Rubyプログラムを高速に実行するための処理系の開発

東京農工大学大学院
笹田　耕一

2005 1/11　第46回プログラミング・シンポジウム

Rubyプログラムを高速に実行するための処理系の開発

日本 Ruby の会
ささだ　こういち

Agenda

- 背景
- 目標
- Ruby の特徴
- 設計と実装
- 最適化手法
- 現時点での性能評価
- まとめ

Smithsonian National Museum of Natural History

背景

- オブジェクト指向スクリプト言語Ruby
  - http://www.ruby-lang.org
  - 使いやすいと評判
  - 世界中で広く利用
    - Ruby-talk ML では一日に百通以上
  - 日本発（主開発者：まつもとゆきひろ氏）
  - コミュニティ活動も活発
    - 日本Rubyの会
    - Rubyist Magazine

背景(cont.)

- 既存のRuby処理系は遅い
  - 構文木を辿って実行(eval関数の再帰呼び出し)
  - 性能上の理由から泣く泣く Ruby 利用を断念
  - 誤った常識の流布

    「Rubyって遅いんでしょ？　使えないよ」
    - あなたが思うほど遅くありません
    - Ruby はとても使える道具

背景(cont.)

- Ruby処理系の処理速度向上が必須
  - バイトコード（仮想マシン型）処理系が適当
  - Lisp, Java, .NET, Smalltalk 処理系など
- いくつかのRubyバイトコード処理系→不十分
  - まつもと氏 平成12年度未踏ソフトウェア創造事業
    - 不完全（命令が不十分・コンパイラも無い）
  - ByteCodeRuby (George Marrows氏)
    - 現状のRuby処理系よりも遅い→ プロジェクト終了
  - その他、ほとんど作りかけ

Rubyプログラムを高速に実行するための処理系の開発

- VM命令セットの設計
- コンパイラの設計・開発
- 仮想機械(RubyVM)の設計・開発
- JIT (Just In Time), AOT (Ahead of Time) compiler

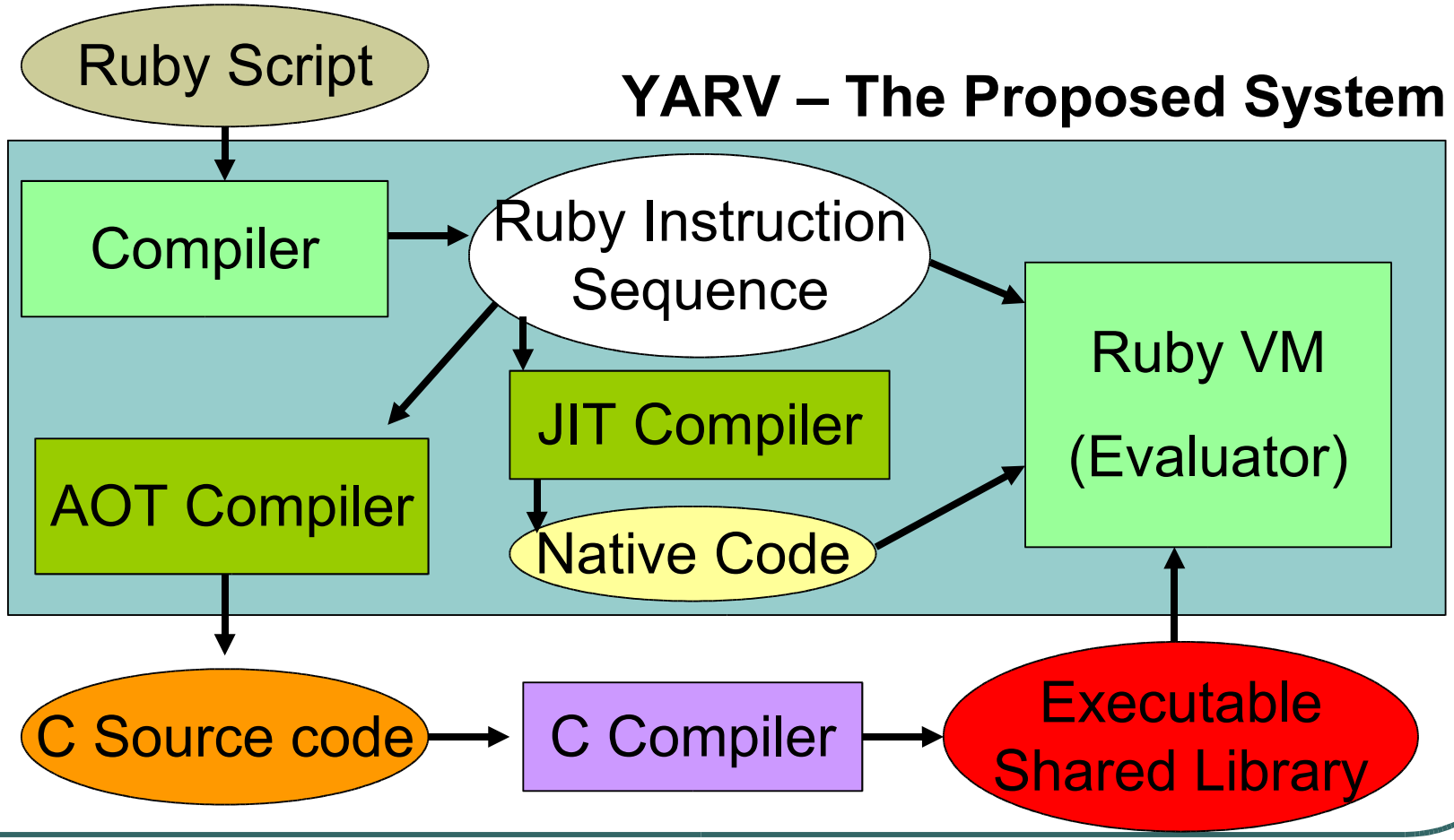
YARV: Yet Another Ruby VM
として開発中（**Open Source Software**）

- IPA 2004年度「未踏ユース」

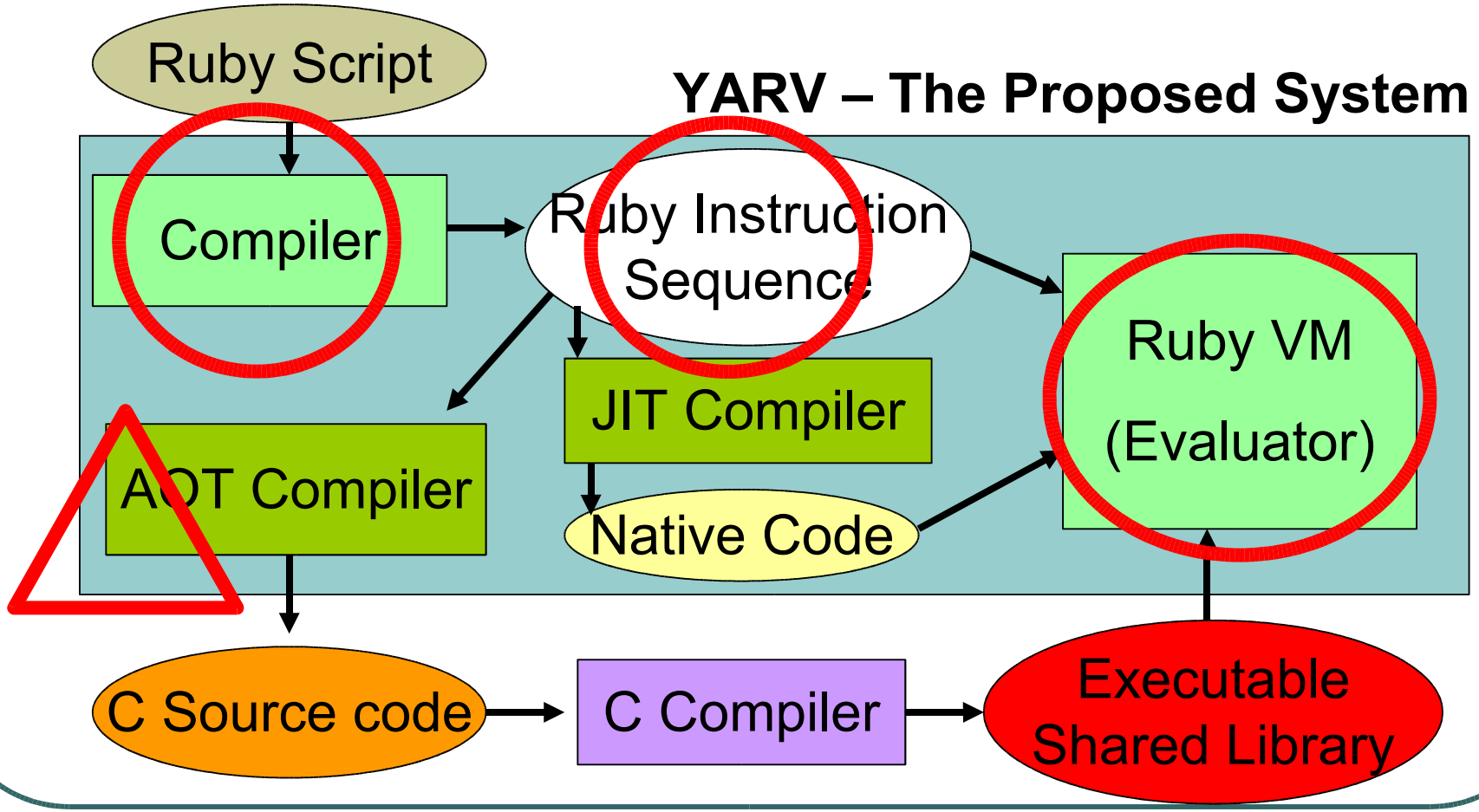
目標

- 世界一速いRuby処理系
  - 基本性能で2倍、JITコンパイラでは5倍
    AOTコンパイラでは10倍の性能向上を目指す
- 世界中の人に使ってもらえるように
  - **RubyVM** としての十分な機能を実装
- 次期Rubyの公式実装 Rite に
  - Ruby2.0 処理系 **Rite** （現在 1.9.0 開発中）
  - 本プロジェクトが成功すれば YARV を **Rite** として採用

システムの全体像


Ruby Script
YARV – The Proposed System
Compiler
Ruby Instruction Sequence
Ruby VM (Evaluator)
JIT Compiler
AOT Compiler
Native Code
C Source code
C Compiler
Executable Shared Library

システムの全体像


Ruby Script
YARV – The Proposed System
Compiler
Ruby Instruction Sequence
JIT Compiler
AOT Compiler
Native Code
Ruby VM (Evaluator)
C Source code
C Compiler
Executable Shared Library

Rubyの特徴

- クラスベースのオブジェクト指向
- ダイナミック
  - すべてがオブジェクト・再定義可能
  - **Evil Eval**
- 例外処理など大域ジャンプ可能
- スレッドのサポート
- Cによる拡張ライブラリ開発が容易

Rubyの特徴(cont.)

- クロージャ
- ブロック付きメソッド呼び出し
  - メソッド呼び出し時，容易に closure を渡すことが可能

```
# (1) in Ruby
[1,2,3].each{|e|
  print e
}
```

```
;; (2) in scheme
(for-each (lambda (e)
            (print e))
          '(1 2 3))
```

YARV 概要

- Simple Stack Machine
- 拡張ライブラリとして実装
- 既存のRuby と連携して機能を利用
  - ガーベージコレクタ
  - Ruby Script パーサ
  - Ruby C API が使える
    - i.e) Memory Management
      - using Array without “free()” is very happy
- 大部分，C言語で実装

YARV命令セット

- Ruby プログラムを表現できる命令セットを定義
- スタック制御，フロー制御，…

```
# Ruby program
print(“Hello”)
```

```
# YARV Insns
putself
putstring “Hello”
send :print, 1
```

命令記述フォーマット

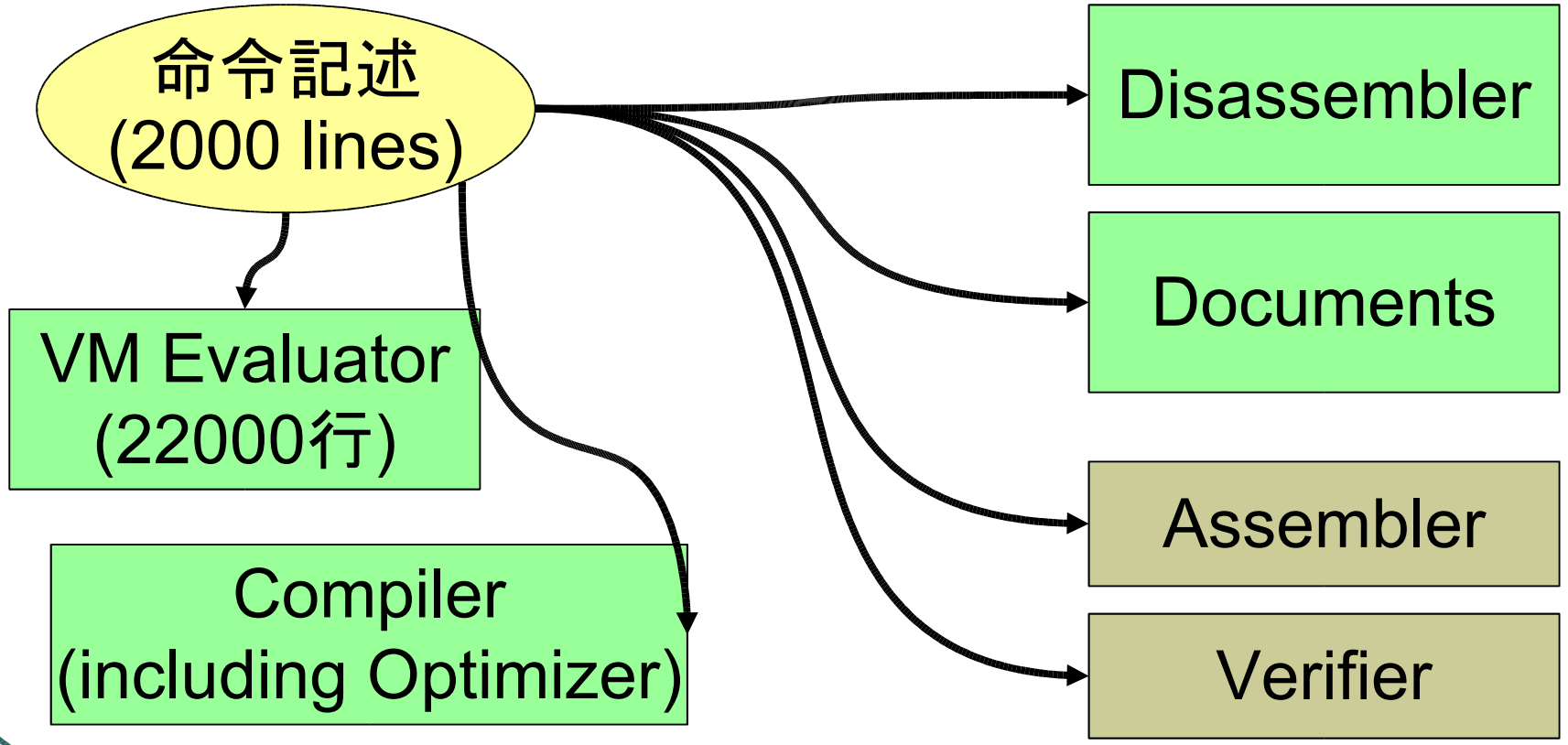
- 命令記述フォーマットを定義
  - 具体的な記述部分は C
  - 次を変数として宣言
    - オペランド
    - スタックからポップする値とプッシュする値
  - VM 実装が非常に楽に
    - これを利用して，いろいろと自動生成
    - デバッグプリント挿入が楽
    - 最適化手法の適用
    - ドキュメント生成が楽（コード埋め込みドキュメント）

命令記述フォーマット(cont.)

```
DEFINE_INSN
putobject           # 命令名
(VALUE obj)         # オペランド
()                  # スタックから取る
(VALUE val)         # スタックへ置く
{
  val = obj;    /* C 言語で実装を記述 */
}
```

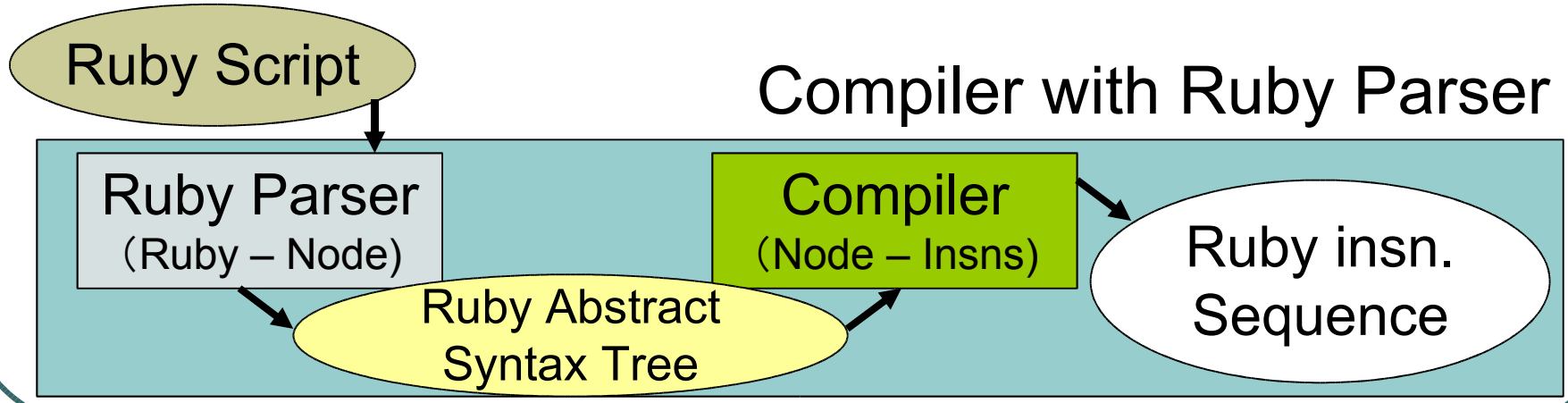
命令記述によるVM生成

- 命令記述から生成されるもののまとめ



コンパイラ概要

- Ruby 構文木を YARV 命令列に変換
  - Ruby Script Parser は構文木を作成
  - 構文木をたどり，命令列に変換
  - 最適化のためのコード変換を適宜行う

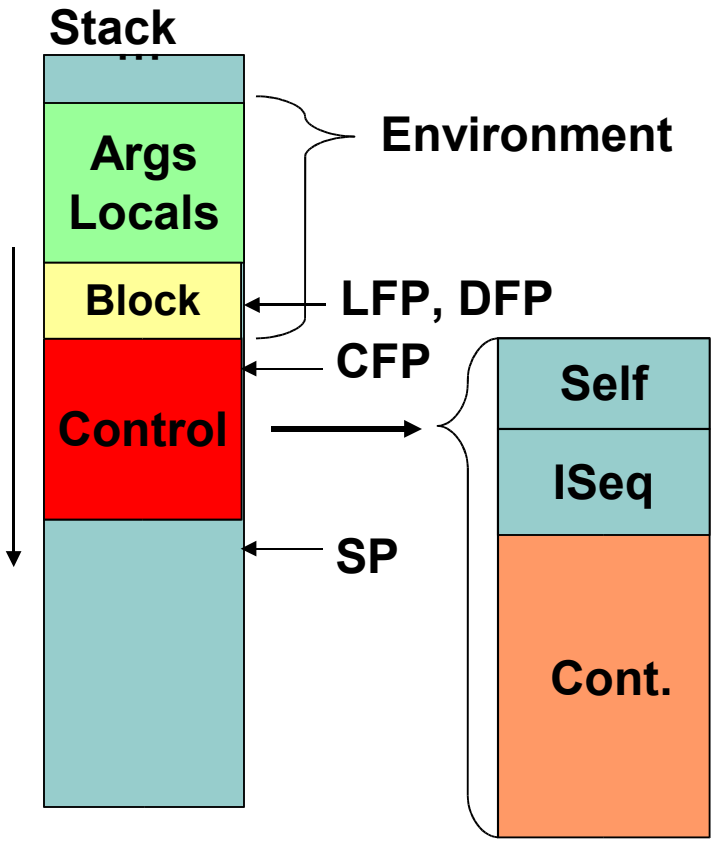


VM概要:レジスタ

- 5 registers
  - PC: Program Counter
  - SP: Stack Pointer
  - LFP: Local Frame Pointer
  - DFP: Dynamic Frame Pointer
  - CFP: Control Frame Pointer

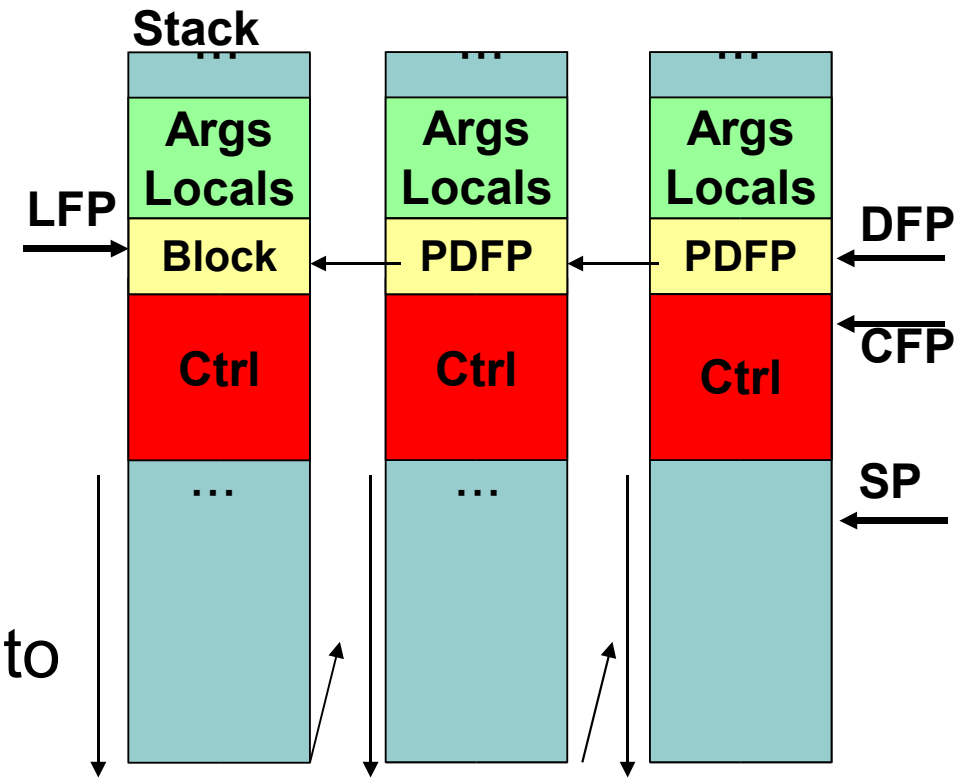
VM概要 – Stack Frames

- Method Frame
  - 他のVMとほぼ同様
  - クラスフレームもほぼ同様
- Control frame
  - 各フレームが持つ
  - レシーバ
  - 命令列情報
  - 継続情報（caller regs）
  - CFP によりアクセス可



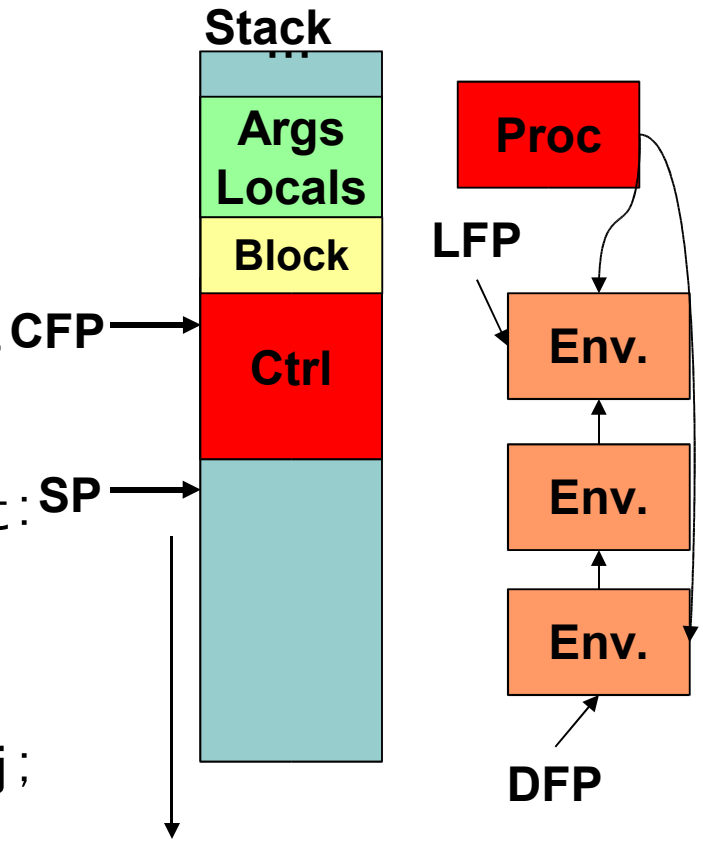
VM概要 – Stack Frames (cont.)

- Block Frame
  - ‘yield’ によって作成
  - LFP points to method local environment
  - DFP point to current environment
  - DFP[0] == PDFP point to previous environment



VM概要 – Proc Object

- Proc オブジェクトの生成
  - 要するにクロージャ
    - 環境をヒープに保存
  - LFP, DFP はヒープ上を指す
    - スタック上の値は遡って張り直す




```
# Proc sample
def m arg; x = 0
  iter{|a| i=1
    iter{|b| j=2
      Proc.new
    }}; end
```

```
Struct ProcObject:
    VALUE self;
    VALUE *lfp;
    VALUE *dfp
    VALUE iseqobj;
```

VM概要 – 例外処理

- 例外処理用テーブルを利用
  - JavaVM などと同様
- 例外処理部分（例外で保護された部分）突入
  - 特になにもしない（実行コスト0）
  - 現在のRuby処理系では毎回 setjmp
- 例外が発生したら
  - スタックフレーム巻き戻し
  - 例外処理用テーブルを参照して処理
- 他の大域ジャンプもこの機構を利用

YARVでの最適化

- インラインキャッシュ
  - 命令オペランドにデータを埋め込み，キャッシュ
  - 定数アクセスは遅い
    - A::B::C は4命令必要
    - 各定数アクセスも結構遅い
  - メソッド検索
    - 現在のRuby処理系はグローバルメソッドキャッシュを利用
  - Global “VM state counter”

YARVでの最適化(cont.)

- スタックキャッシング
  - スタックトップの2つをキャッシュ
  - 5状態
  - 命令数は5倍（各状態ごとに命令を用意）
    - putself_xx_ax
    - putself_ax_ab
    - putself_bx_ba
    - putself_ab_ba
    - putself_ba_ab

YARVでの最適化(cont.)

- 命令融合(未実装)
  - 頻出する連続した命令列を1命令に
- オペランド融合
  - putobject true → puttrue
- 融合するものを指定すれば，自動的に生成
  - 命令記述を解析すれば可能
  - コンパイラ(変換器)も自動生成
- プロファイラ

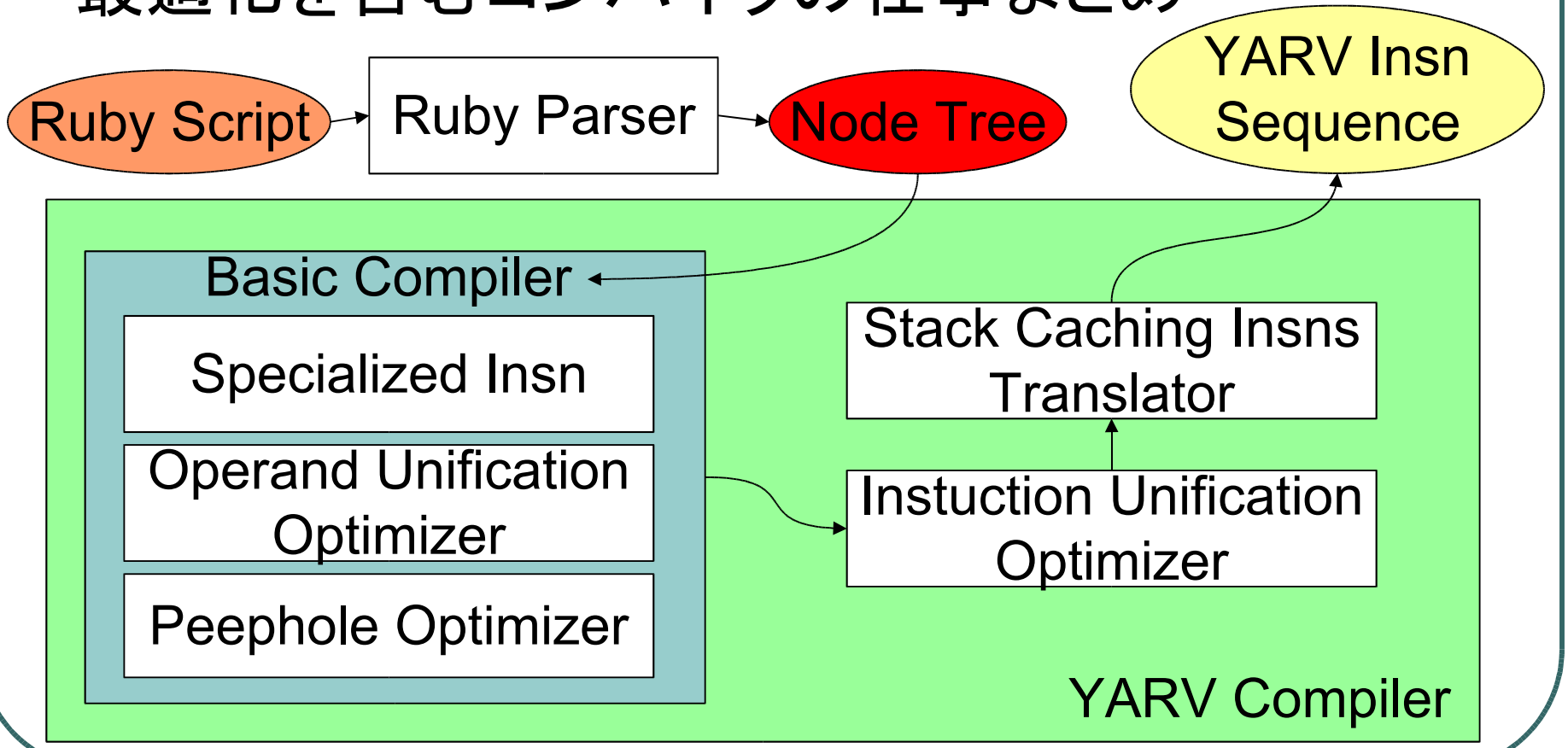
YARVでの最適化(cont.)

- 特化命令
  - 単純な命令を特殊化
  - a + b → opt_plus

```
if(a と b は整数か？)
  if(整数の + メソッドは再定義されていないか？)
    return a+b;
通常のメソッド呼び出し a.+(b) を実行
```

YARVでの最適化(cont.)

- 最適化を含むコンパイラの仕事まとめ



YARVでの最適化(cont.)

- JIT compile
  - 未実装
  - 簡単な方法を探してます
    - JIT コンパイル用ライブラリの利用
    - コンパイル結果のバイナリから切り貼り
- AOT compile
  - Ruby → YARV 命令列 → C Source code → Ruby 拡張ライブラリ
  - YARV 命令列から C 言語変換は比較的容易
    - 命令記述を利用

ベンチマーク結果

- 評価環境
  - CPU: Intel Celeron 1.7GHz, Mem: 512MB
  - OS: Windows2000 + Cygwin 1.52
  - コンパイラ: gcc (GCC) 3.3.3
    - コンパイラオプション: -O2 -f-no-crossjumping
  - 比較対象の Ruby: 1.9.0 (2004-11-30) [i386-cygwin]
- 利用した最適化
  - Direct threading code
  - 特化命令（整数演算向け）
  - オペランド融合

ベンチマーク結果(cont.)

プログラム	Ruby(sec)	YARV(sec)	Ratio
whileloop	7.66	0.69	11.10
times	6.47	3.12	2.07
const	1.90	0.22	8.48
Method	7.25	0.74	9.83
block	13.76	1.60	8.58
Rescue	3.10	0.02	131.87
Rescue2	5.32	3.17	1.69

ベンチマーク結果(cont.)

プログラム	Ruby(sec)	YARV(sec)	Ratio
fib	7.73	0.79	9.82
tak	25.07	3.42	7.33
tarai	20.26	4.13	4.91
matrix	6.37	2.62	2.43
sieve	3.56	0.73	4.88
ackermann	3.06	0.27	11.55
count_words	0.63	0.61	1.03

ベンチマーク結果(cont.)

- スタックキャッシングの効果

プログラム	YARV(sec)	YARV SC(sec)	Ratio
whileloop	0.69	0.40	1.72
fib	0.79	0.56	1.41
sieve	0.73	0.61	1.19

今後の課題

- デバッグ
- Ruby の仕様の完全準拠（まだまだ不十分）
- 最適化器を実装
  - 特に JIT/AOT Compiler
- ベンチマークプログラムの充実
- その他の仕組み
  - コンパイル済み YARV 命令列のダンプ（pre-compile）
- YARV 上で他の動的言語の実装
  - Scheme

YARV 開発について

- 0.1.0 released
- Home page
  - http://www.atdot.net/yarv/
  - インストール方法などもここに
- Mailing list
  - Yarv-dev (in Japanese)
  - Yarv-devel (in English)

おわり

- Special Thanks
  - Yarv-dev / Yarv-devel subscribers
    - NaCl　まつもとゆきひろさん
    - 筑波大　前田敦司助教授
    - 産総研　首藤一幸さん
    - And others
  - Ruby Hackers
  - IPA

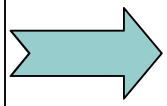
YARVでの最適化(cont.)

- インラインキャッシュ(cont.)
  - Global “VM state counter”
  - VM state counter は再定義時インクリメント
    - メソッド再定義
    - 定数再定義
  - インラインキャッシュ時，このカウンタ値を一緒に格納
  - キャッシュ参照時，現在のカウンタ値と格納されたカウンタ値を比較
    - 同じならキャッシュヒット
    - 違ったらキャッシュミス

VM概要 – Ensure

- 例外処理用テーブルを利用
- Ensure 節部分をコピー

```
# sample
begin
  A
ensure
  B
end
```



```
Compiled:
  Start_A:
    A
  End_A:
    B
  End_B:
  end
```

```
ExceptionTable:
entry:
  type: ensure
  range: Start_A - End_A
  do: B
  restart point: End_B
  restart sp: 0
```

-f-no-crossjumping

- 同じようなコードを共有しようとする
  → ジャンプ命令が増える
  → threaded code 効果半減

```
switch(cond){                    switch(cond){
  case A:                          case A:
    P1; P2; break;     ⟶            P1; L: P2; break;
  case B:                          case B:
    P3; P2; break;                   P3; goto L; break;
}                                }
```

ベンチマーク結果(cont.)

- AOT Compiler の効果

```
i=0
while i<100000000 # 100M
  i+=1
end
```

Ruby	YARV SC	YARV AOT
81.2	6.8 (x11.9)	0.7 (x116.0)

ベンチマーク結果（コンパイル時間）

```
if false
def m
  a = 1
  b = 2
  c = 3
end
# 以下 15万行繰り返し
end
ruby  1.546000   0.078000   1.624000 (  1.648000)
yarv  6.718000   0.156000   6.874000 (  6.913000)
```

ベンチマーク結果（コンパイル時間）

```
if false
  a = 1
  b = 2
  c = 3
  ... 繰り返し  16万行
End
```

```
ruby  1.281000   0.031000   1.312000 (  1.318000)
yarv 96.188000   0.094000  96.282000 ( 96.896000)
```