HPC Ruby Compiler

中村 晃一

自己紹介

- 中村 晃一
- 東京大学情報理工学系研究科コンピュータ科学専攻
- 平木研究室修士2年

- 研究分野：コンパイラ最適化
  - データ転送・配置最適化技術
  - 動的言語のプログラム解析・最適化技術

The Art of Computational Science

How to build a computational lab

Search this site

日本語 Deutsch Espa□l Français Italiano Nederlands T◆k◆

What's New?

- 2009, July 13: Grav-Sim, a C++ port of an extended subset of ACS, by Mark Ridler.

Questions?

Below you can read our various book volumes directly on the web. If you prefer to download a copy, you can choose one of our ACS release versions, which also include the computer programs discussed in the text. We are making all our ACS (Art of Computational Science) material available under the conditions of our ACS open source license. See the What is New? page for a description of new developments, and the FAQ page for answers to frequently asked questions.

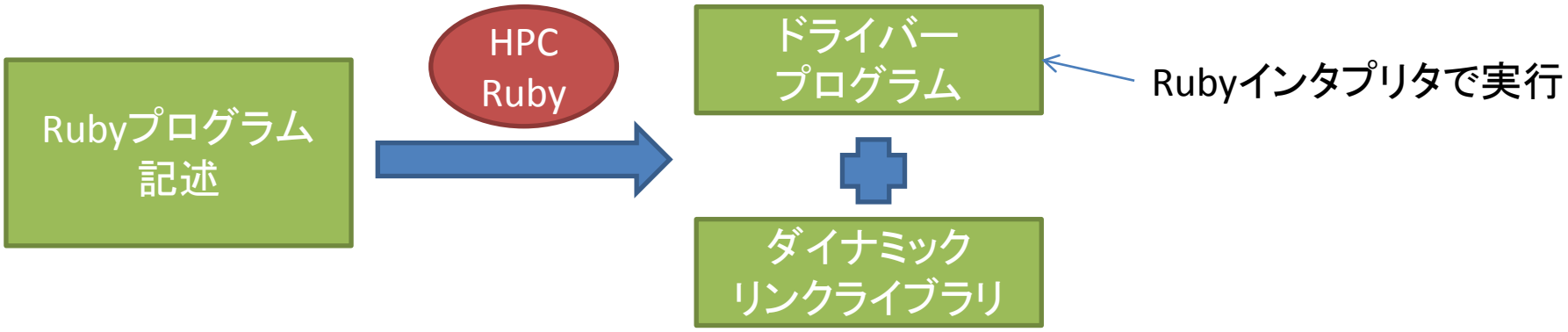
- ACS: Introduction
- An ACS Project: Maya, an Open Lab for Dense Stellar Systems
- The ACS Toolbox
- The first ACS manuscript: Moving Stars Around
- Links to Literature and Related Projects

ACS: Introduction

Having access to the source code is one thing. Having access to the tacit knowledge that went into the process of writing the source code is quite something else. In our ***Manifesto*** below, we describe the philosophy behind the ACS initiative. Briefly, we want to go beyond the notion of *open source* to the much wider notion of *open*

HPC Ruby Compiler

- 静的解析に基づく、Rubyの実行前最適化コンパイラ
- HPC分野でのFortran・Cの置き換えを目指している



HPC分野における高級言語の必要性

- Fortran･C言語の使用を続ける事は限界
  - 低生産性
    - 数万行に及ぶ開発負担
    - 他分野との協働が困難
  - 高度な最適化の困難性
    - 自動並列化
    - データ配置･転送最適化

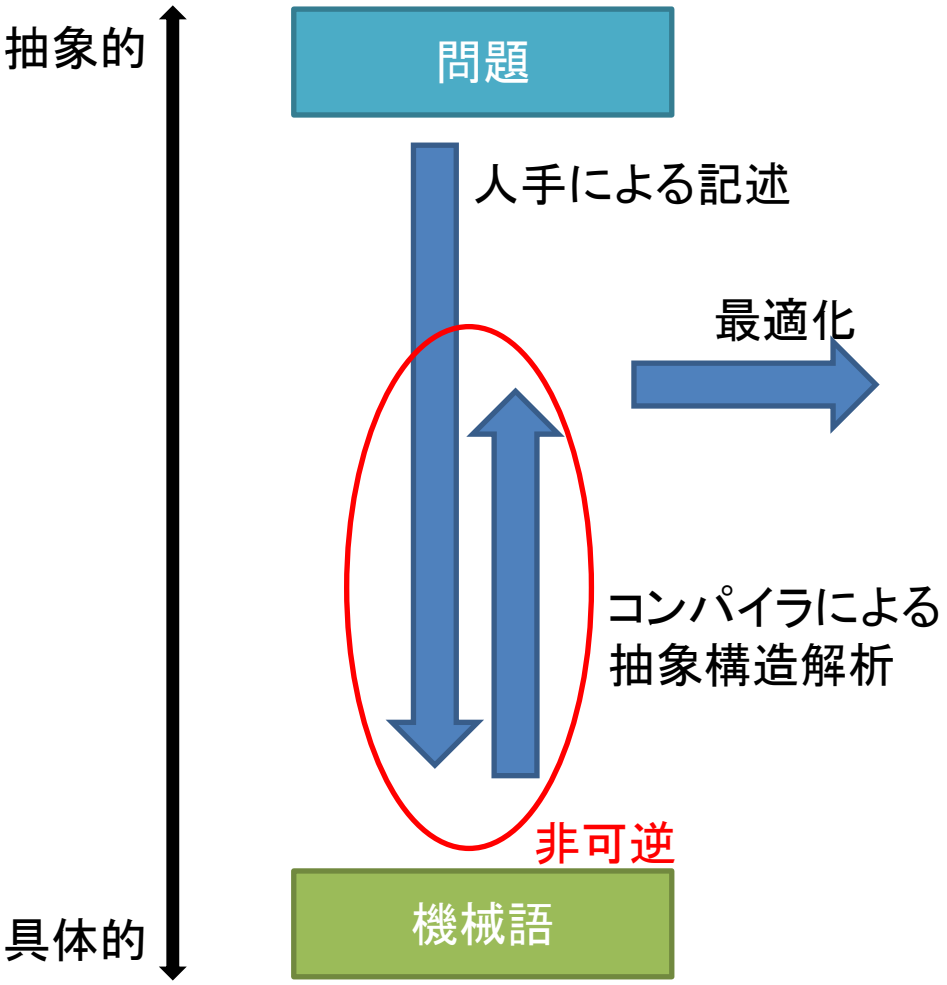
本研究の目的

- 動的言語によるHPCの為の基盤技術の開発
- コンパイラを開発し評価を行う
- 動的言語一般に汎用的に使える技術を開発する

動的言語選択の理由

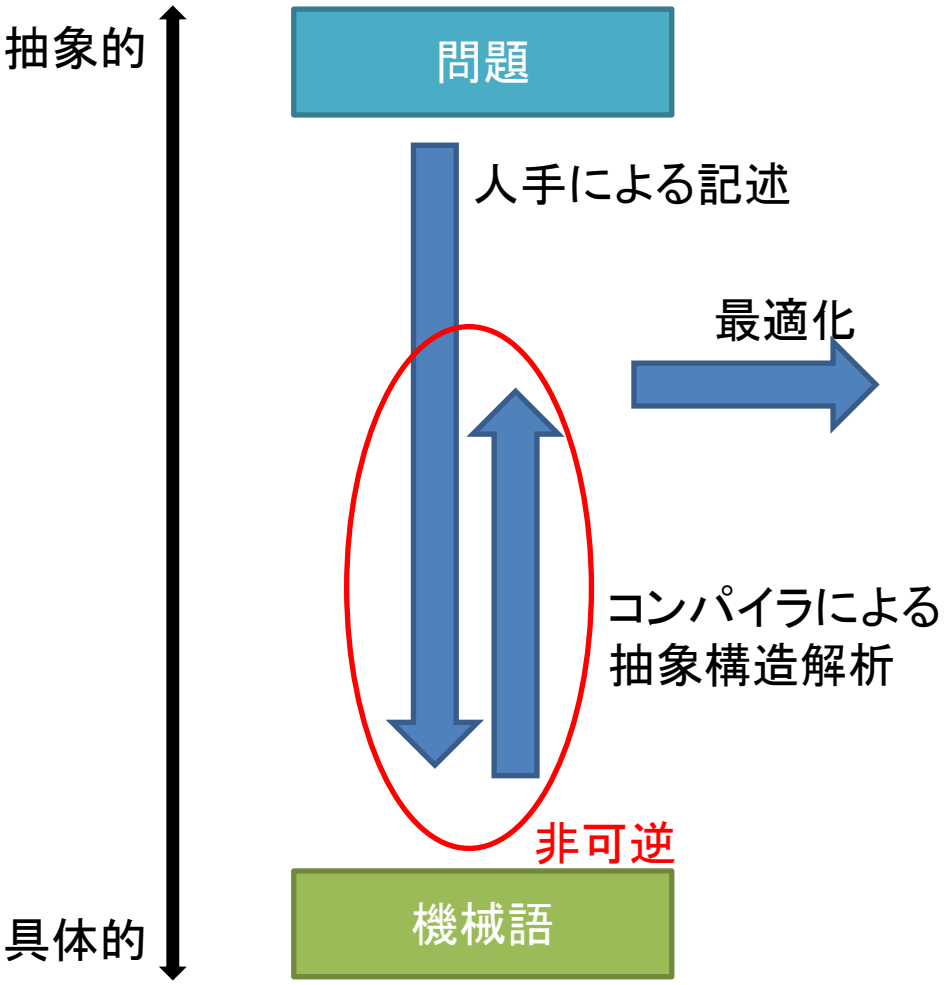
- 学習コストが大変低い
  - 計算科学者にとってのメリット大
- 表現が高級である
  - 抽象的な記述を可能とする
- 動的・柔軟である
  - メタプログラミング機能、リフレクション機能

低級言語使用による情報の損失



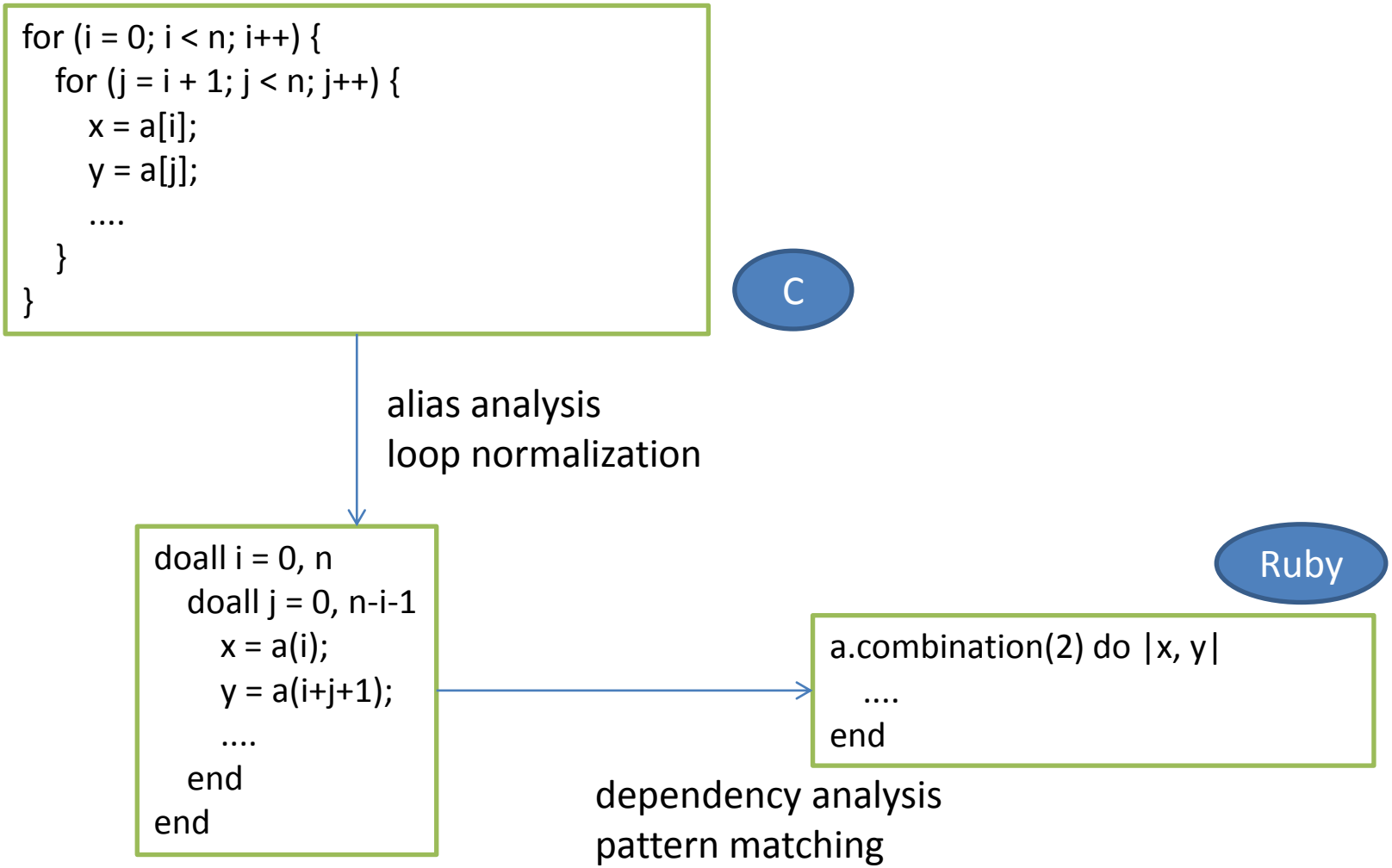
- 失われる情報
  - アルゴリズムの選択可能性
  - データ構造の選択可能性
  - 並列性

低級言語使用による情報の損失



- 失われる情報
  - アルゴリズムの選択可能性
  - データ構造の選択可能性
  - 並列性
- 言語の高級性、動的性がHPCに貢献

Example: 2体のイテレーション



HPC分野でのプログラミング言語に求められる事

1. 高性能である事
2. 高生産性である事
   - 1アプリケーション～数万行
   - 大量の書き捨て
3. 寿命の長さ
   - 30年前に書かれたFortranプログラムが未だに使用される
4. 学習コストの低さ
   - 計算科学者はプログラミングのファンではない

既存のアプローチ(専用言語)

- High Performance Fortran
- Unified Parallel C
- Xcalable MP
- Fortress
- CUDA
- OpenCL
- etc.

既存のアプローチ(専用言語)

- 特定のアーキテクチャに強く依存している
  - ex. 10~20年後のSIMDアーキテクチャはOpenCLのモデルで記述出来るのか？？
- 生産性は非常に低い
- 言語自体の学習コストはそれほど高くない
- アーキテクチャに詳しくないと高性能は難しい

既存のアプローチ(関数型言語)

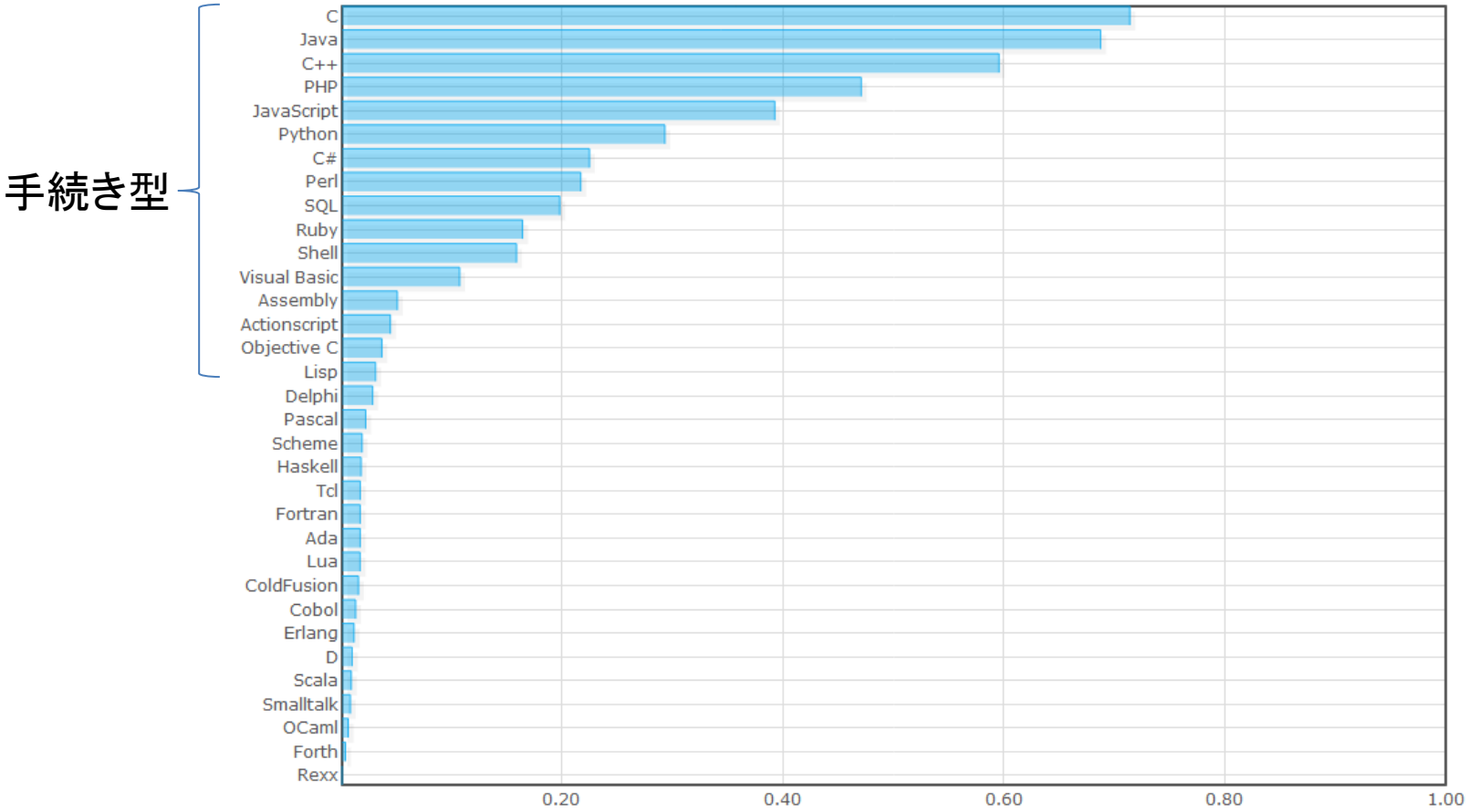
- GHC CUDA backend
- Haskell/repa
- Data Parallel Haskell
- Scala (parallel object)
- Clojure (software transactional memory)

既存のアプローチ(関数型言語)

- 良く言われる事
  - チャーチ・ロッサーの合流性により並列性を完全に引き出す事が可能
  - フォン・ノイマンボトルネックが発生しない
  - 副作用が無いから並列化が容易
  - 型検査によりプログラムのバグが発見出来る
- …

既存のアプローチ(関数型言語)

- 関数型言語は主流に成り得るのか？？？



http://langpop.com/ より

既存のアプローチ(関数型言語)

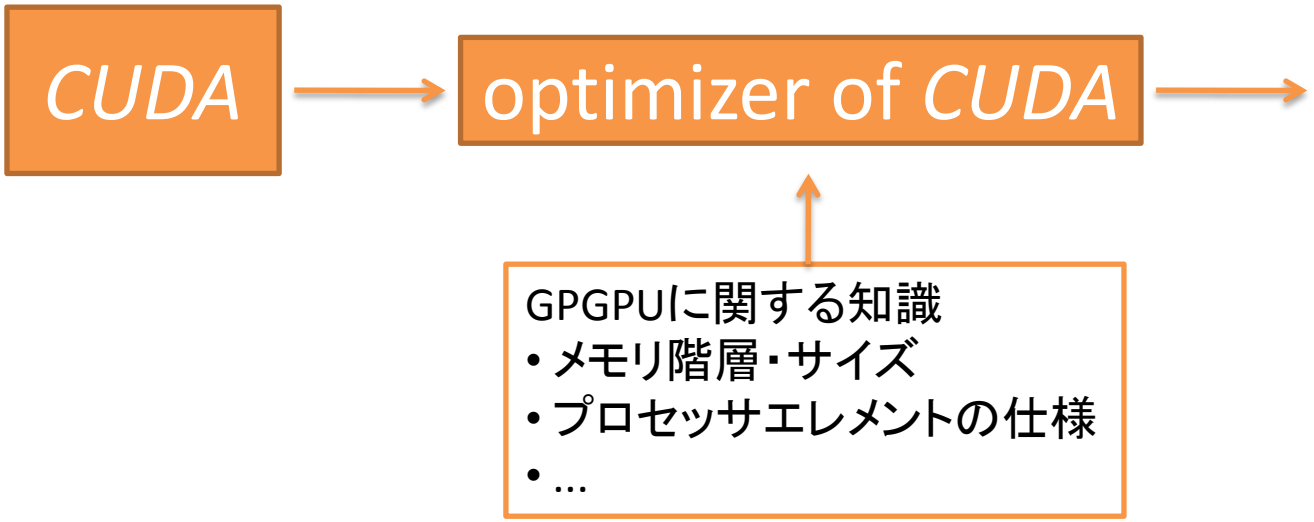
- 関数型言語は主流に成り得るのか？？？
  - 関数型言語が登場してから約50年
  - 私は本質的な問題があると考えている

動的言語はどうか？

- 代表言語: Perl, Ruby, Python, PHP, ...
  1. 高性能である事　　？？
  2. 高生産性である事　　◎
  3. 寿命の長さ　　〇？
  4. 学習コストの低さ　　◎

Domain Specific最適化

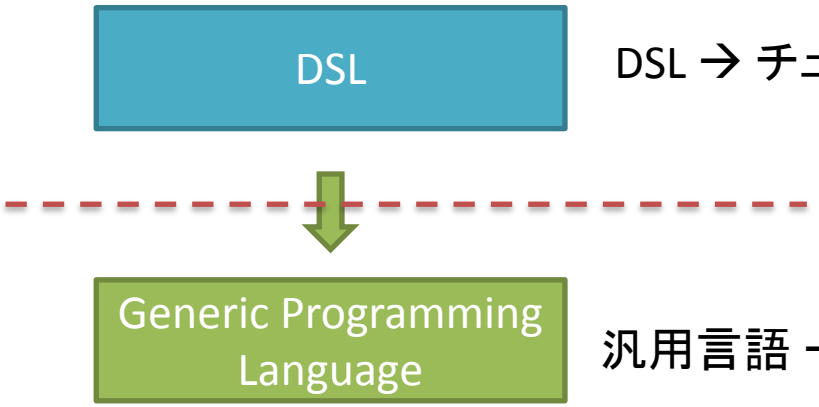
- 真に良い最適化の為に必要な事
- 領域固有知識を利用した最適化



コンパイラ作者の完全雇用定理

Universal optimizing compiler does not exist for Turing-complete language.

- チューリング不完全な言語は厳密な最適化が可能な場合がある
  - (cf. 正規表現の最小受理機械の構成)



DSL → チューリング完全でなくてよい

汎用言語 → チューリング完全

HPC分野におけるDSL

- HPCのプログラム記述におけるタスクいろいろ
  - データ配置の記述
  - データ通信の記述
  - プロセス配置の記述
  - 同期制御の記述
  - 計算カーネルの記述
  - etc.
- RubyはEmbedded DSLの作成が容易
  - 最適化器もRubyのレイヤーで作成出来る

Rubyは
- 学習コストが大変低い・書きやすい

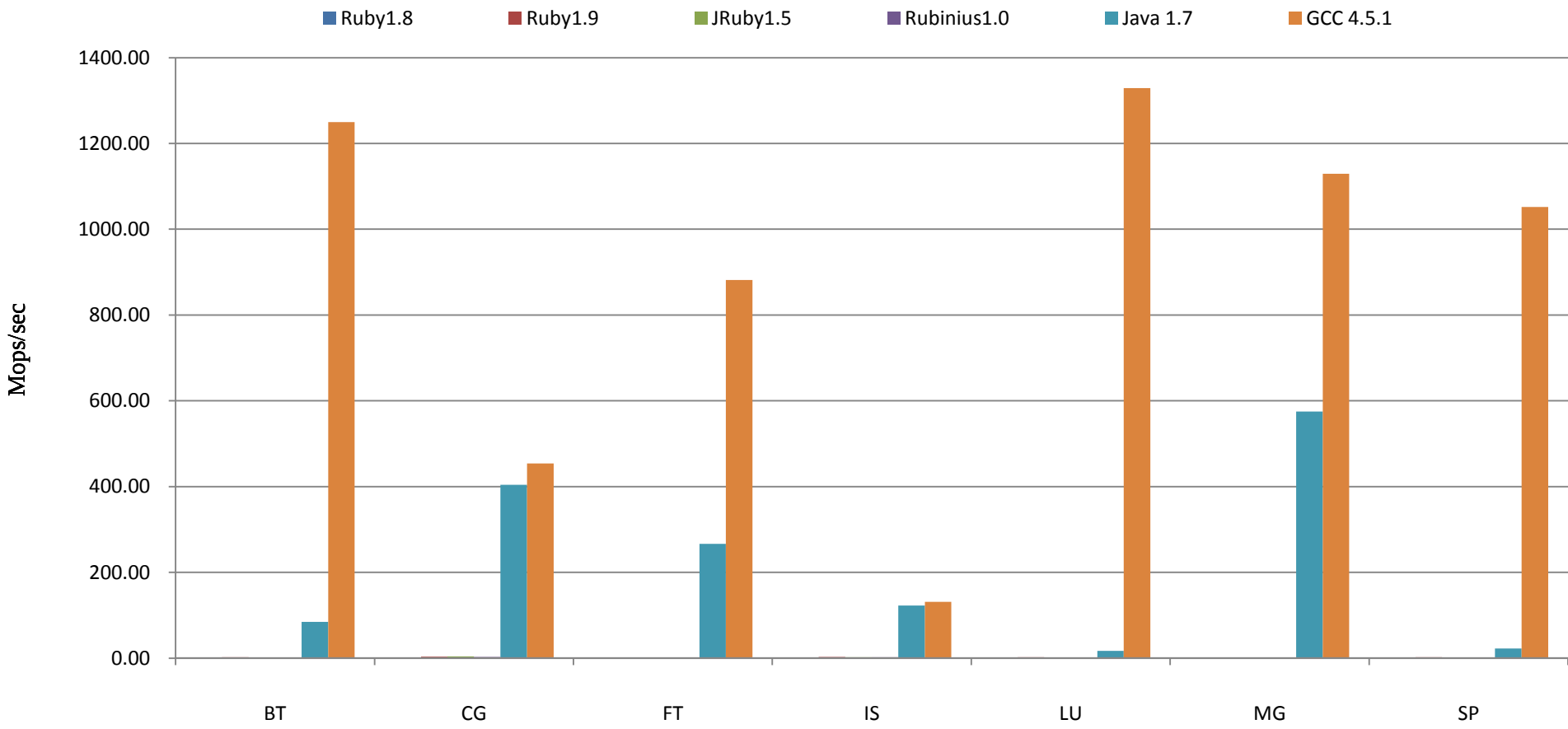
上に
- HPCに有利な特徴を持っている。

Rubyは
- 学習コストが大変低い・書きやすい

上に
- HPCに有利な特徴を持っている。
  (基本性能でC・Fortranと並べる事が前提)

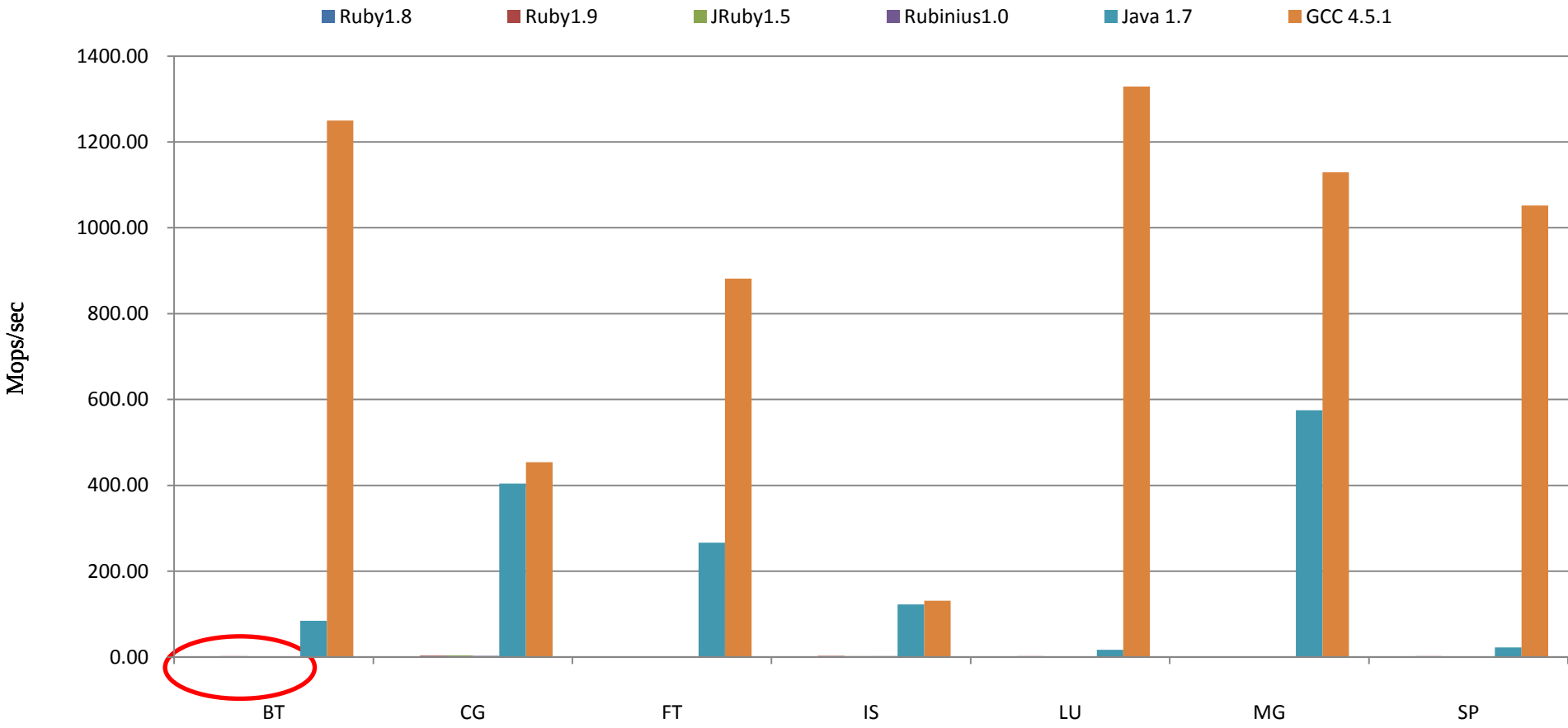
どれくらい遅いか？

NAS Parallel Benchmarks [野瀬2011], SPEC CPUとの良い相関[泊2011]



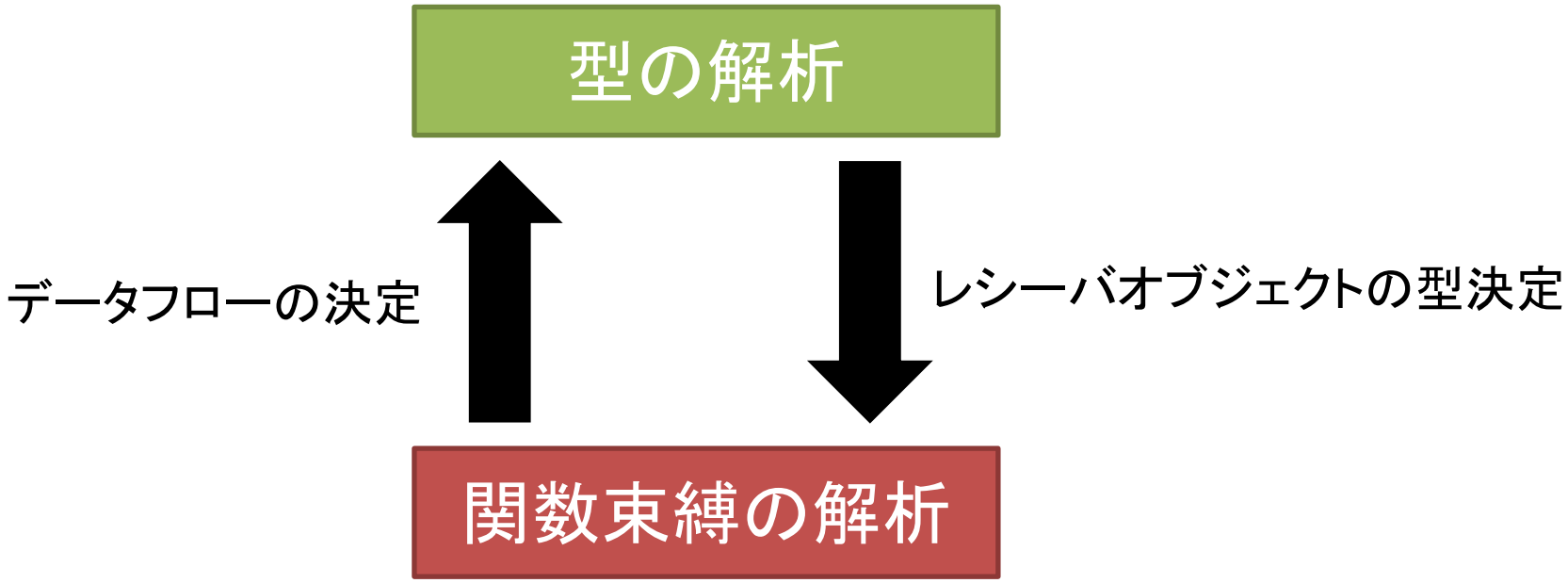
どれくらい遅いか？

NAS Parallel Benchmarks [野瀬2011], SPEC CPUとの良い相関[泊2011]



動的言語の静的解析の難しさ

- 型、関数束縛がメッセージパッシングを介して相互に依存



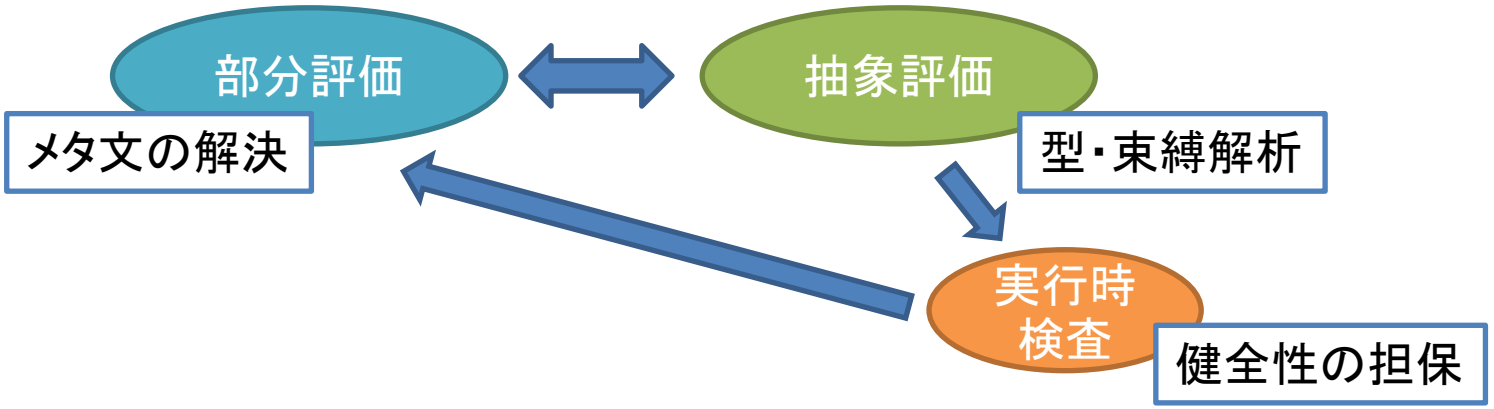
既存研究

- 型システム
  - Soft Typing [R. Cartwright 91, M. Fagan 92]
  - Type Annotation [R.A. MacLachlan 92]
  - Run-time Guard併用 [L.P. Deutsch 84, C. Chambers 89, 90]
- 実行時最適化
  - 関数検索キャッシング
  - JIT コンパイル [T. Lindholm 99, M. Paleczny 01]

- 関数束縛の情報に依存
- ランタイムの存在に依存

提案手法

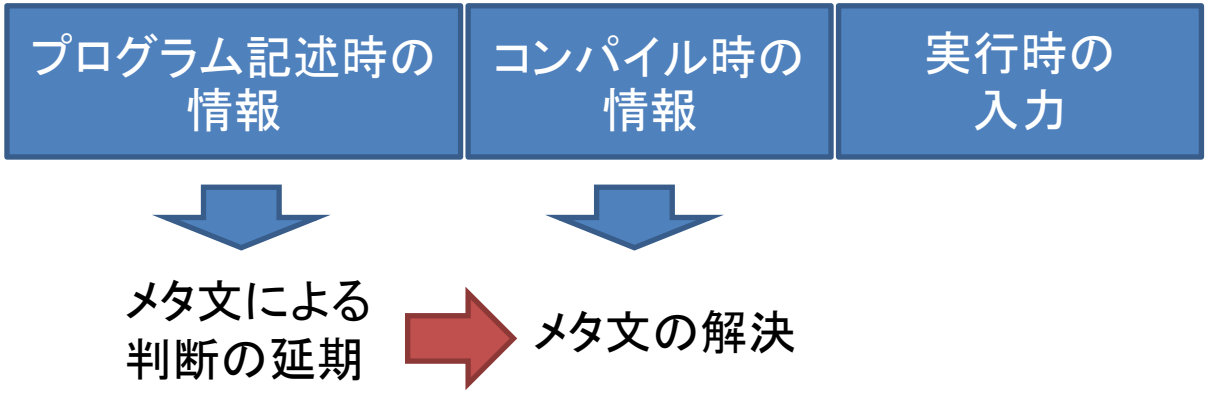
- 部分評価・抽象解釈を組み合わせた、静的型解析・関数束縛解析手法



- 対象言語の仕様に拡張・制限を課さない
  - Evalや動的なメソッド再定義などの使用を想定
- ランタイムの存在を前提としない
  - 分散並列環境・GPUなどへの応用

着眼点

- 動的言語で書かれていようと、現実のプログラムには一定の傾向が存在
  1. 型・関数の定義部、それを用いた計算部は分離している
  2. Eval等のメタ文は、コンパイル時に得られる情報で評価可能



解析の概要

- 目標
  - オペランドの型の静的解決
  - メソッド呼び出しの静的解決
- 手段
  - ステートメント単位の解析
    - 計算部では型・関数の定義が行われない前提に立つ
  - メタ文の静的評価
  - 抽象解析・部分評価を統合する束構造の導入
  - 個々の組み込み関数毎の解析器・変換器の用意

どのようなプログラムだと(現状)上手くいかないか？

- 真の実行時入力にプログラムの性質が依存するケース
  - 対策：JITコンパイル

```
while ...
    ...
    s = readline
    ...
    eval(s)
    ...
end
```

どのようなプログラムだと(現状)上手くいかないか？

- 任意の深さでネストするデータ構造を使っている
  - tree構造など
  - かなり大きな問題
- 現状：ネストの深さがコンパイル時定数なら大丈夫
- 対策：モデル解析の手法の応用を検討している

どのようなプログラムだと(現状)上手くいかないか？

- 拡張ライブラリの中身は高速化されない
- 拡張ライブラリ関数の戻り値の型は分からない
  - 対策案:
    1. 投機的な型の予測  実装中
    2. Type Annotation
    3. 解析器・変換器の作成を容易にする・・・？

例：元の関数

```
static VALUE
fix_plus(x, y)
    VALUE x, y;
{
    if (FIXNUM_P(y)) {
        long a, b, c;
        VALUE r;

        a = FIX2LONG(x);
        b = FIX2LONG(y);
        c = a + b;
        r = LONG2NUM(c);

        return r;
    }
    if (TYPE(y) == T_FLOAT) {
        return rb_float_new((double)FIX2LONG(x) + RFLOAT(y)->value);
    }
    return rb_num_coerce_bin(x, y);
}
```

例:対応する解析器

```
static VALUE
ae_fix_bin(VALUE x, VALUE y)
{
    VALUE r;
    y = to_lat(y);
    r = lat_new(LAT_SET);
    if (lat_include(rb_cFixnum, y) || lat_include(rb_cBignum, y)) {
        lat_set_add(r, rb_cFixnum);
        lat_set_add(r, rb_cBignum);
    }
    if (lat_include(rb_cFloat, y))
        lat_set_add(r, rb_cFloat);
    if (lat_le(y, lat_set_new3(rb_cFixnum, rb_cBignum, rb_cFloat)))
        return r;
    NOT_IMPLEMENTED();
}
static VALUE
ae_fix_plus(VALUE x, VALUE y)
{
    if (FIXNUM_P(x) && (FIXNUM_P(y) || TYPE(y) == T_FLOAT)) {
        return fix_plus(x, y);
    }
    return ae_fix_bin(x, y);
}
```

シリアライゼーションの問題

- XML
- YAML
- JSON
- など

```
YAML.load(STDIN.read)
```

シリアライゼーションの問題

```
module Readable
  def self.included(klass)
    def klass.read(io)
      obj = io.read.deserialize
      if obj.class != self
        raise "#{obj.class.to_s} is not what was expected: #{self.to_s}"
      end
      obj
    end
    ...
  end
end
```

アルゴリズム

$prog' \Leftarrow$ empty list
$self \Leftarrow$ initial self object
$V \Leftarrow$ initial variable table
while $prog$ is not empty **do**
  $e \Leftarrow prog.first$
  $prog \Leftarrow append(P(self, V, e), prog.next)$
  $V_{old} \Leftarrow clone(V)$
  $I'(self, V, e)$
  if e was indeterminable **then**
    $prog' \Leftarrow append(prog', \text{"guard}(e)\text{"})$
    $V \Leftarrow V_{old}$
  else
    $prog' \Leftarrow append(prog', \text{annotate}(e))$
  end if
  $prog \Leftarrow prog.next$
end while

部分評価によるメタプログラム解決

抽象解釈による解析

解析不能文への対処

対象言語のモデル:文法

(programs)	$p ::=$	$e_1 \cdots e_n$
(variables)	$x ::=$	s
		$\$s$
(expressions)	$r, e ::=$	c
		x
		$x = e$
		$e.s$
		$e_1.s = e_2$
		$r.s(e_1, \cdots, e_n)$
		$r.s(e_1, \cdots, e_n)\ b$
		$e_1; e_2$
		$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$
		$\texttt{while}\ e_1\ \texttt{do}\ e_2$
		$\texttt{create\_object}(e)$
		$\texttt{create\_class}()$
		$\texttt{define\_function}(e, s, b)$
		$\texttt{eval}(e)$
(blocks)	$b ::=$	$\{\|x_1, \cdots, x_n\|\ e\}$

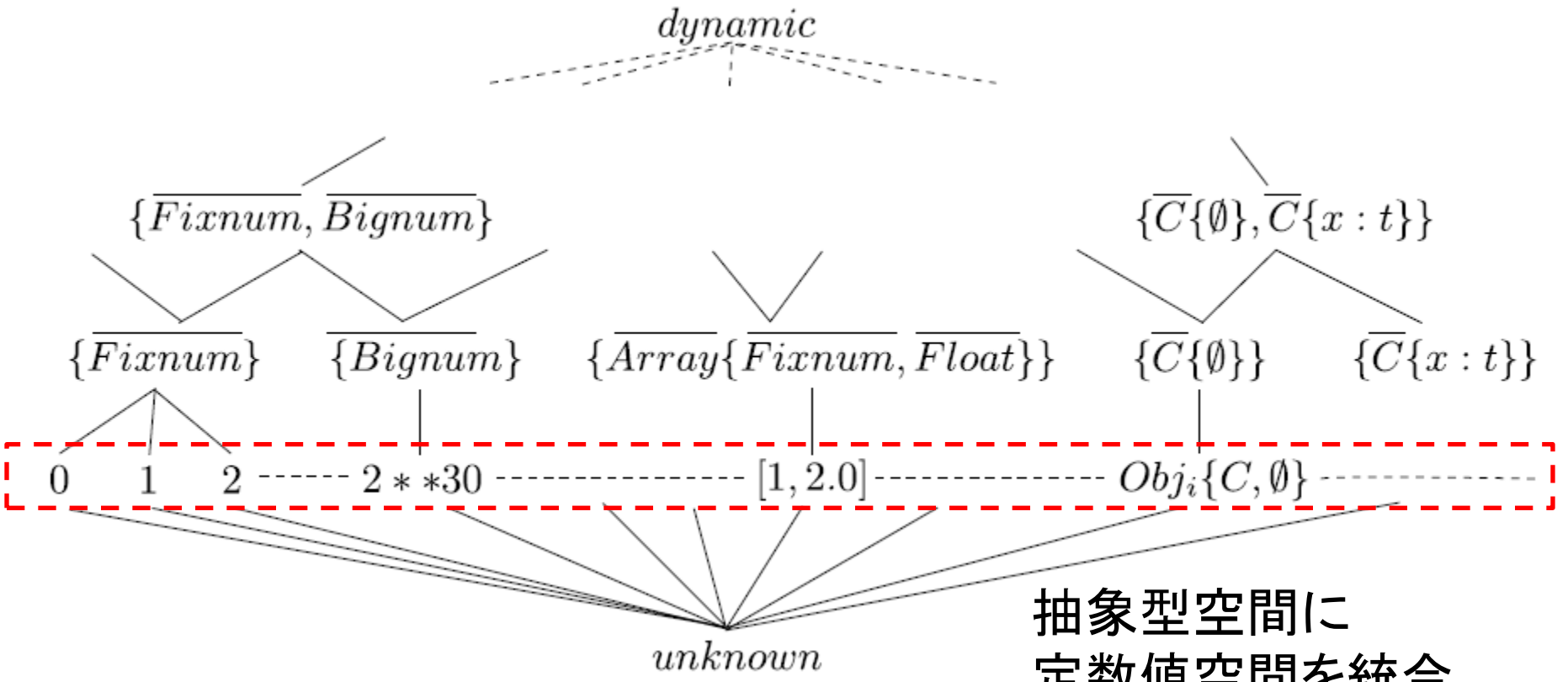
メッセージパッシング

対象言語のモデル:値空間

$$
\begin{array}{lll}
\text{(values)} & v ::= & \text{fixed length integers} \\
 & & \text{multi-precision integers} \\
 & & \text{floating point numbers} \\
 & & \text{strings} \\
 & & \cdots \\
 & & Cls_i\{F\} \\
 & & Obj_i\{v, V\} \\
\text{(functions)} & f ::= & prim \\
 & & b \\
\text{(variable tables)} & V ::= & s_1 : v_1, \cdots, s_n : v_n \\
\text{(function tables)} & F ::= & s_1 : f_1, \cdots, s_n : f_n
\end{array}
$$

- 型オブジェクトが他の値と同一の空間に属する
- 関数テーブルを内部に持つ

解析に用いる束構造



抽象型空間に
定数値空間を統合
⇒部分評価を可能に

抽象解釈

$$
\begin{array}{lcl}
I'(\_, V, c) & = & c \\
I'(\_, V, x) & = & V[x] \\
I'(\_, V, \$x) & = & V[\$x] \\
I'(self, V, x = e) & = & \text{let } v = I'(e) \text{ in } V[x] \leftarrow v;\ v \\
I'(self, V, e.s) & = & \text{case } I'(e) \text{ of} \\
 & & \quad |\{\overline{Cls}_{i_1}\{M_1\}, \cdots\} \Rightarrow M_1[s] \vee \cdots \\
 & & \quad |\text{otherwise} \Rightarrow \text{give up analysis} \\
I'(self, V, e_1.s = e_2) & = & \text{case } I'(e_1) \text{ of} \\
 & & \quad |\{\overline{Cls}_{i_1}\{M_1\}, \cdots\} \\
 & & \qquad \Rightarrow M_1[s] \leftarrow v; \cdots; v \\
 & & \quad |\text{otherwise} \Rightarrow \text{give up analysis} \\
I'(self, V, r.s(e_1, \cdots, e_n)) & = & \text{case } I'(r) \text{ of} \\
 & & \quad |c \Rightarrow \\
 & & \qquad \text{let } v_1 = I'(e_1) \text{ in} \\
 & & \qquad \cdots \\
 & & \qquad \text{let } v_n = I'(e_n) \text{ in} \\
 & & \qquad \text{call}(c, V, \texttt{lookup}(c, s), v_1, \cdots, v_n) \\
 & & \quad |\{t_1, \cdots, t_m\} \Rightarrow \\
 & & \qquad \text{let } v_1 = I'(e_1) \text{ in} \\
 & & \qquad \cdots \\
 & & \qquad \text{let } v_n = I'(e_n) \text{ in} \\
 & & \qquad \text{let } t = \vee_{t_i} \text{call}(t_i, V_i, \\
 & & \qquad\qquad \texttt{lookup}(t_i, s), v_1, \cdots, v_n) \text{ in} \\
 & & \qquad V \leftarrow V_1 \vee \cdots \vee V_m; t \\
 & & \quad |\text{otherwise} \Rightarrow \text{give up analysis} \\
 & & (V_i \text{ is a clone of } V)
\end{array}
$$

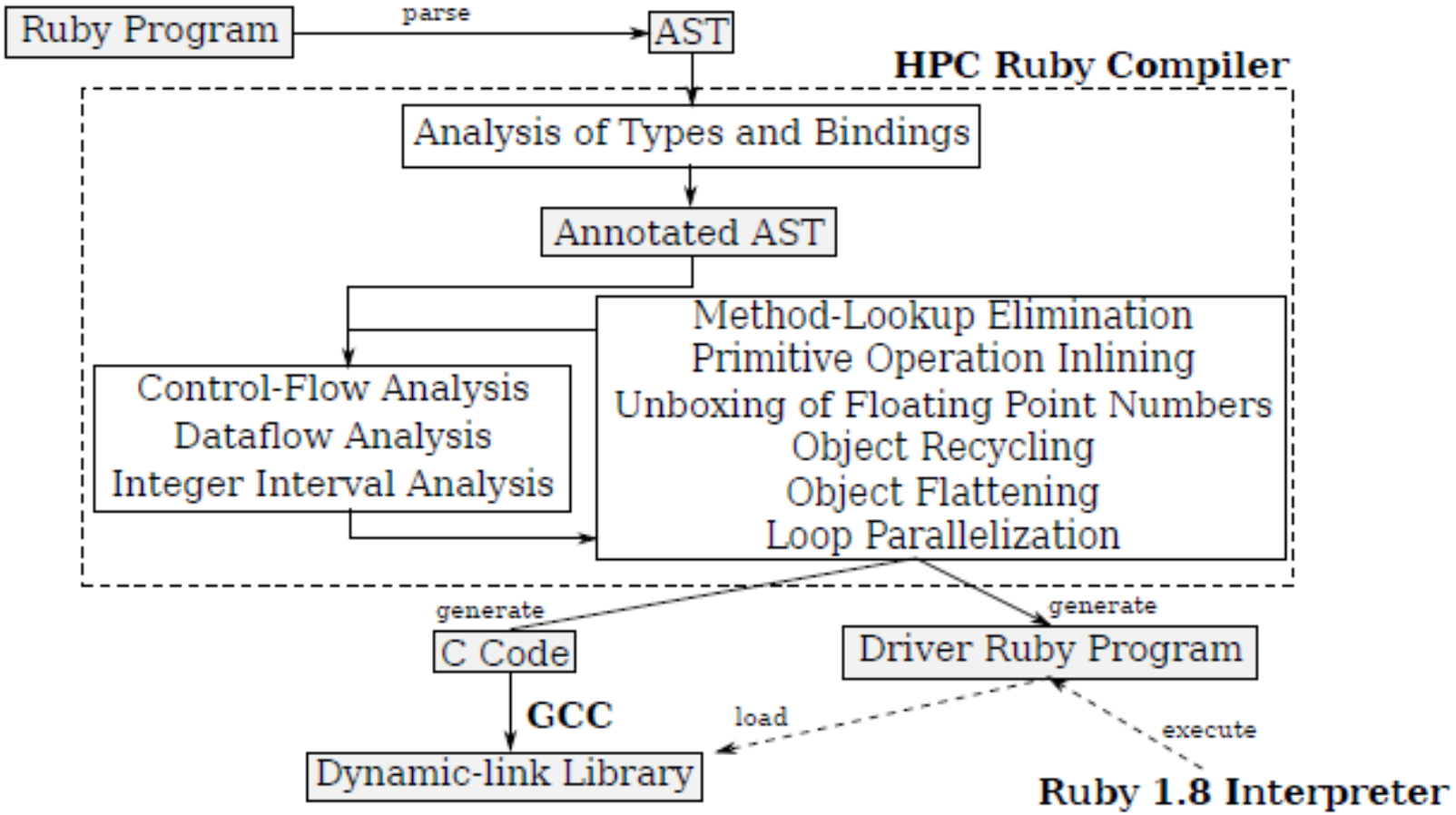
抽象解釈

$$
\begin{array}{lcl}
I'(self, V, \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) & = & I'(e_1); \\
 & & \text{let } t = I'(self, V_1, e_2) \vee I'(self, V_2, e_3) \\
 & & \text{in } V \leftarrow V_1 \vee V_2; t \\
 & & (V_i \text{ is a clone of } V) \\
I'(self, V, \texttt{while } e_1 \texttt{ do } e_2) & = & \texttt{nil} \vee (I'(e_1); I'(e_2) \vee I'(\texttt{while } e_1 \texttt{ do } e_2)) \\
I'(self, V, \texttt{create\_object}(e)) & = & i(I'(e)) \\
I'(self, V, \texttt{create\_class}()) & = & Cls_i\{\emptyset\} \\
I'(self, V, \texttt{define\_function}(e, s, b)) & = & \text{give up analysis} \\
 & & \quad \text{(if it is not placed in top-level scope)} \\
I'(self, V, \texttt{define\_function}(e, s, b)) & = & \text{case } I'(e) \text{ of} \\
 & & \quad |Cls_i\{F\} \Rightarrow F[s] \leftarrow b; \texttt{nil} \\
 & & \quad |\text{otherwise} \Rightarrow \text{give up analysis} \\
I'(self, V, \texttt{eval}(e)) & = & \text{give up analysis} \\
 & & \\
\texttt{lookup}(c, s) & = & \text{let } \overline{Cls_i}\{F\} = i(c) \text{ in } F[s] \\
\texttt{lookup}(\overline{Cls_i}\{F\}, s) & = & F[s] \\
\texttt{lookup}(dynamic, \_) & = & \text{give up analysis} \\
\texttt{call}(self, V, prim, v_1, \cdots, v_n) & = & prim'(self, v_1, \cdots, v_n) \\
\texttt{call}(self, V, b, v_1, \cdots, v_n) & = & I'(self, V \cup x_1 : v_1 \cup \cdots \cup x_n : v_n, e) \\
 & & (b = \{|x_1, \cdots, x_n|e\})
\end{array}
$$

部分評価

$$
\begin{array}{lcl}
P(self, V, x = e) & = & \text{"}x = P(e)\text{"} \\
P(self, V, e.s) & = & \text{"}P(e).s\text{"} \\
P(self, V, e_1.s = e_2) & = & \text{"}e_1.s = P(e_2)\text{"} \quad (I'(e_1) \text{ is a constant}) \\
P(self, V, e_1.s = e_2) & = & \text{"}P(e_1).s = e_2\text{"} \quad (I'(e_1) \text{ is non constant}) \\
P(self, V, r.s(e_1, \cdots, e_n)) & = & \mathtt{expand}(P(r), F[s], e_1, \cdots, e_n) \\
 & & \quad (\text{if } \{\overline{Cls_i}\{F, \_\}\} = I'(r)) \\
P(self, V, \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3) & = & \text{"}e_1; e_2\text{"} \quad (\text{if } I'(e_1) = true) \\
P(self, V, \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3) & = & \text{"}e_1; e_3\text{"} \quad (\text{if } I'(e_1) = false) \\
P(self, V, \mathtt{while}\ e_1\ \mathtt{do}\ e_2) & = & \text{"}P(e_1); e_2; \mathtt{while}\ e_1\ \mathtt{do}\ e_2\text{"} \\
 & & \quad (\text{if } I'(e_1) = true) \\
P(self, V, \mathtt{while}\ e_1\ \mathtt{do}\ e_2) & = & \text{"}\mathtt{nil}\text{"} \quad (\text{if } I'(e_1) = false) \\
P(self, V, \mathtt{eval}(e)) & = & c \quad (\text{if } c = I'(e) \text{ is a constant string}) \\
 & & \\
\mathtt{expand}(self, prim, e_1, \cdots, e_n) & = & \text{"}prim(self, e_1, \cdots, e_n)\text{"} \\
\mathtt{expand}(self, b, e_1, \cdots, e_n) & = & \text{"}y_1 = e_1; \cdots; y_n = e_n; e[x_i \rightarrow y_i]\text{"} \\
 & & \quad (b = \{|x_1, \cdots, x_n|e\})
\end{array}
$$

HPC Rubyコンパイラ



HPC Rubyコンパイラ

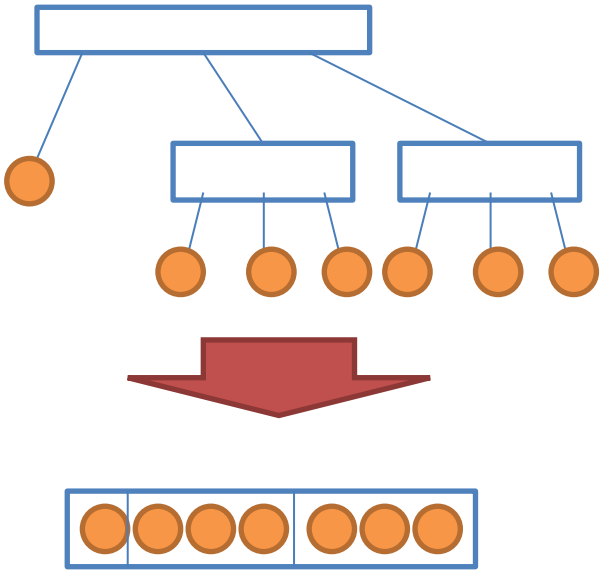
```
% rubyc foo.rb [OPTION]
% ls
Makefile extconf.rb foo.rb  foo_opt.c foo_opt.rb
foo_opt.so
% ruby foo_opt.rb
```

実装した最適化技術

- Method Lookup Elimination
- Interval Analysis
- Primitive Operation Inlining
- Unboxing of Floating Point Numbers
- Object Recycling
- Object Flattening
- Auto Parallelization of Iteration

Object Flattening

- ネストした構造から1次元構造へ
  実行時のエイリアシング検査を利用



```
if aliasing_test(body) then
   body' ← flatten(body)
   computation using body'
   recover(body, body')
else
   original version of computation
end if
```

イテレータ記述からの自動並列化

- Rubyのイテレータによる記述からの並列

```
def internal_epot(bodies)
   bodies.map{|b| b.external_epot(bodies)}.inject(&:+)/2.0
end
```

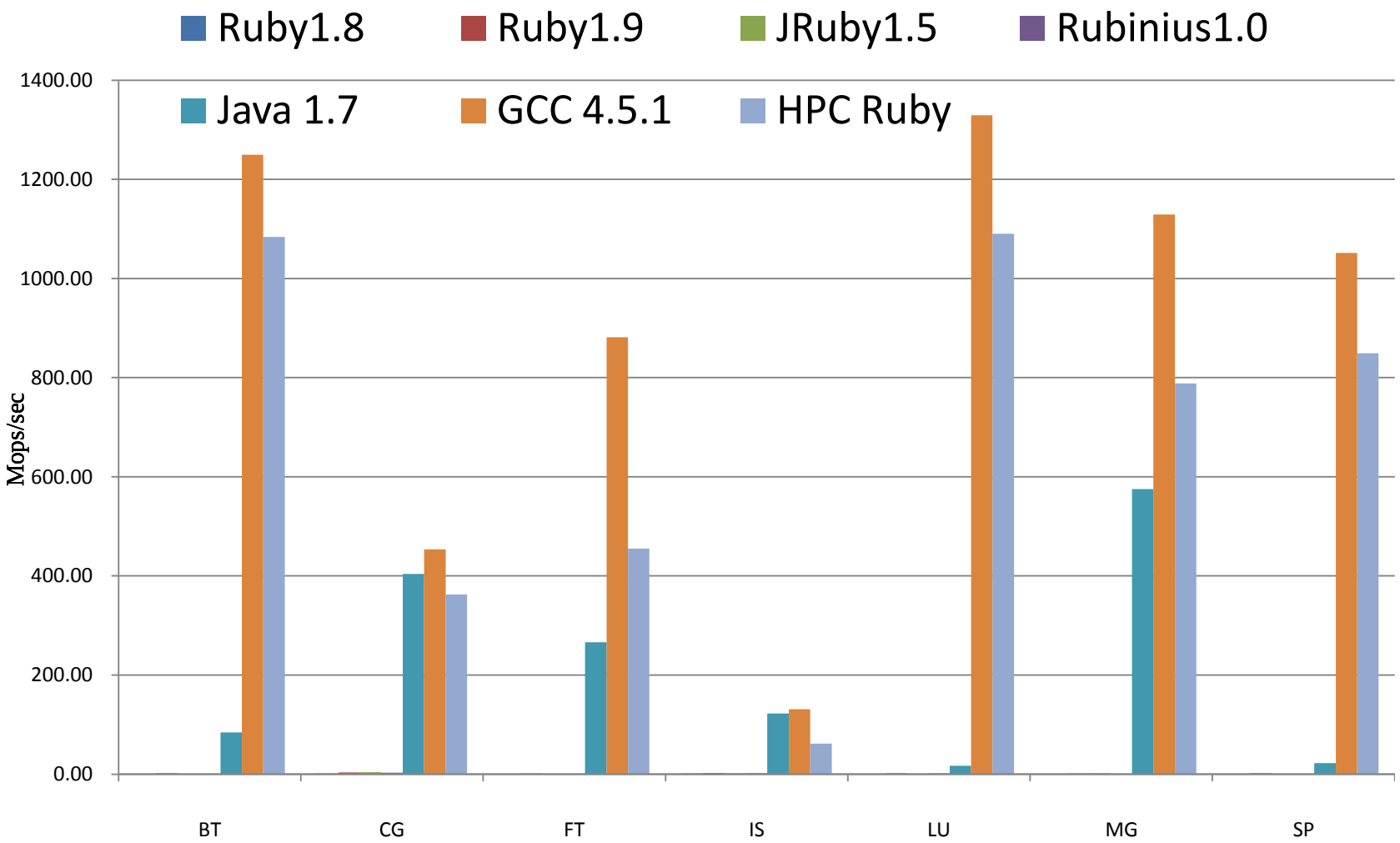
要素別演算　　縮約演算

```
...
#pragma omp parallel for reduction(+:t)
for (i = 0; i < n; i++) {
   t = t + external_epot(b, bodies[i])
}
return t/2.0
```

評価

- 動的機能を用いず記述されたプログラムで評価
- NAS Parallel Benchmarks 3.0
  - Java → Rubyトランスレータを使用 [野瀬]
  - 変換により型情報は損なわれている
- 熱拡散方程式の陽解法
  - 手動で新規に実装
- 環境
  - Core 2 Duo, 2.40GHz
- シングルスレッド実行での評価

評価:NAS Parallel Benchmarks

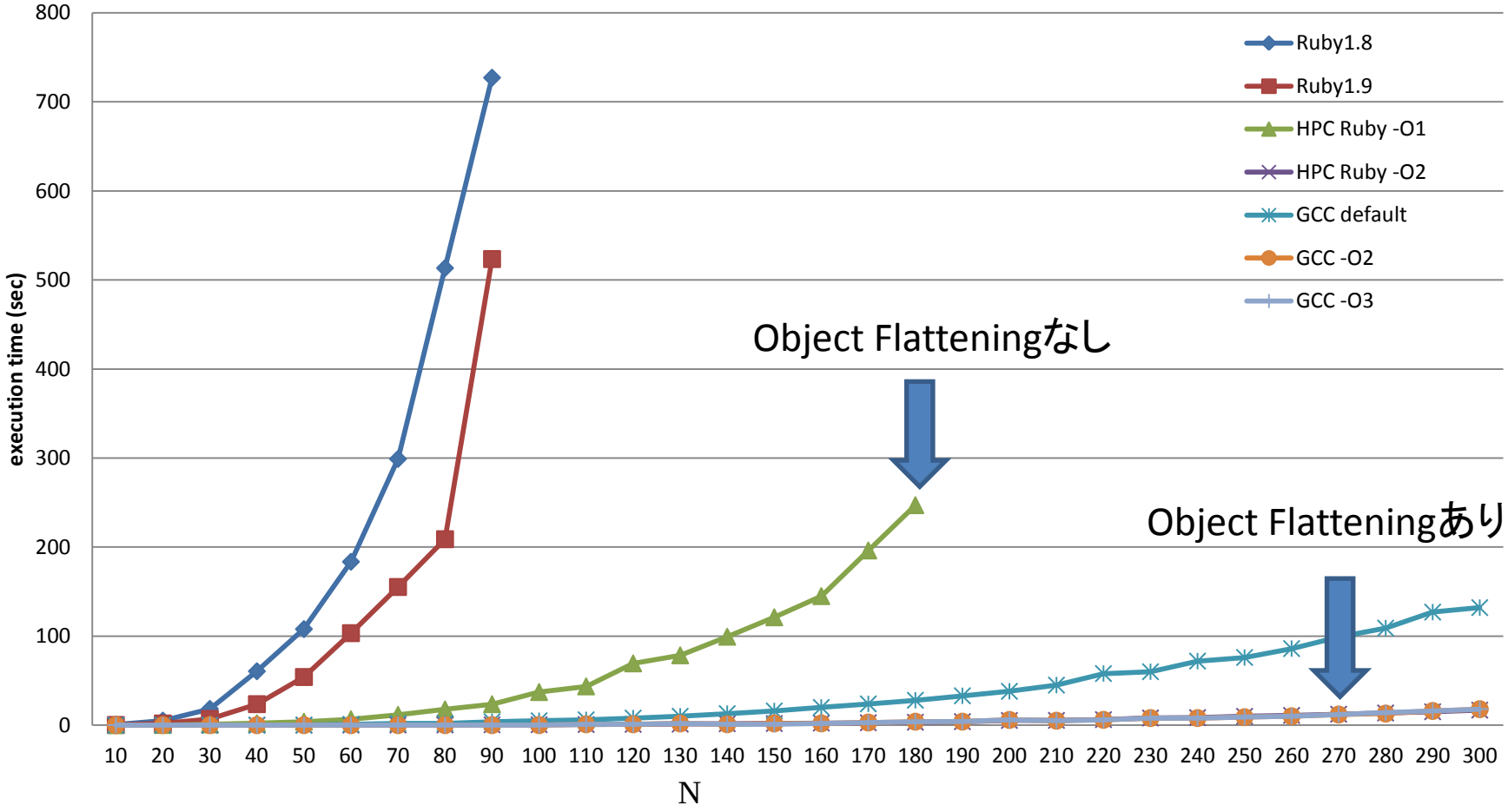


評価:NAS Parallel Benchmarks

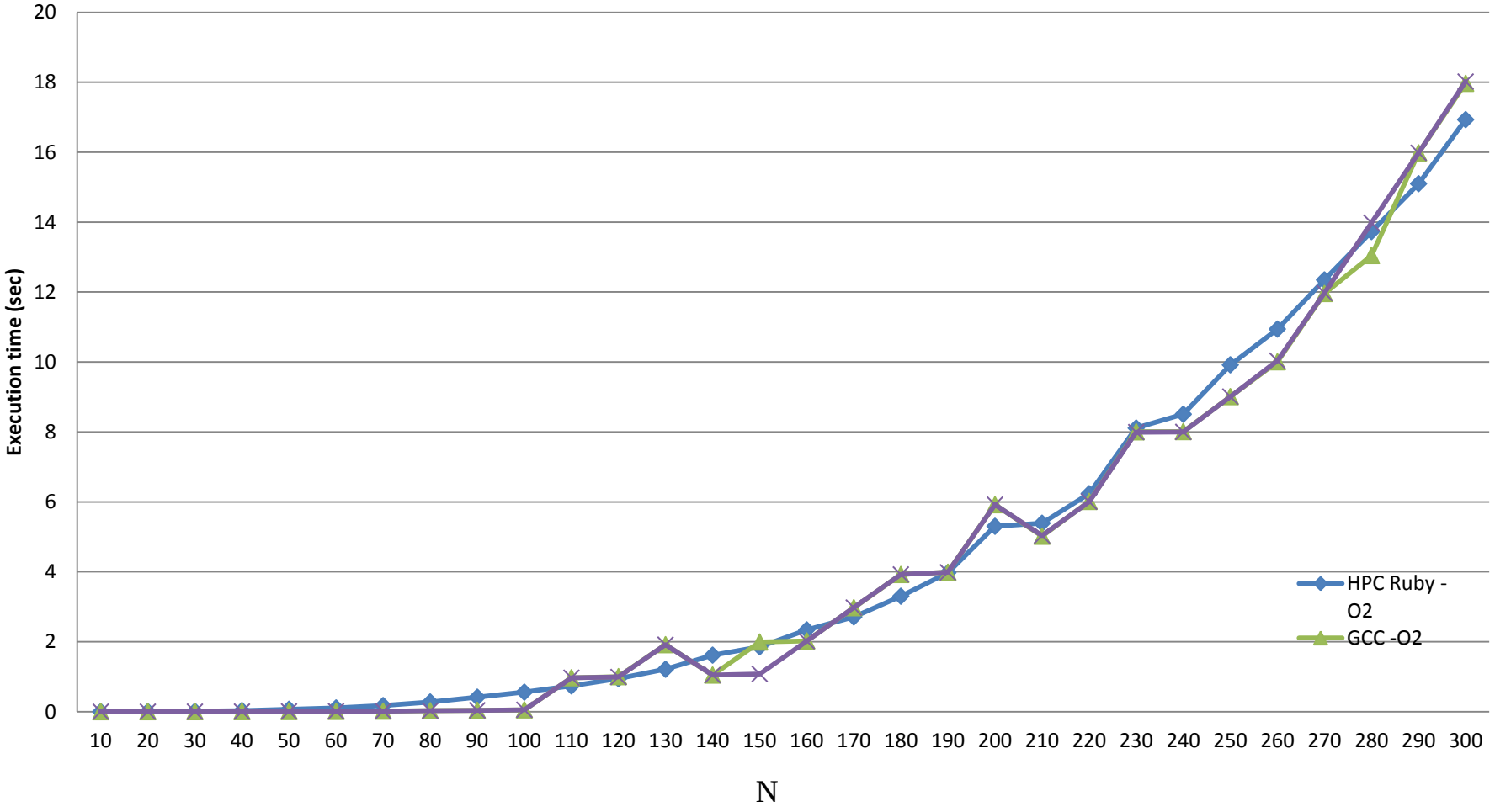
- Ruby1.8に比べ最大1889倍、平均700倍
- Ruby1.9に比べ最大556倍、平均255倍
- GCCに比べ最大86.7%の性能、平均67.7%

- 出力コード中に実行時検査の使用は無し
  - 静的な記述であれば、動的言語で記述されていようが解析が可能である

評価：熱拡散方程式


execution time (sec)
800
700
600
500
400
300
200
100
0
10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200 210 220 230 240 250 260 270 280 290 300
N
Ruby1.8
Ruby1.9
HPC Ruby -O1
HPC Ruby -O2
GCC default
GCC -O2
GCC -O3
Object Flatteningなし
Object Flatteningあり

評価：熱拡散方程式


Execution time (sec)
0
2
4
6
8
10
12
14
16
18
20
10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190 200 210 220 230 240 250 260 270 280 290 300
N
HPC Ruby -O2
GCC -O2

評価：熱拡散方程式

- Object Flatteningを併用する事により GCCと遜色ない性能を発揮
- 静的言語との性能差を解消する為には メモリ構造に手を入れる事が必要
  - 動的最適化に対する優位性を示す

結論

- 動的言語の為のプログラム解析手法を提案
  - 言語仕様に拡張・制限を加えず実現
- HPC Rubyコンパイラを開発した
  - 科学計算を対象として、十分な性能を達成

- HPC向けの高度な技術の開発に必要な基盤技術を揃えた

今後の研究

- 並列処理(マルチスレッド)
- 超並列処理(SIMD, 分散並列)
  - 解析・最適化技術開発
  - コンパイラ開発
  - 科学計算ソフトウェア開発
- 動的言語を用いる事により実現・実証