキーワード class を特別扱いするからである

```
struct RArray {
 struct RBasic basic;
 long len;
 VALUE *ptr;
 VALUE aux;
};
```

図 3 配列表現

すメモリ領域にある．このメモリ領域は malloc() によって獲得され，GC によってそのオブジェクトが回収・解放されるときに free() によって解放される．

なお，Ruby 1.9 からは長さが短い文字列や配列は RVALUE の領域中にデータの実体を記録するようにしてメモリ使用量の最適化が施されている．これにより，小さな領域のために malloc() と free() を繰り返すことがなくなった．

文字列と配列以外のデータ型については，ユーザ定義オブジェクト，クラス，モジュール，正規表現，ハッシュ，Ruby レベル構造体，Bignum，ファイル，拡張用データ，そして浮動小数点数を示す Float データ型がある．このデータ型の表現方法については後述する．

2.2 VALUE 埋め込み表現

Ruby プログラミングで利用するデータはすべてオブジェクトなので，例えば 1 のような整数もオブジェクトとして扱われる．しかし，このような整数を RVALUE データ構造として割り当て，解放を管理すると非効率なので，VALUE 型はヒープ上のオブジェクト実体へのポインタ以外も格納可能なような設計になっている．

RVALUE データへのポインタは 4 バイトアラインメントとなるように調整されているので，VALUE 型のデータがポインタである場合は，その値の下位 2 ビットは 0 である．この特徴を利用して，下位 2 ビットが 0 で無い場合を特殊な表現手法であると認識する．本稿では下位 2 ビットの領域をタグと呼び，VALUE 型の変数にポインタ以外を格納する表現を VALUE 埋め込み表現と呼び，この方式で表現されたオブジェクトを VALUE 埋め込みオブジェクトと呼ぶ．VALUE 埋め込みオブジェクトのデコード方式はそれぞれ異なり，それぞれはタグによって識別される．タグによるデータ表現は動的型をもつプログラミング言語で一般的に利用される手法である．

VALUE 埋め込みオブジェクトとしては Fixnum オブジェクト，Symbol オブジェクト，true，false，nil オブジェクトがある．

Fixnum オブジェクトは，範囲固定の整数型である．VALUE 型の下位 1 ビットが 1 のとき，Fixnum オブジェクトとして認識される．デコード方式は右へ 1 ビット算術シフトすることで行われる．32 ビット CPU ではタグとして 1 ビット消費するため，31 ビットで表現できる範囲の整数を Fixnum オブジェクトとして表現可能である．64 ビット CPU では同様に 63 ビットで表現できる整数を表現する☆．なお，Ruby では計算中に Fixnum の範囲を超えた整数は自動的に Bignum オブジェクトに拡張されるようになっている．Bignum は RVALUE によって表現されており，ヒープ上に確保されるため，メモリが許す限り大きな数が表現可能である．

```
struct RFloat {
 struct RBasic basic;
 double float_value;
};
```

図 4 Float オブジェクトの RVALUE 表現

VALUE 型の値の下位 8 ビットが 16 進数で 1e のときは Symbol オブジェクトとして扱われる．Symbol オブジェクトを示す VALUE 型のデータには，タグに利用する下位 8 ビット以外の領域，つまり 32 ビット CPU で 24 ビット，64 ビット CPU で 56 ビットの領域にシンボル表のキーが格納されている．

なお，本論文では C 言語の表現に則り 0x1e のように 16 進数を表記する．

true，false，nil オブジェクトはそれぞれ 1 つの値しか存在しないので，VALUE 型の値がそれぞれ 0x02，0x00，0x04 のときにそれらのオブジェクトとして認識される☆☆．

VALUE 埋め込み表現では RVALUE の割り当て・解放が不要なので，実行時間的にもメモリ消費量的にも効率的である．

2.3 浮動小数点数の表現

Ruby では浮動小数点数を表現する Float オブジェクトを RVALUE データ構造によって表現する．Float オブジェクトの RVALUE は図 4 に示す RVALUE データによって表現されている．float_value メンバが倍精度浮動小数点数（double 型）として Float オブジェクトが表現する数値自体を記録する．つまり，Float オブジェクトはメモリ管理の対象となるオブジェクトである．

この Float オブジェクトの表現手法には次のような問題点がある．

まず，実行効率の問題がある．数値計算プログラムにおいて Float オブジェクトは大量に，とくに一時的なオブジェクトとして生成される．この結果，RVALUE の生成および初期化やガーベージコレクタの起動が頻

☆ ただし，Microsoft Windows のような，64 ビット CPU で long 型が 32 ビットである OS 上では，互換性のために Fixnum オブジェクトは 31 ビットで表現できる範囲の整数となる．

☆☆ この表現は，ヒープ領域の最小アドレスが 0x04 よりも大きいことを仮定している．

```
i = 0; f = 0.0
while i<30_000_000
  i += 1
  f += 0.1; f -= 0.1
  f += 0.1; f -= 0.1
  f += 0.1; f -= 0.1
  f += 0.1; f -= 0.1
end
```

図 5 浮動小数点数を計算するトイプログラム

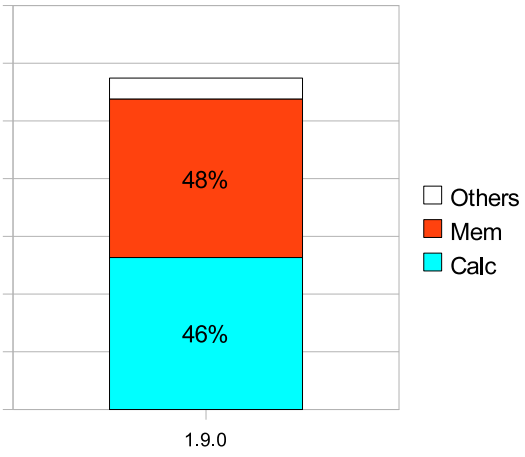

図 6 トイプログラムの実行時間の内訳

発することになり実行時間に大きな影響を与える．

第二に非効率なメモリ消費が挙げられる．RVALUE データ表現として 3 ワード割り当てられるが，double 型は 32 ビット CPU の場合には 2 ワード，64 ビット CPU では 1 ワードで表現できるため，未使用領域が発生する．RVALUE へのポインタである `VALUE` 型を合わせると，1 つの `Float` オブジェクトを表現するためには 6 ワード消費することになる．C 言語では 2 ワードないし 1 ワードで表現できる double 型の値を Ruby で利用するためには必ず 6 ワード消費することになり，メモリ消費量的に問題である．

問題点のうち実行効率の問題を確認するために，図 5 で示すような簡単なトイプログラムを作成し，4 章で述べる環境で Ruby 1.9.0 を実行した結果から，その実行時間に占める各処理の割合を図 6 示す．なお，この内訳は OProfile[6] を利用して算出した．Calc は VM の実行や数値計算にかかる時間，Mem はメモリ管理，つまりオブジェクトの生成やガーベージコレクションにかかる時間である．このプログラムでは単純な数値計算の繰り返ししか行わないが，メモリ管理のために費やす時間が全実行時間の 46%と，大きいことがわかる．このような単純な数値計算においてメモリ管理のオーバヘッドが大きく占めているのは問題である．

アプリケーションによっては 64 ビット倍精度浮動小数点数をあきらめ，精度を落として一部をタグとして用いることで VALUE 埋め込み表現とすることができるが，計算精度を必要とするアプリケーションの記述が不可能となり問題である．

3. 64 ビット CPU 上での浮動小数点数の VALUE 埋め込み表現の実現

現在の Ruby 処理系において，`Float` オブジェクトを利用する数値計算は非効率であることを示した．そこで，本章では `Float` オブジェクトを VALUE 埋め込みオブジェクトとすることでメモリ管理のオーバヘッドを削減する手法を提案する．

提案する手法では，64 ビット CPU において IEEE 754[1] 倍精度浮動小数点数の精度を落とすことなく，簡単なビット演算を利用してほとんどの場合にポインタ埋め込み表現とすることが可能であり，数値計算におけるメモリ管理のオーバヘッドを大きく削減する．

ポインタ埋め込み表現とするための基本的なアイデアは，現在の Ruby 処理系が実現している `Fixnum` オブジェクトの VALUE 埋め込み表現のように，下位 1 ビットを 1 としてポインタ型と区別できるようにするという単純なものである．ここで，64 ビット倍精度浮動小数点数の精度を落とさずに，そして VALUE 埋め込み表現への高速な符号化および double 型への復号を実現する工夫が本研究での提案の骨子である．

3.1 倍精度浮動小数点数の VALUE 埋め込み表現

64 ビット倍精度浮動小数点数を精度を落とさずにポインタ埋め込み表現とするためには 63 ビットで表現可能な領域においてのみポインタ埋め込み表現とし，それ以外の範囲では従来通りの RVALUE を用いた表現とすればよい．

そこで，63 ビットで表現可能な倍精度浮動小数点数の範囲を検討し，その範囲に限って実際に埋め込み表現にエンコード，デコードするための処理を述べる．

3.1.1 63 ビットで表現可能な倍精度浮動小数点数の範囲の検討

まず，IEEE754 で規定されている倍精度浮動小数点数のフォーマットについて確認する．64 ビットの最下位ビットから最上位ビットまでの各ビットを $b_i (0 \leq i \leq 63)$（ただし，b_0 が最下位ビット）と置くと，b_{63} が符号ビット s，$b_{62} \sim b_{52}$ の 11 ビットがバイアスのかかった指数 e，$b_0 \sim b_{51}$ の 52 ビットが仮数 m を構成し，$1 \leq e \leq 2046$ の範囲での正規化数の値 f は $(-1)^s 2^{e-1023} m$ （ただし，$m = 1 + \sum_{i=0}^{51} b_i / 2^{52-i}$）となる．

ここで，倍精度浮動小数点数が表現可能な数値の範囲を考える．一般的に，絶対値の小さい数，および絶対値の大きな数は，その間の数よりもプログラム中で

の出現頻度は少ないことが期待できる．そこで，指数が $512 \leq e \leq 1535$ の範囲である数値 f の場合には VALUE 埋め込み表現とすることを考える．この範囲外の数値は，0 以外，実験に利用した Ruby プログラムであまり出現しないことを確認してある．

さて，$512 < e < 1535$ の範囲，つまり $0x200 \leq e \leq 0x5FF$ の範囲では，e の上位 2 ビット，数値 f のビット表現での b_{62} と b_{61} が $b_{62} \neq b_{61}$ という関係となっている．つまり，b_{61} は b_{62} を反転したもので表現可能である．この性質を利用すると，この範囲では 63 ビットで倍精度浮動小数点数が表現できることになる．そこで，e がこの範囲にある数値 f を不要となった 1 ビットをタグとすることで VALUE 埋め込みオブジェクトとして表現し，e がこの範囲でないものは，今まで通り RVALUE データ構造を用いるようにする．

この工夫により，絶対値が極端に大きいか小さい数値以外は VALUE 埋め込み表現で `Float` オブジェクトを表現することができる．また，その範囲以外の数値も，従来の手法を用いることで表現可能である．つまり，`Float` オブジェクトが表現可能な数値の精度を保っている．

3.1.2 ビット演算による実現手法

VALUE 埋め込み表現を利用する範囲 $512 < e < 1535$ であるかどうかをチェックするためには $b_{62} \neq b_{61}$ であればよいことがわかった．つまり，これはビット演算によって確認できる．VALUE 埋め込み表現を実現するためのタグの挿入は，この不要となった 1 ビットを利用して行う．基本的な考え方は以上だが，実際に VALUE 埋め込み表現を効率的に実現するために，ビット演算手法を工夫したのでここで紹介する．

なお，`Float` オブジェクトを VALUE 埋め込みオブジェクトとするために，下位 1 ビットをタグとして利用する．このため，従来は下位 1 ビットをタグとして表現していた `Fixnum` オブジェクトなどの表現を変更する必要があったが，これについては次節で述べる．

まずは double 型の値を VALUE 型へ符号化するルーチン `rb_float_new()` を図 7 に示す．

まず，`BIT_DOUBLE2VALUE()` 関数によって double 型のビット表現を `VALUE` 型に取り出し `v1` とする．そして，`v1` を `BIT_ROTL()` によって 3 ビット左へローテートして `v2` とする．この操作により，b_{62} と b_{61} のビットが `v2` の下位 2 ビットへ位置する．そして，b_{62} と b_{61} がそれぞれ異なるビットであるかを調べるために `(v2+1) & 0x02` という評価を行い，下位 2 ビットがそれぞれ違うか，つまり e が VALUE 埋め込み表現可能な範囲内であるかどうかを確認している☆．もし埋め込み表現が可能な範囲であれば，タグとして下位 1 ビットを 1 にしてその値を返す．下位 1 ビットは，b_{61} がローテートされた箇所なので，この情報を上書きしても，b_{62} によって復元可能である．e が埋め込み可能な範囲でなければ，従来通りメモリ管理の対象となるオブジェクトをヒープ上で獲得，初期化して返す．

```
VALUE rb_float_new(double dbl) {
  VALUE v1 = BIT_DOUBLE2VALUE(dbl);
  VALUE v2 = BIT_ROTL(v1, 3);
  if ((v2 + 1) & 0x02) {
    return v2 | 0x01;
  }
  else {
    if (dbl == 0) {
      return ruby_float_zero;
    }
    else {
      return rb_float_new_in_heap(dbl);
    }
  }
}
```

図 7 double 型から VALUE 型への変換ルーチン

ビットのローテートを利用することで，最下位ビットの演算に帰着したことが高速な符号化，および後に述べる復号のための工夫となっている．ローテートは，CPU に対応する命令があれば 1 命令で実現可能である．

ここで，数値 +0 が埋め込み表現可能な範囲にないことに注意が必要である（数値 +0 では $e = 0$ となる）．+0 はプログラム中に頻出するため，毎回 RVALUE データ構造の領域を確保するのは無駄である．そこで，+0 用の領域を事前に確保しておき，+0 だった場合にはそれを返すようにした．この工夫により +0 が出現するたびにメモリ確保が発生するようなことが無いようにした．−0 も同様にサポート可能である．

`rb_float_new()` ルーチンによって符号化した `Float` オブジェクト表現から，double 型の値を取り出す `RFLOAT_VALUE()` ルーチンを図 8 に示す．まず，下位 1 ビット目が 1 であるかを調査し，そうであれば VALUE 埋め込み表現の `Float` オブジェクトとしてデコードする．デコードは，下位 2 ビット目と下位 1 ビット目の xor を下位 1 ビット目とすることで，それぞれが違う符号とするように調整し，右へ 3 ビットローテートすることで元の double 型の値を示すビッ

☆ ここは少しわかりづらいが，評価環境では図に示したプログラムがもっとも性能が良かったためこのプログラムを記載した．簡単にするためには `v2 & 0x03` が 1，もしくは 2 であることを確認すれば良い．

```
double RFLOAT_VALUE(VALUE v) {
  if (v & 1) {
    VALUE v1 = v ^ ((v >> 1) & 1);
    VALUE v2 = BIT_ROTR(v1, 3);
    return BIT_VALUE2DOUBLE(v2);
  }
  else {
    return RFLOAT(v)->float_value;
   }
}
```

図 8 VALUE 型から double 型への変換ルーチン

ト列へ戻す．そして，BIT_VALUE2DOUBLE() によってビット列を double 型へ変換して返す．下位ビットが 1 でなければ，通常の RVALUE データ構造として扱い，ポインタを辿って値を返す．

3.2 言語仕様への影響

Float オブジェクトの VALUE 埋め込み表現の実現では，従来の Ruby 処理系との互換性を極力保つように設計したが，2 点変更が必要な点があったのでここに紹介する．なお，この 2 点は Ruby 利用者にはほぼ影響がないものである．

まずは Fixnum オブジェクトの表現を変更し，従来は 63 ビットで表現できる範囲を Fixnum オブジェクトとして表現できたが，これを 61 ビットの範囲に狭めることにした．具体的には，下位 3 ビットをタグとし，下位 3 ビットが 0x02 であった場合，Fixnum である，というように Fixnum オブジェクトの表現を変更した．しかし，整数の範囲が 61 ビットで足りないことはあまりないため，問題はない．なお，この変更に伴い，true の表現も変更が必要となった．

次に，各オブジェクトにユニークに付与されるオブジェクト ID についての変更である．従来の Float オブジェクトはそれぞれがヒープ上で確保されるオブジェクトであるため，同一の値であってもオブジェクト ID が異なる可能性があった．しかし，VALUE 埋め込み表現可能な Float オブジェクトは同じ値であれば必ず同じオブジェクト ID が返ることになった．しかし，数値計算において Float オブジェクトのオブジェクト ID に依存したプログラミングを行うことは考えづらいので，この変更も影響はない．

4. 評　　価

本章では提案した手法を実際に Ruby 1.9.0 に実装し，評価した結果を示す．

4.1 評 価 環 境

評価環境は，Intel Xeon CPU E5335 2.00GHz のプロセッサ上で行った．利用したソフトウェアは次の通りである．OS は GNU/Linux 2.6.18 x86_64，コンパイラは gcc version 4.1.2 20061115 を利用した．比較対象とする Ruby 処理系は ruby 1.9.0 (2008-01-04 revision 0) [x86_64-linux]☆，ruby 1.8.6 (2008-01-04 patchlevel 5000) [x86_64-linux]，Java で実装した Ruby 処理系である JRuby 1.1b1[8] (java version "1.6.0_03" 上で実行) を用いた．Ruby 1.9.0 の VM の最適化オプションは文献[17] で述べた最適化オプションで融合操作，スタックキャッシング最適化以外を適用した．評価は各プログラムを 3 回実行し，もっとも短い実行時間を評価値として採用した．

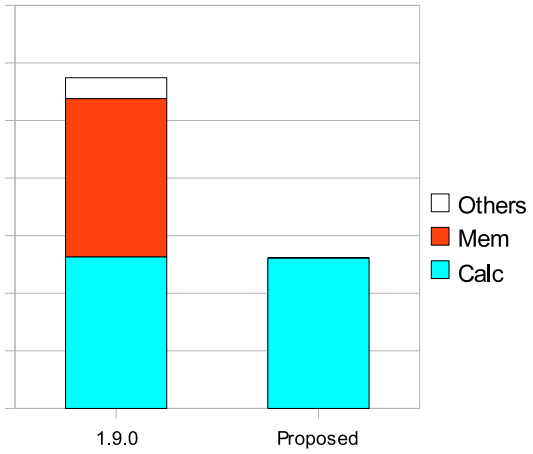

図 9 トイプログラムの実行時間の内訳の比較

なお，本章では ruby 1.9.0 に提案手法を実装した Ruby 処理系を 1.9.f と呼称する．

4.2 メモリ管理オーバヘッド削減の確認

図 5 で示したトイプログラムについて，提案手法を実装した 1.9.f で実行したときの処理時間の割合を 1.9.0 と 1.9.f で比較した結果を図 9 に示す．処理時間の割合は同様に OProfile[6] を利用して計測した．なお，縦軸は実行時間である．比較を見ると，1.9.f では明らかにメモリ管理のオーバヘッドが削減できており，その削減分が全体の実行時間の改善にそのまま貢献していることが確認できた．

4.3 ベンチマークプログラムでの評価

浮動小数点数を利用するベンチマークプログラムを，いくつかの Ruby のバージョンで実行して評価を行った．ベンチマークプログラムは The Computer Language Benchmarks Game[4] にある，マンデルブロ集合を求める mandel，n 体問題を解く nbody，素朴な数値計算の連続である partial-sums を利用した．これらを評価した結果を図 10 に示す．また，この結果を ruby 1.9.0 からの速度向上率としてまとめたものを図 11 に示す．

まず，Ruby 1.8.6 もしくは JRuby と比較して VM

☆ 提案手法を実装したベースもこのバージョンである．

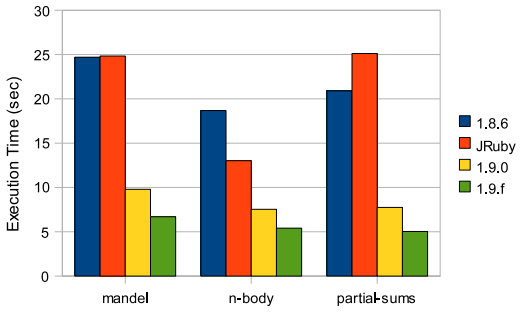

図 10 ベンチマークプログラムの実行結果（実行時間）

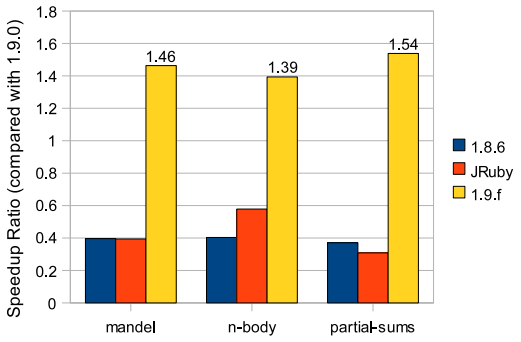

図 11 実行結果（1.9.0 との速度比較）

を導入した Ruby 1.9.0 は 2 倍以上高速であることを確認した．そして，その Ruby 1.9.0 に提案手法を適用することにより，1.9.0 と比較して各ベンチマークで約 1.5 倍の速度向上を得ることができた．Ruby 1.8.6 と Ruby 1.9.0 はメモリ管理はほぼ同等の仕組みである．また，JRuby も Box 化した浮動小数点数を利用している点で高速化が難しい．Ruby 1.9.0 での VM 化による速度向上により，Ruby 1.9.0 ではメモリ管理のオーバヘッドがより支配的となった．メモリ管理の負荷を削減する本手法は Ruby 1.9.0 において，従来の Ruby 処理系よりも有効な高速化手段といえる．

なお，本手法を適用するにあたり，特化命令（演算子などで表現される特別なメソッド用の専用命令）[17] を VALUE 埋め込み表現の `Float` オブジェクトに対応させ，`Float` オブジェクト同士の計算を高速化した．このとき，命令開始時に整数値のチェックを先に行うため整数演算同士の計算速度は変わらないが，文字列などの非数値オブジェクト同士の場合，`Float` オブジェクトのチェックが純粋な負荷となるため速度低下が発生した．この点は今後の課題である．

4.4 他言語との比較

他の言語処理系との実行時間を比較するために，図 5 で示したトイプログラム相当のプログラムを他のプログラミング言語で実装し，それらの言語処理系で実行した結果を図 12 示す．比較対象とする他の言語処理系は，Perl v5.8.8[9]，Python 2.4.4[10]，java version

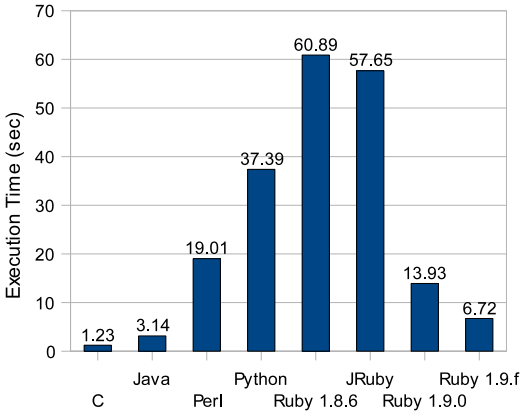

図 12 他言語との比較

"1.6.0_03"[11]，そして Ruby をビルドするときに利用した gcc (C) である．なお，Java と C については，図 5 の変数 `f` に相当する変数を，最適化によるプログラムの自体の削除を避けるために `volatile` として宣言しており，変数へのアクセスは毎回メモリを介して行うようにしている．そのため，最適化を制限する特殊な C や Java プログラムとなっていることに注意されたい．

結果を見ると，Ruby 1.9.0 および Ruby 1.9.f は Perl や Python といったスクリプト言語よりも高速に実行できていることがわかった．しかし，C や Java といったオブジェクトとして数値を扱う必要がない型付きの静的プログラミング言語のレベルにはまだ達していないことを確認した．

5. 関 連 研 究

Ruby の浮動小数点数の利用にはメモリ管理の負荷が問題となることはよく知られている．これを回避するために NArray[12] という，数値しか格納することができない配列型を実現する拡張ライブラリがある．数値しか格納しないため，メモリ管理の対象ではないため負荷はない．NArray は，数値計算が配列上での計算だけで済む場合は効率が良いが，数値を配列から取り出して操作を行うときにメモリ管理の負荷が発生する．

プログラム中で浮動小数点数を利用する数値計算を行う箇所が限定されている場合は，その箇所を C 言語で書き直し，拡張ライブラリとして実装することが可能である．また，拡張ライブラリではなく，RubyInline[2] によって Ruby プログラム中に C 言語で問題の箇所を記述することも可能である．つまり，Ruby 以外の言語を利用することで Ruby 処理系の問題を回避するという手法である．しかし，これらの解決方法は C 言語の知識を必要とするため難易度が高く，また Ruby だけで記述していれば起こり得ないメモリ保護

違反などの危険を招くことになる．

言語処理系での浮動小数点数を表現するオブジェクトの軽量な表現手法としては多くの研究があるが，例えばSML#の研究[7]が挙げられる．これは，オブジェクトが置かれるスタックやヒープに対して，コンパイル時にメモリ上のどこにどの型が配置されるかを示すビットマップを用意することでオブジェクトごとに型情報を付加することを防ぐ．このような手法はコンパイル時に型が解析できるシステムにおいて有用であるが，Ruby ではこのような解析は困難である．

浮動小数点数の指数部の一部をタグビットに利用するというアイデアはすでに提案されている[3]が，タグビットに 3 ビットを利用する点，タグの埋め込みや埋め込み可能かの判断をナイーブな計算で行っており効率が悪い．

オブジェクトの表現からヘッダ情報を取り除く一般的な手法として，64 ビット CPU の広大なアドレス空間を利用してオブジェクトの配置するアドレスと型を関連づける TVA (Typed Virtual Addressing)[13] やこの手法を適用する型を選別する STVA があるが，ポインタ参照自体を取り除くものではない．また，メモリ管理の負荷は依然として発生する．

6. ま と め

本論文では 64 ビット CPU 上で倍精度浮動小数点数を表現する `Float` オブジェクトについて，精度を落とすことなく軽量な表現形式へ変換する手法を提案した．評価の結果，従来の `Float` オブジェクトの表現では数値計算において大きなオーバヘッドであったメモリ管理が不要となることを確認し，浮動小数点数を利用するマイクロベンチマークプログラムにおいて最大 1.54 倍の実行性能の向上を確認した．

本提案手法はビット演算のみを利用しているため，IEEE 754 の浮動小数点数を扱う言語処理系であれば Ruby に限らず適用可能である．また，本手法は 32 ビット CPU 上での単精度浮動小数点数でも利用できる．32 ビット CPU を利用する計算機環境は依然として数多く存在するため，倍精度を必要としない計算に利用することができる．

謝　　辞

本論文で提案したアイデアは，Ruby コミュニティの方々，とくに三浦英樹氏に頂いた意見を参考にしています．協力してくださった方々にこの場を借りて感謝いたします．

本研究の一部は，日本学術振興会科学研究費補助金若手研究（スタートアップ），課題番号 19800007 の助成を得て行いました．

参 考 文 献

1) *IEEE standard for binary floating-point arithmetic*. Institute of Electrical and Electronics Engineers, New York, 1985. Note: Standard 754–1985.
2) Ryan Davis and Zen Spider Software. Rubyinline. http://www.zenspider.com/ZSS/Products/RubyInline/.
3) Jason Evans. Tagged unboxed floating point numbers. http://www.canonware.com/ttt/2007/07/tagged-unboxed-floating-point-numbers.html, 9 2007.
4) Brent Fulgham. The computer language bench. game. http://shootout.alioth.debian.org/.
5) David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, Vol. 23, No. 1, pp. 5–48, 1991.
6) John Levon and Philippe Elie. Oprofile. http://oprofile.sourceforge.net/.
7) Huu-Duc Nguyen and Atsushi Ohori. Compiling ml polymorphism with explicit layout bitmap. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming*, pp. 237–248, New York, NY, USA, 2006. ACM.
8) Charles Oliver Nutter and Thomas E Enebo. Jruby java powered ruby implementation. http://jruby.codehaus.org/.
9) O'Reilly Media, Inc. Perl.com: The source for perl – perl development, perl conferences. http://www.perl.com/.
10) Python Software Foundation. Python programming language. http://www.python.org/.
11) Sun Microsystems. Java テクノロジ. http://jp.sun.com/java/.
12) Masahiro Tanaka. Numerical ruby narray. http://narray.rubyforge.org/index.html.ja.
13) Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. Java object header elimination for reduced memory consumption in 64-bit virtual machines. *ACM Trans. Archit. Code Optim.*, Vol. 4, No. 3, p. 17, 2007.
14) 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.
15) まつもとゆきひろ他. オブジェクト指向スクリプト言語 ruby. http://www.ruby-lang.org/ja/.
16) 松本行弘. Ruby の真実. 情報処理, Vol.44, No.5, pp. 515–521, 2003.
17) 笹田耕一, 松本行弘, 前田敦司, 並木美太郎. Ruby 用仮想マシン yarv の実装と評価. 情報処理学会論文誌 (PRO), Vol. 47, No. SIG 2(PRO28), pp. 57–73, 2 2006.
18) 青木峰郎. Ruby ソースコード完全解説. インプレス, 2002.