Vol.2011-HPC-130 No.63
2011/7/29

HPC Ruby: 静的解析に基づく Ruby の高度最適化コンパイラ

中村晃一[†1]　野瀬貴史[†1]
稲葉真理[†1]　平木敬[†1]

高性能計算分野では他の分野では使われる事の少なくなってきた Fortran・C 言語が使われ続けている．これらの言語はその生産性の低さと並列計算機向けの最適化の困難さが問題であり，高級なプログラミング言語を使用可能とする事は今後の重要な研究課題である．その様な目的の研究では並列構文を備えた専用の言語の研究・静的型付けの関数型言語の研究が主流であるが，これらが計算科学者にとって使い易いものであるとは言い難い．

我々は Ruby を用いて高性能計算を行う試みを行っている．Ruby はその記述の容易さ・高機能さから広く使われており，本言語に高性能計算に使用可能な性能を与える事の恩恵は大きい．本稿では我々の試みの第一歩として，Ruby の為の型解析手法の提案を行う．動的言語である Ruby は静的言語に比べ解析が困難であると考えられているが，部分評価手法と抽象解釈手法を組み合わせることにより十分な精度での解析を行う事が可能である．また，我々は開発した解析手法に基づいて，Ruby の実行前最適化コンパイラ **HPC Ruby** を開発した．本稿時点では単一プロセッサ向けの基本的な最適化を実装している．NAS Parallel Benchmark を用いた性能評価では最新の Ruby 処理系と比べ 100 倍以上の高速化を達成し，C 言語の性能の約 90%を実現する事が出来た．

HPC Ruby: High-Performance Optimizing Compiler for Ruby based on Static Analysis

Koichi Nakamura,[†1] Takahumi Nose,[†1] Mary Inaba[†1]
and Kei Hiraki[†1]

Computational scientists have been using Fortran and C which are being used less than before in other fields. Because these languages have low productivity and optimization of them for massively parallel computers is difficult, it is important to make high performance computation using high-level languages possible. Although, for this purpose, study of dedicated parallel languages and study of statically-typed functional languages are mainstream, these languages are not easy to use for computational scientists.

We have been studying high performance computation with Ruby. Giving high performance to Ruby brings significant benefits since Ruby is widely used because of its rich functionality and flexibly. This paper proposes a versatile method of static type analysis that is sufficiently efficient for Ruby. It combines partial evaluation framework and abstract interpretation framework to make analysis of dynamic language possible. We have implemented an ahead-of-time optimizing compiler, **HPC Ruby**, which performs classical optimization techniques based on the analysis targeting single core processors. HPC Ruby has achieved hundred times of speed-up against to latest Ruby interpreter in experimentation with Nas Parallel Benchmarks. Its performance has reached almost 90% of performance of C.

1. はじめに

高性能計算分野 (以下 HPC 分野) では長らく Fortran・C 言語が使用され続けている．自動ベクトル化技術やキャッシュ最適化技術等，HPC 向けの最適化手法がこれまで数々研究されて来たがこの分野では言語自体に大きな進化はない．これらの言語は低級で生産性が低くもはや他の分野に於いては使われる事が少なくなってきているが，匹敵する性能を持つプログラミング言語が存在しない為，計算科学者は数万行に及ぶプログラミング作業に時間を使わなければならない状況が続いている．

低級な言語の使用は性能という観点でも問題である．今後大規模化・複雑化が進んでいく並列計算機に対して，高性能で動作するプログラムを手動で記述する事は困難が伴うと予想され，大規模並列化・データ転送の最適化等の自動化技術が求められる．これらの技術を実現するには，プログラムの高度な制御フロー構造・データフロー構造を解析する事が重要となるが，低級な記述から高度な構造を逆解析する事は困難である．

以上の背景から，高級言語を用いた HPC の研究が進められている．Allen らの Fortress[1)]，Charles らの X10[2)] は並列機能を備えた HPC に特化した言語を開発しようというアプローチである．これらの言語ではユーザが明示的に並列プログラムを記述する事が可能である．既存言語からの自動並列化というアプローチでは静的型付けの関数型言語に於ける研究が

†1 東京大学
The University of Tokyo

盛んである．関数・変数の型として並列性を表現する事により処理系が自動的に並列化を行う事が可能となる．この分野の研究ではデータ並列パラダイムに基づく Chakravarty らの Data Parallel Haskell[3] が有名である．これらのアプローチの問題点は関数型言語，型推論に基づく静的型付け等の概念が，計算機科学が専門でない多くの計算科学者にとってなじみやすい物ではないという事である．

そこで，我々は Ruby を用いた HPC を実現する為の研究を行っている．Ruby はまつもとらによって開発されたオブジェクト指向スクリプト言語[4] である．Ruby はその記述の容易さ・高機能さ・柔軟さから人気があり，Web アプリケーション分野を中心として広く使われている．既に多くのユーザが存在し，計算科学者にも容易に習得できる言語である為，Ruby に HPC での使用に耐える性能を与える事の恩恵は大きい．

Ruby は動的言語に分類される．動的言語の定義は曖昧であるが一般に**動的型付け**や**動的関数束縛**の特徴を持つ．これらの特徴は実行時の文脈に応じて柔軟に振舞うプログラムを記述する事を可能とし Ruby の人気の源であるが，一方で実行時のオーバヘッドをもたらす．実行時コンパイルや動的型推論などの実行時に行う最適化技術の研究が精力的に行われているが，依然として最新の Ruby 処理系と Fortran・C の間には数十倍から数百倍の性能差が存在する．

本稿では Ruby を用いた HPC 技術研究の第一歩として，Ruby プログラムの型を静的に解析する手法の提案を行う．動的言語で記述されたプログラムの静的型解析は困難である・不可能であると考えられがちであるが，

動的言語で記述される $\not\rightarrow$ 動的に振舞う

であるので，その困難性は一般的に考えられているほど自明ではない．2 章の冒頭で詳しく述べるが，人間の記述するプログラムには一定の傾向がある事を利用すると，こと HPC 分野に於いては多くのプログラムが静的に解析可能である．提案する解析手法は部分評価手法と抽象解釈手法を相補的に組み合わせた物であり，補助的に投機的ガード命令を利用する．また，本手法は Ruby の言語仕様に変更を加える事無く実行可能であるように設計している．

我々は Ruby の為の実行前最適化コンパイラ **HPC Ruby** を開発した (3 章)．HPC Ruby は開発した解析手法に基づき種々の基本的な最適化を行う．現段階ではまず単一コア上での性能差を解消する為の最適化技術を実装している．4 章では Nas Parallel Benchmark を用いた評価結果を紹介し，HPC Ruby が Ruby プログラムを C 言語の性能に匹敵するほど高速化する事が出来る事を示す．5 章では既存研究の研究と本研究の比較を行い，6 章に於いてまとめを述べる．

2. Ruby の為の静的解析手法

本章では Ruby の為の静的型解析アルゴリズムを説明する．提案手法は部分評価手法と抽象解釈を組み合わせる事により実現される．以下，アルゴリズムの詳細に立ち入る前に我々のアルゴリズムのアイデアを具体例を用いて述べる．本研究の重要な着眼点は 1 章で述べた様に動的言語で記述される事はただちに動的に振舞う事を意味しないという事である．ここで「動的に振舞う」の意味は解析対象とする性質 (これは文脈により決まる) が，実行時に与えられる入力に真に依存して決定される場合を指す．

例えば，以下のステートメントを考えてほしい．

```
eval(STDIN.read)
```

これは標準入力から任意の Ruby プログラムを文字列として受け取りそれを実行する物である．このステートメントの戻り値の型や実行中の副作用を解析対象とした場合，このプログラムのみから静的にそれらを解析する事が不可能であるのは自明である．対照的に以下のプログラムを考えてほしい．

```
["foo", "bar", "baz"].each do |name|
    eval("def do_#{name}; end")
end
```

これは C 言語におけるマクロの様な `eval` の使い方であるが，`do_foo`，`do_bar`，`do_baz` という名の 3 つの関数を定義する．このプログラムは実行時に与えられる入力に依存しない為，コンパイル時に静的に実行する事が可能である．

さて，Ruby プログラムが前者の様な真に動的な記述であふれているならばそれを静的に解析する事は不可能であろうが，現実のプログラミング活動に於いて，実行時になるまでどのような動作が行われるか全く予想不可能であるようなコーディングを行う事は大変稀であると考えられる．動的言語を用いて記述されていようが，現実のプログラムの大部分は静的な挙動を行うはずであるという観察が本研究の動機の根底にある．

以上の観察に基づき，我々は**部分評価手法**と**抽象解釈手法**を組み合わせたプログラム解析アルゴリズムを開発した．図 1 にアルゴリズムの構成を示す．提案アルゴリズムでは 1 ステートメント毎に順に解析を行う．`analyze` は部分評価と抽象解釈を一つの束空間内で同時に行う．静的解析が不可能なステートメントが登場する可能性は常にあるが，その様な場

```
procedure analyze_program(prog):
  env ← new environment
  while prog is not empty do
    s ← first statement of prog
    analyze(env, s)
    if is_indeterminable(s)
      insert_guard_after(s)
    end if
    remove s from prog
  end while
end
```

図 1 Ruby の静的型解析アルゴリズム

合の対処として我々のアルゴリズムは**投機的ガード命令**を利用する (`insert_guard_after`). ガード命令の利用は，任意のプログラムに対してアルゴリズムを実行可能とする為の物であり，提案アルゴリズムにとって重要な位置付けはない．あくまで補助的な物である．

以下の章でアルゴリズムを具体的に述べる．

2.1 Ruby プログラムのモデル

最初に，Ruby プログラムのモデルを述べる．Ruby プログラムの基本構成要素は変数，オブジェクト (数値，文字列，配列等)，ブロックである．変数はその値 (オブジェクト) と型 (Ruby ではクラスと呼ぶ) を持つ．Ruby の大きな特徴として，クラス自身もオブジェクトであり操作可能である．ブロックは 0 個以上の引数と，ステートメントの列から構成される．

任意の Ruby プログラムはステートメントの列として構成される．Ruby は 90 種類以上のステートメントを持つが，本質的には以下のステートメントを組み合わせる事で構成出来る．

(1) メソッド呼び出し
(2) 変数への代入
(3) 条件分岐
(4) while 型ループ

後者 3 つのセマンティクスは C 言語等のそれと同様なので割愛する．以後，アルゴリズムの説明はこれらに限定をして行う．

Ruby のメソッド呼び出しはメッセージパッシングモデルに基づいており，全てのメソッド呼び出しは以下の様に表される．

```
recv.method(a0, a1, ...)
```

ここで，`recv`, `a`N，はオブジェクトでありレシーバオブジェクトと引数オブジェクトと呼ばれる．`method` はメソッドの名前である．

一部のメソッドはブロックを最後の引数として受け取る事が可能である．

```
recv.method(a0, a1, ...) do |b0, b1, ..|
   ... statements ...
end
```

ここで `b`N はブロックの引数である．

以下，注意点をいくつか述べる．

- Ruby ではモジュール・クラス・メソッドの定義も内部的にはメソッド呼び出しとして扱われる．例えば，メソッドの定義は `define_method` メソッドの呼び出しとして処理される．
- 複合オブジェクトのメンバ変数へのアクセスはメソッド呼び出しとして扱われる．例えば，`x.m = 1` というメンバ変数への代入は `m=`メソッドの `x` に対する呼び出しとして処理される．

Ruby でのメソッド呼び出しは以下の手順で行われる．

(1) レシーバオブジェクトを評価する．
(2) 引数を評価する．
(3) レシーバオブジェクトのクラスとメソッド名に基づいて，メソッド本体の探索を行う．
(4) メソッド本体の実行を行う．

以後述べるアルゴリズムに於いてメソッドの探索手順の詳細について考慮する必要はない．クラスとメソッド名によって決定されるという事実をここでは確認しておく．

2.2 部分評価と抽象解釈

部分評価手法[5] とは，プログラムの中でコンパイル時に評価可能である箇所をあらかじめ実行しておく事でその実行時間を削減する最適化のアプローチである．また，抽象解釈手法[6),7] とは，プログラムの型・副作用など特定の性質を解析する場合に使用される古典的な

手法であり，解析対象の属性値からなる束と呼ばれる代数構造を用意しその上で擬似的にプログラムを実行する事で行う．

本手法では，実行時間の削減ではなくモジュール・クラス・メソッドの定義をコンパイル時に得る事を目的として部分評価を行う．そして得た情報を用いて抽象解釈を行う事で型の解析を行う．例として以下のプログラムを用いて説明する．

```
class A
   def f
      ...
   end
end
x = A.new
y = x.f()
```

このプログラムではクラス A を定義し，A のオブジェクトを生成した後，メソッド f を呼び出している．この場合クラス定義部を部分評価してしまえば，その後の A.new や x.f() で呼ばれるメソッド・戻り値の型等を抽象解析により調べる事が出来る．

2.3 束の構成

あるステートメントの実行中にモジュール・クラス・メソッドの定義が行われるか否かを事前に知る事は難しいので，提案手法では部分評価と抽象解釈を融合し同時に実行する方法を取る．以後，この両者を総称して**擬似実行**と呼ぶ事にする．我々は図 2 に示す束構造を定義する事により擬似実行アルゴリズムを実現した．

図に於いて，*Unknown* は型を未解析である事，*Dynamic* は型が真に動的で決定不可能である事を表し，束の最小元・最大元に対応する．すなわち任意の束の元 x に対し，結び演算 $\vee$ は

$$Dynamic \vee x = x \vee Dynamic = Dynamic$$
$$Unknown \vee x = x \vee Unknown = x$$

となる．最小元の直上には定数リテラル (0，1，0.1，"foo"，[] 等) と定数変数 (String，Fixnum 等) が配置される．以後，これらを総称して定数元と呼ぶ．定数元の上は型の集合の部分集合関係による半順序構造となる．誤解のない限り，以降はこれら束上の元の事を単純に型と呼ぶ．定数元 c と型集合 t の大小関係は

$$c < t \ \leftrightarrow \ c \text{の型} \in t$$

として定義される．

複合型の型は、その型の名前を表す定数変数とメンバ変数の型の組を用いて C{m1:t1,m2:t2} 等と表記する．ここで C は型の名前を表し，m1，m2 はメンバ変数名，t1，t2 はそれらの型を表す．例えば，以下のプログラムに於いて

```
x = A.new                    # A 型のオブジェクトを生成
x.instance_eval { @m = 0 } # x に m というメンバ変数を追加
```

一行目実行後の x の型は A{} であり，二行目実行後の型は A{m:0} となる．ここで異なるメンバ変数を持つ複合型は異なる型として扱う．すなわち，今の例に於いて A{} と A{m:0} は異なる型として取り扱う．

配列型 (Array) など，Ruby 標準の型についても同様に取り扱う．例えば [1,2,5.0] というリテラルの型は Array{element:{Fixnum,Float}} と考えればよい．本章においては簡便にこれを Array{Fixnum,Float} と表記する事とする．

型 a，b の結び演算は

$$a \vee b \quad \equiv \quad a \leq x, b \leq x \text{となる最小の} x$$

と定義する．例えば

$$1 \vee 2 = \{\texttt{Fixnum}\}$$
$$1 \vee 0.5 = \{\texttt{Fixnum,Float}\}$$
$$1 \vee \{\texttt{Fixnum}\} = \{\texttt{Fixnum}\}$$
$$\{\texttt{Array\{Fixnum\}}\} \vee \{\texttt{Array\{Float\}}\} = \{\texttt{Array\{Fixnum,Float\}}\}$$
$$\{\texttt{A\{foo:\{Fixnum\}\}}\} \vee \{\texttt{A\{foo:\{Float\}\}}\} = \{\texttt{A\{foo:\{Fixnum,Float\}\}}\}$$
$$\{\texttt{A\{foo:\{Fixnum\}\}}\} \vee \{\texttt{A\{bar:\{Float\}\}}\} = \{\texttt{A\{foo:\{Fixnum\}\},A\{bar:\{Float\}\}}\}$$

等となる．

以上の束の定義だと *Unknown* から *Dynamic* までに無限上昇列が存在し得る．入れ子になった複合型のネストの深さ・型集合の要素数・メンバ変数の数に上限が存在しないからである．次章で述べる擬似実行アルゴリズムが必ず停止する為には，全ての上昇列が有限でなければならないので，現在のアルゴリズムでは三数の上限を設定し，上限に達した場合に達した場合に *Dynamic* に丸める事を行っている．型集合の要素数・メンバ変数の数については実際のアプリケーションにおいても，大部分が有限個に収まるので問題とはならないが，

Vol.2011-HPC-130 No.63
2011/7/29

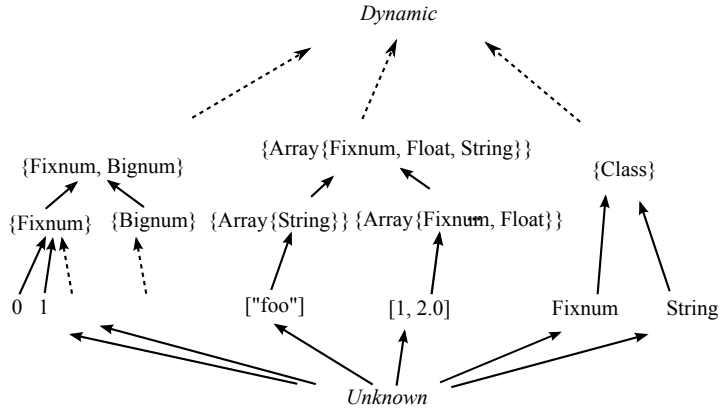

図 2 解析に用いる束構造

任意の深さでネストするデータ構造 (木構造等) を利用するプログラムでは解析精度が低下してしまう．再帰型の検出を行う事によりこの精度を向上させる事は可能であると考えられるが，本稿に於いてはこれを将来課題とし丸めによるアルゴリズムを用いて評価を行う．

2.4 擬似実行

本章では，Ruby のステートメントの擬似実行アルゴリズムについて述べる．一般的な部分評価アルゴリズムや抽象解釈アルゴリズムでは，制御フロー解析・データフロー解析の順に行い，変数使用定義グラフ上で擬似的にプログラムを評価する事を行う[6),7)]．しかし，動的関数束縛を行う Ruby に於いては制御フローのみを単独で解析する事が困難である．

そこで，提案手法では制御フロー解析を行わず，Ruby プログラムの構文木上で直接擬似実行を行う．図 3，図 4 が章で述べた 4 つのステートメントに対する擬似実行アルゴリズムである．関数 `analyze` は実行環境と式を受け取り，式を実行後の環境と式の評価値の型を返す．定数リテラル・変数に対する `analyze` の定義は自明であるのでここでは省略する．アルゴリズム中の $\vee$ 演算子は束上での結び演算を表しており，実行環境同士の結び演算は全ての変数についてそれぞれ行う事を表す．ここで，代入文の擬似実行関数中の *topscope* 変数はその代入分が独立した 1 ステートメントである場合に真となる変数である．これは 2.3 章では定数変数を型として用いているが，Ruby の仕様上は定数変数を上書きする事が可能である事に対応する物である．

図 3 内の $\alpha(f)$ は，C 言語で実装された Ruby の組み込み関数 f に対応する擬似実行を行う関数を表す．C 言語による実装を自動的に解析する事は現在困難であるので，$\alpha(f)$ は全て手動で実装を行う．その際，$\alpha(f)$ の実装は次の条件を満たす様にする．

(1) 任意の入力に対して停止する．
(2) 与えられた引数の値・型に対応する f の戻り値の値・型を返す．
(3) 変数定義やメソッド定義が行われた場合には，環境を更新して返す．
(4) 解析不能が不能ある場合には，解析を中断する (後にガード命令が挿入される)．

2.3 章で述べた束には定数元を組み込んであるため，与えられた引数が定数値である場合にはその命令を具体的に解釈する事が出来る．これによって抽象解釈と部分評価を区別せずに実行する事が出来る．

図 5 に代表的な組み込み関数に対する実装を例示する．それぞれメソッドの定義，固定長整数の加算，n 回繰り返しに対応する関数である．実際の実装では引数の数の検査なども行う．

2.5 投機的ガード命令

真に動的に振舞う静的解析が不可能なステートメントが存在した場合には，以後のステートメントの解析を継続する事が出来ない．しかし，ステートメントの挙動についてある程度の予想が可能である場合には**投機的ガード命令**を使用する事により解析を継続する事が出来る．ガード命令の使用は Smalltalk-80[8)] や Self[9),10)] の処理系に採用された事で有名である．例えば `n - 1` という式を考えてほしい，`n` の型が完全に解析出来ないがおそらく整数であると予想可能であるならば，以下の様に検査を挿入する事で検査を通過した部分について最適化を行う事が出来る．

```
if n is an Integer then
    prim_sub(n, 1) # 組み込みの整数演算
else
    n - 1          # メソッドの呼び出し
end
```

提案手法に於いては，プログラムの構造に関する副作用 (既存の定数変数の書き換え，モジュール・クラス・メソッドの定義) が解析不能なステートメント内で発生する事はないという仮定をおき，これら副作用に対する検査を挿入する事により以後の解析を継続する．前者は，検査命令前後での変数表の上書きを調べる事によって実装をし，後者は `define_method` 命令の呼び出し他，これらの定義を行うメソッドの呼び出し箇所で検出を行っている．実行

Vol.2011-HPC-130 No.63
2011/7/29

procedure analyze($env, recv.method(arguments)$):
  $(env, recv\_ty) \leftarrow$ analyze$(env, recv)$
  if $recv\_ty = Dynamic$
    exit(indeterminable statement)
  end if
  for all a in $arguments$ except block **do**
    $(env, a\_ty) \leftarrow$ analyze(env, a)
  end for
  $ret\_ty \leftarrow Unknown$
  $ret\_env \leftarrow \emptyset$
  for all r in $recv\_ty$ **do**
    $f \leftarrow$ lookup_method$(r, method)$
    $tmp\_env \leftarrow$ copy env
    if f is builtin function
      $(tmp\_env, t) \leftarrow \alpha(f)(tmp\_env,$ arguments and block$)$
    else
      $tmp\_env \leftarrow tmp\_env \cup$ arguments and block
      $(tmp\_env, t) \leftarrow$ analyze$(tmp_env,$ body of $f)$
    end if
    $ret\_ty \leftarrow ret\_ty \vee t$
    $ret\_env \leftarrow ret\_env \vee tmp\_env$
  end for
  return (ret_env, ret_ty)
end

図 3 メソッド呼び出しの擬似実行

procedure analyze($env, x = expr$):
  if x is a constant-variable **and not** $topscope$
    exit(indeterminable statement)
  end if
  $(env, type) \leftarrow$ analyze$(env, expr)$
  return $(env \cup \{x = type\}, type)$
end
procedure analyze($env,$ `if` c `then` $e1$ `else` $e2$ `end`):
  $(env, type) \leftarrow$ analyze(env, c)
  if $type =$ `false` **or** $type =$ `nil`
    return analyze$(env, e2)$
  end if
  $e1\_env \leftarrow$ copy env
  $e2\_env \leftarrow$ copy env
  $(e1\_env, t1) \leftarrow$ analyze$(e1_env, e1)$
  $(e2\_env, t2) \leftarrow$ analyze$(e2_env, e2)$
  return $(e1\_env \vee e2\_env, t1 \vee t2)$
end
procedure analyze($env,$ `while` c $body$ `end`):
  $ret\_ty \leftarrow$ `nil`
  $changed \leftarrow$ `true`
  while $changed$ **do**
    $(env, t) \leftarrow$ analyze(env, c)
    $(env, t) \leftarrow$ analyze$(env, body)$
    $ret\_ty \leftarrow t \vee ret\_ty$
  end while
  return ret_ty
end

図 4 代入・条件分岐・ループの擬似実行

```
procedure α(define_method)(env, {class, name, func}):
  if class is not a constant-variable or not topscope
    exit(indeterminable statement)
  end if
  add method to env
  return (env, nil)
end
procedure α(fix_plus)(env, {a, b}):
  if b ≤ {Fixnum, Bignum}
    return (env, {Fixnum, Bignum})
  else
    ...
  end if
end
procedure α(fix_times)(env, {n, block}):
  ret_ty ← nil
  if n is constant integer
    for i ← 0 to n − 1 do
      (env, ret_ty) ← analyze(env, block)
    end for
  else
    changed ← true
    while changed do
      (env, t) ← analyze(env, body)
      ret_ty ← t ∨ ret_ty
    end while
  end if
  return ret_ty
end
```

図 5 組み込み関数の擬似実行

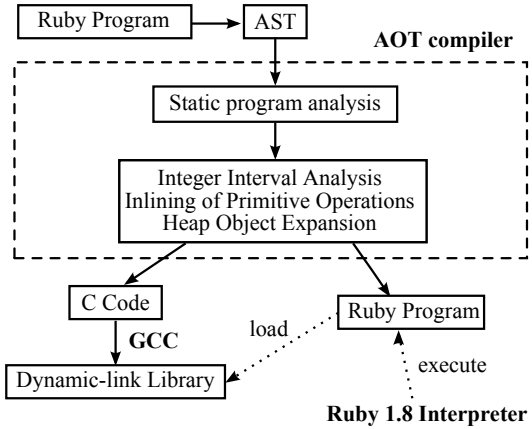

図 6 HPC Ruby コンパイラの構成

時に副作用が発生した場合には最適化されていない元のプログラムを実行するコードを生成する事により，プログラムの正しさを保って投機的な最適化を行う事が出来る．

3. HPC Ruby コンパイラ

我々は開発した型解析手法に基づき，Ruby の実行前 (AOT) 最適化コンパイラを開発した．HPC Ruby コンパイラは Ruby 1.8 の処理系に組み込む形で実装を行った．図 6 にコンパイラの構成を示す．

与えられた Ruby プログラムは，まず構文解析器により抽象構文木へと変換される．続いて，2 章で述べた型解析を行い，その情報を利用して制御フロー解析・データフロー解析を行う．型解析時にメソッドの束縛情報も得られているので，これらの解析は既存のアルゴリズムをそのまま実行する事が可能である．続いて以後の章で述べる最適化を行ったのち，最適化されたステートメントを C 言語のプログラムへと変換する．生成された C プログラムは GNU Compiler Collection (GCC) によりダイナミックリンクライブラリへとコンパイルされる．また，元の Ruby プログラムは生成されたライブラリの関数を使用するように修正が加えられる．生成されたプログラムは Ruby 1.8 により実行する事が出来る．

3.1 最適化手法

我々は，数値計算プログラムに焦点を絞って以下に述べる最適化技術を実装した．

3.1.1 整数範囲解析

Ruby は 31-bit 整数を表す `Fixnum` と任意長整数を表す `Bignum` の 2 つの整数型を標準で備える．`Fixnum` 型の値と `Bignum` 型の値の変換は自動的に行われるが，その為のオーバヘッドが発生する．2 章で述べた型解析アルゴリズムでは整数変数の変域が `Fixnum` に収まるか否かを判定する事が出来ない為，我々は整数範囲解析の実装を行った[11]．すなわち，以下の様な範囲演算を用いて抽象解釈する事により行う．

$$[a,b]+[c,d]=[a+c,b+d]$$
$$[a,b]-[c,d]=[a-d,b-c]$$
$$[a,b]\times[c,d]=[\min\{ac,ad,bc,bd\},\max\{ac,ad,bc,bd\}]$$

3.1.2 基本命令の置き換え

型解析の結果に基づき，固定長に収まる整数の四則演算やループ命令を C 言語の整数演算や for ループを用いた等価な実装に置き換える事を行う．例えば，

```
n.times do
    ... do something ...
end
```

に対して，n が `Fixnum` であると解析出来たならば

```
int i, n_ = FIX2INT(n);
while (i = 0; i < n_; i++) {
    ... do something ...
}
```

というコードを生成する．

3.1.3 オブジェクトの非ボックス化

ボックス化されたオブジェクトの非ボックス化を行う．現在の実装では `Float` 型の変数の C 言語の double 変数への置き換えと，`Array{Float}` 型の配列の double 型の配列への置き換えを行う．ここで，前者は常に安全に行う事が出来るが後者についてはエスケープ解析[12]と，配列添えへの範囲を整数範囲解析に基づいて解析し安全性の検査を行う．

4. 評　　価

我々は NAS Parallel Benchmak (NPB) 3.0[13] の 7 つの科学計算ベンチマーク (BT，CG，FT，IS，LU，MG，SP) を用いて HPC Ruby コンパイラの評価を行った．Java で記述された NPB プログラムを手動での最適化を施さずに機械的に Ruby プログラムに変換し，それを利用した．今回は型解析に基づく基本最適化の効果を評価する為，単一スレッド実行での性能を測定した．

比較には GNU Compiler Collection (GCC) 4.5.1，Java 1.7.0 と現在広く使われている以下の Ruby 処理系を採用した．

Ruby1.8

Ruby1.9 が登場するまでの標準の処理系．構文木を再帰的に辿りながら実行を行う．

Ruby1.9

最新の標準処理系であり Ruby の為に開発された仮想マシン Yet Another Ruby VM (YARV)[14] をバックエンドとして採用している．本処理系では Ruby プログラムは実行前にバイトコードへと変換され，それを YARV で実行する．YARV はメソッド探索のキャッシングや数値演算などの組み込み処理の専用命令化等の高速化技術を実装している．

JRuby 1.5[15]

Java Virtual Machine (JVM)[16] をバックエンドとして採用した Ruby 処理系．JVM は HotSpot[17] 等の高度な JIT 最適化を行う．

Rubinius1.0[18]

Low Level Virtual Machine (LLVM)[19] をバックエンドとして採用した Ruby 処理系．LLVM は高度な動的最適化，静的最適化を実装する近年注目されている仮想マシンである．

図 7 がベンチマーク結果である．縦軸は Mops(Million Operations Per Second) 値の対数表示である．これらの結果は我々の HPC Ruby コンパイラが Ruby プログラムの実行時性能を劇的に改善する事を示している．HPC Ruby の性能は Ruby 1.8 に比べ 200 倍以上，Ruby 1.9 に比べ 100 倍以上高く，C 言語の性能の約 90%に迫っている．数値計算プログラムでは実行時の型チェック，動的メソッド呼び出し，浮動小数点数のボックス化のオーバヘッドが大きくなるが，今回の実験ではこれらを完全に除去する事が出来ていた．

NPB の最適化されたコードでは投機的ガード命令が一度も生成されていなかった．これ

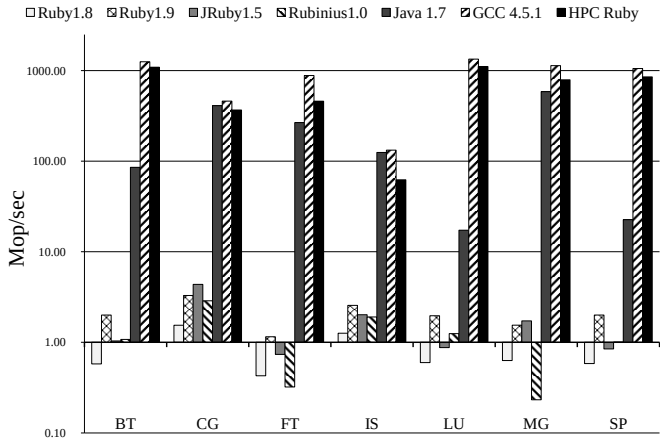

図 7 Nas Parallel Benchmark 結果

は，NPB には動的に振舞う解析不可能なステートメントが存在しなかった為である．従ってガード命令によるオーバヘッドは発生していない．

5. 関 連 研 究

高級言語を利用した HPC に関する研究には主に 2 つのアプローチがある．Allen らの Fortress[1]，Charles らの X10[2] は並列機能を備えた HPC に特化した言語を開発しようというアプローチである．これらの言語ではユーザが明示的に並列プログラムを記述する事が可能である．既存言語からの自動並列化というアプローチでは静的型付けの関数型言語に於ける研究が盛んである．型システム上で並列性を表現する事により処理系が自動並列化を行う事が可能となる．データ並列パラダイムに基づく Chakravarty らの Data Parallel Haskell[3] が有名である．本研究はこれらとは異なり，計算機科学が専門でない多くの計算科学者にとって使いやすい Ruby を利用する事を目的としている．

動的言語向けの最適化手法としては，実行時 (JIT) 最適化手法[20] が最も成功しているアプローチであり多くの仮想マシンが実装をしている．実行時に得られる具体的な情報を用いる事で，JIT コンパイラは高効率な機械語を生成する事が可能となる．JVM[16] は長く JIT 技術研究の中心であり，HotSpot JVM[17] は特に有名である．LLVM[19] はその性能の高さから近年注目され，既存の言語処理系のバックエンドを置き換えるものとして利用されている．Ruby に於いてもこれらの仮想マシンを利用した言語処理系が開発されており，JRuby[15] は JVM を，Rubinius[18] は LLVM をバックエンドとして採用している．HPC Ruby は静的に最適化を行うコンパイラであり，これら JIT 技術と対立するものではない．事前最適化ではバイトコードに落とす前の段階で変換を行えるため，JIT に比べより積極的なコード変換が可能である．

6. ま と め

HPC 分野に於いては長らく C，Fortran が使われ続けている． 高級な言語を利用した HPC は，計算科学者のプログラミング労力を軽減し，処理系による高度な自動最適化を可能とする為に重要である．

本稿に於いて，我々は Ruby を用いた HPC 研究の為の要素技術として，Ruby の為の静的型解析手法を開発した．提案手法は人間の記述するプログラムに一定の傾向がある事を利用し，部分評価手法と抽象解釈手法を組み合わせる事により十分な精度を実現する．また，我々は提案手法に基づき Ruby の静的最適化コンパイラ HPC Ruby を開発した．Nas Parallel Benchmark では単一スレッドでの評価に於いて，C 言語と同等の速度を達成可能である事を示した．

6.1 将 来 課 題

2.3 章で述べた型解析の束構造の丸め手法には改善の余地がある．差分方程式に基づく計算等，配列をデータ構造として用いる事の多い HPC では現在の実装でも多くのアプリケーションが存在するが，再帰的な型に対応する事で HPC Ruby の適用可能性をより広げる事が可能である．

Ruby は高級な構文を持ちプログラムの構造を解析する事が低級な言語に比べて容易であると考えている．本研究に続いて，我々は Ruby の為のループ依存解析・並列性解析手法を研究しており，それに基づいて HPC Ruby に自動並列化機能を実装する．大規模並列計算に於いて重要となるデータ転送最適化技術の研究も同時に行う．

現実のアプリケーションへの応用としては現在計算天文学者との共同研究を行っており，重力多体計算の Ruby による実装を進めている．

参 考 文 献

1) Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J., Ryu, S., SteeleJr, G., Tobin-Hochstadt, S., Dias, J., Eastlund, C. et al.: The Fortress language speci-

fication, *Sun Microsystems*, Vol.139, p.140 (2005).
2) Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., Von Praun, C. and Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing, *ACM SIGPLAN Notices*, Vol. 40, No. 10, ACM, pp. 519–538 (2005).
3) Chakravarty, M., Leshchinskiy, R., Jones, S., Keller, G. and Marlow, S.: Data Parallel Haskell: a status report, *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, ACM, pp.10–18 (2007).
4) Flanagan, D. and Matsumoto, Y.: The ruby programming language (2008).
5) Jones, N., Gomard, C. and Sestoft, P.: *Partial evaluation and automatic program generation*, Peter Sestoft (1993).
6) Cousot, P. and Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ACM, pp.238–252 (1977).
7) Wegman, M. and Zadeck, F.: Constant propagation with conditional branches, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol.13, No.2, pp.181–210 (1991).
8) Deutsch, L. and Schiffman, A.: Efficient implementation of the Smalltalk-80 system, *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ACM, p.302 (1984).
9) Chambers, C. and Ungar, D.: Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs, *LISP and Symbolic Computation*, Vol.4, No.3, pp.283–310 (1991).
10) Chambers, C. and Ungar, D.: Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language, *ACM SIGPLAN Notices*, Vol.24, No.7, pp.146–160 (1989).
11) Moore, R.: Interval analysis, *Englewood Cliffs, New Jersey* (1966).
12) Choi, J., Gupta, M., Serrano, M., Sreedhar, V. and Midkiff, S.: Escape analysis for Java, *ACM SIGPLAN Notices*, Vol.34, No.10, pp.1–19 (1999).
13) Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Frederickson, P., Lasinski, T., Schreiber, R. et al.: The NAS parallel benchmarks, *International Journal of High Performance Computing Applications*, Vol.5, No.3, p.63 (1991).
14) Sasada, K.: YARV: yet another RubyVM: innovating the ruby interpreter, *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM, p.159 (2005).
15) Nutter, C. and Enebo, T.: Jruby java powered ruby implementation, *URL http://jruby. codehaus. org.*
16) Lindholm, T. and Yellin, F.: *Java Virtual Machine Specification*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition (1999).
17) Paleczny, M., Vick, C. and Click, C.: The java hotspot TM server compiler, *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*, USENIX Association, p.1 (2001).
18) Phoenix, E. et al.: Rubinius: The Ruby Virtual Machine, *URL http://rubini. us.*
19) Lattner, C. and Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation (2004).
20) Ishizaki, K., Kawahito, M., Yasue, T., Takeuchi, M., Ogasawara, T., Suganuma, T., Onodera, T., Komatsu, H. and Nakatani, T.: Design, implementation, and evaluation of optimizations in a just-in-time compiler, *Proceedings of the ACM 1999 conference on Java Grande*, ACM, pp.119–128 (1999).