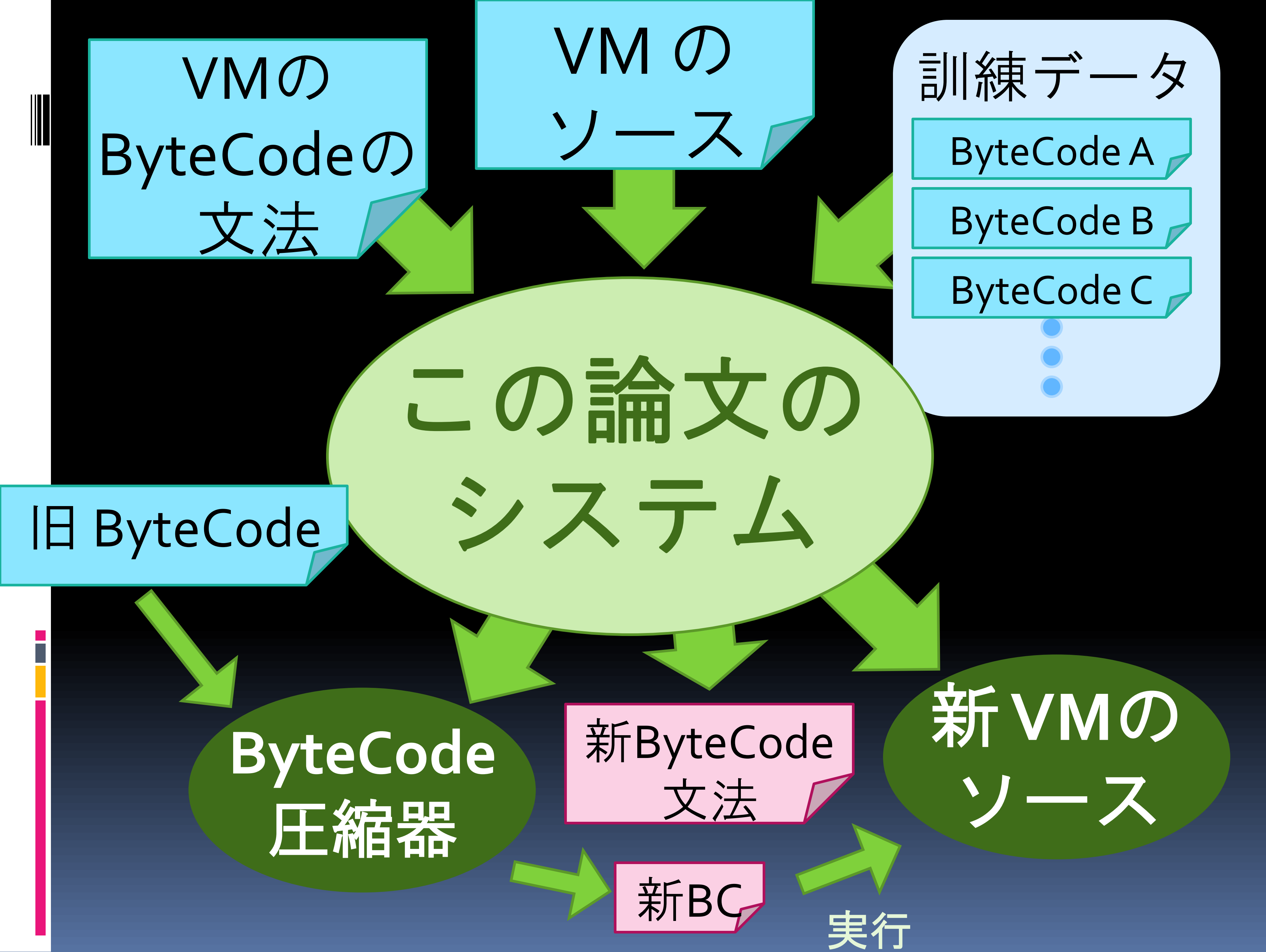
- This slide was
  - a material for the “Reading PLDI Papers (PLDIr)” study group
  - written by Kazuhiro Inaba (www.kmonos.net), under my own understanding of the papers published at PLDI
    - So, it may include many mistakes etc
- For your correct understanding, please consult the original paper and/or the authors’ presentation slide!

k.inaba (稲葉 一浩), reading the following paper:

PLDIr #5
Jan 6, 2010

paper written by W. S. Evans and C. W. Fraser

Bytecode Compression via Profiled Grammar Rewriting



VMのByteCodeの文法
VM のソース
訓練データ
ByteCode A
ByteCode B
ByteCode C
この論文のシステム
旧 ByteCode
ByteCode圧縮器
新ByteCode文法
新VMのソース
新BC
実行

要するに

- 与えられた命令セットを変換して
- 与えられた訓練プログラム（に似たプログラム）を短く表現できるような
- 新・命令セット　と　それ用VM
- を自動生成するシステム

入力１：VMのByteCodeの文法

- こんなの（例はスタックマシン）
  - スタックマシンなことは本質ではなさそう

```
<start> =
<start> = <start> <x>

<v> =           <v0>
<v> = <v>       <v1>
<v> = <v> <v>   <v2>

<x> =           <x0>
<x> = <v>       <x1>
<x> = <v> <v>   <x2>
```

```
<x0> = JUMPV <byte> <byte>
<x0> = LocalCALLV <byte> <byte>
<x0> = RETV
```

```
<x1> = ARGB | ARGD | ARGF | ARGU
<x1> = BrTrue <byte> <byte> | CALLV
<x1> = POPD | POPF | POPU
<x1> = RETD | RETF | RETU
```

```
<x2> = ASGNB | ASGNC | ASGNS | ASGNU
<x2> = ASGND | ASGNF
```

```
<v2> = ADDD | DIVD |        MULD | SUBD
<v2> = ADDF | DIVF |        MULF | SUBF
<v2> =        DIVI | MODI | MULI
<v2> = ADDU | DIVU | MODU | MULU | SUBU
<v2> = BANDU | BORU | BXORU
<v2> = EQD | GED | GTD | LED | LTD | NED
<v2> = EQF | GEF | GTF | LEF | LTF | NEF
<v2> =       GEI | GTI | LEI | LTI
<v2> = EQU | GEU | GTU | LEU | LTU | NEU
<v2> = LSHI | LSHU | RSHI | RSHU
```

```
<v1> = BCOMU
<v1> = CALLD | CALLF | CALLU
<v1> = CVDF | CVDI | CVFD | CVFI
<v1> = CVID | CVIF
<v1> = CVI1I4 | CVI2I4 | CVU1U4 | CVU2U4
<v1> = INDIRC | INDIRS | INDIRU
<v1> = INDIRD | INDIRF
<v1> = NEGD | NEGF | NEGI
```

```
<v0> = ADDRFP <byte> <byte>
<v0> = ADDRGP <byte> <byte>
<v0> = ADDRLP <byte> <byte>
<v0> = LocalCALLD <byte> <byte>
<v0> = LocalCALLF <byte> <byte>
<v0> = LocalCALLU <byte> <byte>
<v0> = LIT1 <byte>
<v0> = LIT2 <byte> <byte>
<v0> = LIT3 <byte> <byte> <byte>
<v0> = LIT4 <byte> <byte> <byte> <byte>
```

```
<byte> = 0 | 1 | ... | 255
```

入力２： VM のソース

- こういう２つの関数で実装してね

```
void interpret1(
  unsigned char op,
  istate *istate
) {
  switch (op) { … }
}
```

OpCode
受け取って
一命令実行

```
void interp(istate *istate) {
  while (1)
    interpret1(
      istate->code[istate->pc++],
      istate
    );
}
```

メイン
ループ

出力１： 新VMのソース

```
void interp(istate *istate) {
  while (1)
    interpNT(istate, NT_start);
}
```

新メイン ループ

- interpret1 はそのまま
- interpNT は、新命令をデコードして最終的に interpret1 を呼び出す

出力２： ByteCode圧縮器

- 圧縮器　　　＝　パーザ
- 新ByteCode　＝　構文木

旧ByteCode:　ZERO ONE ADD

0: <START> = <X>
0: <X> = <X> <X> ADD
1: <X> = ZERO
2: <X> = ONE

圧縮器

(START0 (X0 (X1 ZERO)
(X2 ONE)
ADD))

新ByteCode: ０ ０ １ ２

出力３：新ByteCode文法

- 「構文木が小さくなる」のが圧縮率が高い文法
- よくあらわれるパターンを一つの規則にまとめる
  - 例えば４つの加算が頻出なら

```
0: <START> = <X>
0: <X> = <X> <X> ADD
1: <X> =  ZERO
2: <X> = ONE
3: <X> = <X> <X> ADD <X> <X> ADD ADD
```

出力３：新文法

- よく使うパターンを新しい文法規則に

(X0 (X0 ? ? ADD) (X0 ? ? ADD) ADD) = 00??0??

圧縮→ (X3 ? ? ? ?) = 3????

- 注意！やみくもに増やすと、規則を表す番号が大きくなりすぎて逆に効率が悪い
  - 丁度 0 ～ 255 で表せるくらいの範囲で増やす
  - それより減らすとビットデコードが必要で(速度)効率が悪化する、らしい
- JUMP先をまたいでまとめるとJUMPしにくい
  - 圧縮は基本ブロックごとにしかやらない

実験結果

input	original	compressed			
		trained on gcc		trained on lcc	
		bytes	ratio	bytes	ratio
gcc	1,423,370	471,111	33%	577,814	41%
lcc	199,497	75,077	38%	57,722	29%
gzip	47,066	19,466	41%	19,706	42%
8q	436	138	32%	152	35%

k.inaba (稲葉 一浩), reading the following paper:

paper written by R. Shaham, E. K. Kolodner and M. Sagiv

Heap Profiling for Space-Efficient Java

オブジェクトが不要になるのは…

- オブジェクトが
  - 最後に使用された時

ココ ← を短くしたい

drag time

  - Unreachableになった時
  - GCで解放された時

どうやって？

1. Drag time 測定用に改造したVMを実装
   - 全オブジェクトについて最終使用時刻を記憶
   - GCを頻繁に回してUnreachable時刻をほぼ特定
2. プロファイル情報が得られる
   - 各 new 式ごとに、作ったオブジェクトが
     - どのくらい長く drag されてるか
     - どの式が last use になっているか
3. 手動で最適化！

実験結果 ＆ 自動化に向けて

Benchmark Program	Reduced In-Use Integral ($M\ Byte^2$)	Reduced Reachable Integral ($M\ Byte^2$)	Original In-Use Integral ($M\ Byte^2$)	Original Reachable Integral ($M\ Byte^2$)	Drag Saving Ratio (%)	Space Saving Ratio (%)
javac	566.49	937.09	656.19	1015.4	21.8	7.71
jack	50.58	82.24	57.07	141.93	70.34	42.06
raytrace	127.47	220.59	128.42	317.62	51.28	30.55
jess	74.01	231.91	73.67	260.86	15.47	11.1
euler	1421	1459.64	1424.34	1574.28	76.46	7.28
mc	10969.61	11010.44	11310.73	11747.09	168.82	6.27
juru	159.83	210.92	159.83	236.86	33.68	10.95
analyzer	196.19	409.84	195.9	482.46	25.34	15.05

- 「こういうAnalysisがあれば（今回は手動でやった）最適化を自動化できる」という議論が最後に１ページ程度並んでいる