

# Java バイトコード 上での実行時プログラム特化

増原英彦

東京大学大学院 総合文化研究科 広域システム科学系

masuhara@graco.c.u-tokyo.ac.jp

米澤明憲

東京大学大学院 理学系研究科 情報科学専攻

yonezawa@is.s.u-tokyo.ac.jp

## 概要

実行時プログラム特化は、機械語レベルのプログラム生成によって高速な特化を行う技術であり、プログラムの実行中に得られる情報を利用した特化を可能にする。しかし、生成されるプログラムの最適化の度合いが低いことや、複数の言語・機械への対応が困難であるという問題がある。本論文は、スタック仮想機械語上でプログラム生成を行い、それをさらに実機械語へ変換するバイトコード特化を提案する。この方法では、バイトコードから実機械語への変換時に最適化を行うことで、比較的小さなオーバーヘッドで効率的なプログラムを生成することができる。さらに、ソースプログラムを用いずに仮想機械語のプログラムを直接解析することで、複数の言語への対応を容易にしている。現在までに、Java 仮想機械のサブセットに対する処理系が作成され、従来の実行時特化よりも効率的なプログラムが生成できることが確められている。

## 1 はじめに

プログラムの特化 (program specialization) は、プログラムに部分的な入力を与えて、その入力に最適化されたプログラムを生成する技法である。これによって、汎用的で可読性の高いプログラムを特定の用途に最適化して実行できる。特にプログラムをソース言語レベルで特化するものは部分計算 (partial evaluation) と呼ばれ、多くの研究がなされている [6, 8]。

近年、与えられたプログラムから、予め機械語レベルの特化を行うプログラム (特化器) を作成してしておき、高速に特化を行う実行時プログラム特化 (run-time specialization<sup>1</sup>) 技術が注目されている [2-5, 9, 10, 12, 13]。高速化によって、プログラムの実行時に特化を行っても全体の効率向上が望めるため、幅広い応用が期待されている。コンポーネントウェアのように、部品化された機能を実行時に合成して使用するスタイルのプログラムの最適化など、実行時特化への期待は大きい。

一方、実行時特化では機械語プログラムを直接操作するため、コンパイラや実行機械に強く依存した技術になってしまっている。また、特化されたプログラムの効率が低下しているという問題もある。そこで本論文では、従来の実行時特化と同様の高速な特化をしつつ、より効率的なプログラムを生成するバイトコード特化 (bytecode specialization) を提案する。この技法では、機械語上で特化を行うかわりに、一度仮想機械語上で特化を行い、それを最適化された機械語命令列へ変換することでより効率的なプログラムを生成する。また、コンパイルされた仮想機械命令列を直接解析して特化器を作成することで、ソース言語に依存しない技術となっている。

仮想機械としては Java 仮想機械 (JVM) [11] を選択した。その理由は、(1) スタック機械であるために命令列を合成することが容易であることと、(2) 局所的なメモリ (フレーム変数) 操作と大域的なメモリ (オブジェクト) 操作が分離されているなどの静的解析を容易にする性質を備えている、といった点である。さらに、既存の Just-In-Time (JIT) コンパイラを利用した効率的な機械命令列の生成が期待できたり、Java 以外の言語から JVM へのコンパイラも数多く開発されている ([7] など) ため、複数の言語への対応が容易になる、という利点もある。

仮想機械命令列の解析には、スタック/フレームの状態や副作用を正しく扱う必要があるが、基本的な制約を Stata と Abadi による型体系 [14] を拡張したもので与え、制御フロー解析によって副作用に関する制約を加えるような束縛時解析の手法を与えることで解決している。現時点までに、JVM 言語のサブセットを扱う処理系が作成されている。

<sup>1</sup> “Run-time code generation” あるいは “dynamic code generation” とも呼ばれる。

以下、2節では、既存の実行時特化技術の概略とその問題点を紹介する。続いて、提案するバイトコード特化について、3節で全体像を、4節で対象言語を示し、5節と6節で中心となる技法である束縛時解析と特化器の作成手法をそれぞれ説明する。7節では処理系の現状と簡単な性能測定について報告する。

## 2 従来の実行時特化とその問題点

バイトコード特化の説明に先立って、プログラム特化の一般的な手法を説明し、さらに、実行時特化の技術を紹介する。最後に、実行時特化の持つ問題点を紹介する。

### 2.1 プログラム特化

プログラム特化の技法は、束縛時解析 (binding-time analysis) と特化 (specialization) の2つの段階から成る。束縛時解析では、与えられたプログラムを、特化時に計算ができる (静的/static という) 式<sup>2</sup>と、特化されたプログラム中に残る (動的/dynamic という) 式に分解する。静的な式とは、定数、静的な引数、静的な式の値のみに依存している式である。動的な式は、それ以外、つまり動的な引数や動的な式に依存した式である。

例えば、以下の冪乗を計算する Java のメソッド:

```
class Power
{
  static int power(int x, int n)
  {
    if (n==0) return 1;
    else return x*power(x,n-1); } }

```

を、 $n$  が静的、 $x$  が動的であるとして束縛時解析を行うと、式  $n==0$ 、条件分岐、式  $n-1$ 、`power` の再帰呼出、`return` 文が静的であり、`then` 節の値 `1`、式 `x*power(...)` (の乗算部分) が動的であると分解される。

特化は静的な式だけを実行し、その際に実行されるべき動的な式を「特化された式」として出力する。関数呼出は、被呼出側が出力した式を呼出側が出力する式の中に埋め込んだものを出力する。例えば、`power` を  $n=3$  であるとして特化すると、 $n==0$ 、条件分岐、 $n-1$ 、`power` の再帰呼出などを実行しつつ、動的な式 “ $x * \square$ ” や “`1`” を出力してゆき (四角は再帰呼出によって作られる特化された式が埋め込まれる領域を示す)、“ $x*(x*(x*1))$ ” という式を得る。

### 2.2 部分計算と実行時特化

部分計算と実行時特化はいずれも上述のような特化を行う技術であるが、ソースプログラムを機械語にコンパイルするタイミングが大きく異なる。図 1 (a) に部分計算の、(b) に実行時特化の処理順序を示す。注目するのは、静的な入力 (static input) が与えられてから実行可能な特化されたプログラム (楕円で描かれた specialized program) を得るまでの手間である。ここではそれをオンラインな処理と呼ぶことにし、図中では灰色の背景によって示す。

部分計算 (図 1 (a)) は、束縛時解析されたプログラム (annotated source) と、静的な入力を与えられると、静的な式を解釈実行しつつ、特化されたプログラムをソース言語レベルで生成する。これをさらにコンパイルすることによって特化されたプログラムが実行可能になる。この場合、オンラインに、解釈実行とコンパイルという、低速の処理が含まれている。

一方、実行時特化 (図 1 (b)) では、束縛時解析されたプログラム中の動的な式 (dynamic expression) を「用意された命令列を生成する式」に置きかえた、特化器 (specializer) をソース言語レベルで作成し、コンパイルする。また、動的な式はそれぞれコンパイルされ、機械語命令列として用意しておく。特化時に静的な入力を与えられると、特化器は、用意した命令列をメモリ上に並べることで特化されたプログラムを生成する。この場合のオンラインの処理は、(1) 静的な式は解釈実行でなく、コンパイルされたものが実行され、(2) 動的な式をコンパイルせずに特化されたプログラムが得られるため、部分計算と比べると非常に効率的である。実際、従来の実行時特

<sup>2</sup>本論文ではプログラム中に現われる文、式などをまとめて式と呼ぶ。

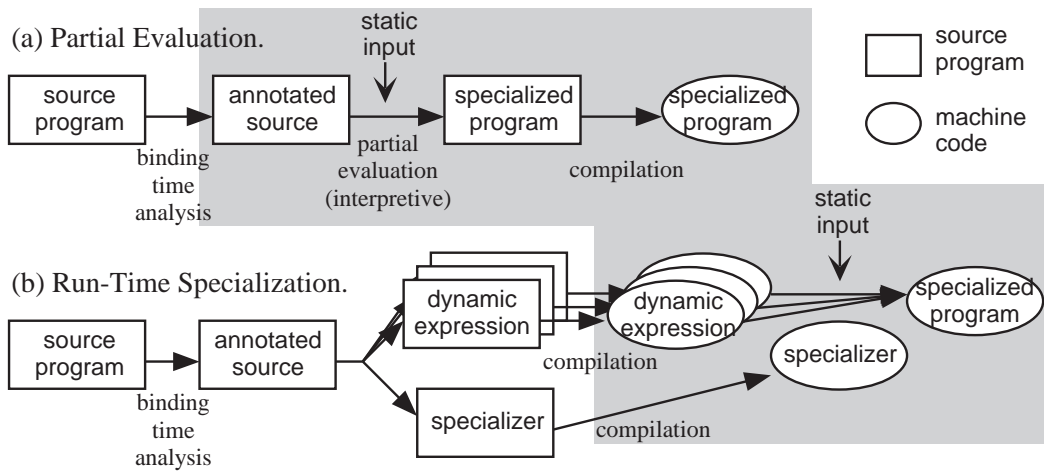


図 1: 部分計算と実行時特化の動作

化の研究では、特化されたプログラムの 1 命令を生成するのに、数命令~数十命令の実行しか要しないことが報告されている [3-5, 9, 10, 12]。

例えば、Intel x86 を対象として power を特化する場合は、(1) 関数の入り口、(2)if 文の then 節、(3)else 節、(4) 関数の出口に対して、図 2 のような命令列 (のバイナリ表現) が用意される。(四角は、特化器の再帰呼出によって生成された命令列が挿入される場所を示す。) 実際に  $n = 3$  として特化器を呼び出した場合は、用意された命令列が 1,1,1,1,2,4,3,4,3,4,3,4 という順で並んだものが特化されたプログラムとして生成される。

(1) 関数の入り口	(2) then 節	(3) else 節	(4) 関数の末尾
<pre>pushl %ebp movl %esp,%ebp pushl %ebx movl 4(%ebp),%ebx</pre>	<pre>movl \$1,%eax</pre>	<pre>pushl %ebx <div style="border: 1px solid black; width: 100px; height: 20px; margin: 5px 0;"></div> imull %ebx,%eax</pre>	<pre>movl -4(%ebp),%ebx movl %ebp,%esp popl %ebp</pre>

図 2: 従来の実行時特化で予め用意される機械命令列

## 2.3 問題点

### 2.3.1 生成コードの効率

実行時特化によって生成されたプログラムは、部分計算によって特化されたプログラムには見られないオーバーヘッドを持つことがある。まず、生成に用いる機械命令列がオフラインで作成されたものなので、用意した命令列をまたがる最適化 (例えばレジスタ割当や命令スケジューリングなど) が行えない<sup>3</sup>。また、ソースプログラム中に関数呼出があった場合、呼出側・被呼出側の特化器がそれぞれ生成した命令列が連続して配置されるものの、これらの命令列間でデータを受け渡すために、関数呼出と同様のレジスタ/スタック操作が必要となる。これらは、ソース言語レベルで特化された場合には見られないオーバーヘッドである。

前出の power の例で見ると: (i) else 節において、power の再帰呼出が生成する命令列へ  $x$  を渡すためにスタックを用いている (pushl %ebx)、(ii) 渡された  $x$  をレジスタ (%ebx) に読み込むために、関数の先頭と末尾でレジスタの退避・復旧が行われている、といったオーバーヘッドが見られる。実際、このプログラムの実行時間を比較すると、従来の実行時特化によって生成されたプログラムは、ソース言語レベルで特化されたものと比べて約 3 倍遅くなっている (7 節参照)。

<sup>3</sup>Strength reduction のように個々の機械命令を最適化することは可能である [9]。

このようなオーバーヘッドは、従来の実行時特化技術の多くが、特化されたコードの効率よりも特化自体の効率を改善を重視していたために生じていたと言える。しかし我々は、インタプリタを特化してコンパイルする場合 [15, 16] のように、特化されたコードが再利用される回数が充分多い場合には、特化の効率をある程度犠牲にしても効率的なコードを生成することで、全体の性能が向上するものも多いと予想する。

### 2.3.2 移植性

実行時特化は、部分計算同様、ソースプログラム上の解析が必要である。さらに、プログラム中の式を部分的にコンパイルして機械語命令列を得るためには、独自のコンパイラを作成する必要もある。そのため、複数の言語へ実行時特化の技術を適用することは容易でない。機械語プログラムを直接解析して特化器を作成できれば、複数の言語への対応は容易となるが、一般的な機械語を解析することは困難であるし、可能であったとしても実行機械に強く依存した解析になってしまうと予想される。

## 3 BCS: バイトコード特化

我々は、スタック型の仮想機械語のプログラムを解析し、その仮想機械語上で特化を行い、生成された仮想機械語命令列を実機械語に変換して実行するバイトコード特化 (BCS: bytecode specialization) を提案する。この方式の特徴は、特化された命令列を実機械語へ変換する際に Just-In-Time コンパイラ技術 ([1] など) による最適化を行うことで、効率的なプログラムが生成可能な点である。また、ソースプログラムを必要としないため、ソース言語から仮想機械語へのコンパイラと、仮想機械語から実機械語へのコンパイラをそれぞれ独立して用いることができ、複数の言語・複数の実機械への対応が容易になっている。

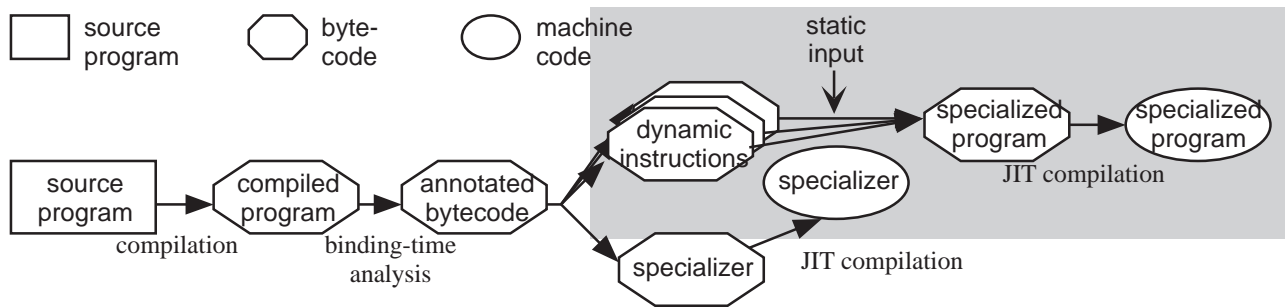


図 3: バイトコード特化の動作

本論文では、仮想機械として Java 仮想機械 (JVM) を選び、機械語への変換は JVM 上の既存の JIT コンパイラを用いる。

図 3 にバイトコード特化系の概略を示す。処理系は、Java などのソース言語からコンパイルされた仮想機械命令列 (compiled program) を受け取り、それを束縛時解析して、特化器 (specializer) を作成する。オンラインの処理では、JIT コンパイルされた特化器に静的な引数が渡されると、特化器は (オフラインで用意されていた) 動的な命令列 (dynamic instructions) をメモリ上に並べることで、特化プログラム (specialized program) を仮想機械語で生成する。このプログラムは、JIT コンパイラによって最適化された実機械語へ変換された後に実行される。

## 4 対象言語: JVM*L*<sub>*i*</sub>

### 4.1 命令セット

簡単のため、本論文では Java 仮想機械語 (JVM*L* と略す) を簡略化した仮想機械語 JVM*L*<sub>*i*</sub> を用いて議論する。JVM*L*<sub>*i*</sub> は JVM*L* と同様、フレーム変数とメソッド呼出の機能を持つスタック機械であるが、以下のような制限が課されている。

- 扱うデータ型は `int` 型のみ
- メソッドはクラスメソッド (`static` 宣言されたメソッド) のみ
- サブルーチン (`jsr` および `ret` 命令)、例外処理は使われていない

JVM*L*<sub>*i*</sub> プログラムは、複数のメソッドから成る。メソッドは、メソッド名、arity、命令列を持つ。命令列は 0 から始まる番地でアクセスされる。命令には次の 7 種類がある:

$$\textit{instruction} ::= \textit{iconst } n \mid \textit{iadd} \mid \textit{iload } x \mid \textit{istroe } x \mid \textit{ifne } L \mid \textit{invoke } m \ n \mid \textit{ireturn}$$

それぞれの命令の基本的な動作は、スタックの先頭から値をいくつかとり出し、計算を行い、結果をスタック上へ積み、というものである。また、メソッド起動毎にフレームが割り当てられ、メソッドの引数やメソッド中の局所変数などは、この上のフレーム変数として読み書きされる。各命令の働きは以下の通りである: `iconst n` は、定数 *n* をスタックの先頭に積む。`iadd` は、スタックの先頭から 2 要素をとり出し、その和をスタックに積む。加算の他に減乗除算 (`isub`, `imul`, `idiv`) などがある。`iload x` は、番号 *x* で指定されるフレーム変数の値をスタックの先頭に積む。`istore x` は、スタックの先頭から要素をとり出し、フレーム変数 *x* に代入する。`ifne L` は、スタックの先頭から値をとり出し、それが 0 でなかった場合は *L* が示す番地へ制御を移す。`invoke m n` は、スタックの先頭から *n* 個の値を引数としてメソッド *m* を呼び出す<sup>4</sup>。具体的には、(1) 現在のフレーム変数および実行番地を退避し、(2) スタックの先頭から *n* 個の値をとり出してフレーム変数 0, ..., (*n* - 1) に代入し、(3) メソッド *m* の 0 番地へ制御を移す。`ireturn` は、メソッドの呼出元へ制御を戻す。具体的には、`invoke` の際に退避していたフレーム変数を復元し、呼び出した `invoke` 命令の次の番地に制御を移す。スタック上の値は、返り値として呼び出し元で使われる。

例として、前出の `power` を JVM*L*<sub>*i*</sub> へコンパイルした結果を図 4 に示す。

Method <code>int Power.power(int,int)</code>	6 <code>iload 1</code> // <i>n</i> を積む
0 <code>iload 1</code> // <i>n</i> を積む	7 <code>iconst 1</code> // 1 を積む
1 <code>ifne 4</code> // <i>n</i> ≠ 0 のときは分岐	8 <code>isub</code> // <i>n</i> - 1 を計算 (第 2 引数)
2 <code>iconst 1</code> // <i>n</i> = 0 のときは 1 を積む	9 <code>invoke Power.power 2</code>
3 <code>ireturn</code> // メソッド終了	// メソッド呼出
4 <code>iload 0</code> // <i>n</i> ≠ 0 のとき: <i>x</i> を積む	10 <code>imul</code> // ( <i>x</i> × 返り値) を計算
5 <code>iload 0</code> // <i>x</i> を積む (第 1 引数)	11 <code>ireturn</code> // メソッド終了

図 4: JVM*L*<sub>*i*</sub> で記述された `power`

## 5 束縛時解析

### 5.1 基本方針

束縛時解析は、JVM*L*<sub>*i*</sub> のメソッド群と、メソッドの各引数に対する束縛時を入力として、メソッド中の各命令の束縛時 (静的か動的であるか) を決定する。ソースプログラム中の変数や式の値は、JVM*L* 上ではスタックやフレームの上に置かれているため、解析においてはこれらを適切に扱う必要がある。特にフレーム変数は、(1) ソー

<sup>4</sup>JVM*L* の `invokestatic` に相当する。

$$\begin{array}{c}
P[pc] = \text{iadd} \\
\frac{P[pc] = \text{iconst } n \quad F_{pc} \subseteq F_{pc+1} \quad T_{pc} = \alpha \cdot B[pc] \cdot \sigma}{A, R, B, F, T, pc \vdash P} \quad \frac{P[pc] = \text{iload } x \quad F_{pc} \subseteq F_{pc+1} \quad T_{pc+1} = \alpha \cdot T_{pc}}{A, R, B, F, T, pc \vdash P} \quad \frac{P[pc] = \text{istore } x \quad F_{pc}[x \mapsto B[pc]] \subseteq F_{pc+1} \quad \alpha \cdot T_{pc+1} = T_{pc}}{A, R, B, F, T, pc \vdash P} \\
\frac{B[pc] \leq \alpha \quad \alpha \leq B[pc] \leq \beta \quad F_{pc}[x] = B[pc] \leq \alpha}{A, R, B, F, T, pc \vdash P}
\end{array}$$

$$\begin{array}{c}
P[pc] = \text{invoke } m \ n \\
\frac{P[pc] = \text{ifne } L \quad F_{pc} \subseteq F_{pc+1} \quad T_{pc} = A_m[n-1] \cdots A_m[0] \cdot \sigma \quad \alpha \cdot T_L = \alpha \cdot T_{pc+1} = T_{pc} \quad \alpha \leq B[pc]}{A, R, B, F, T, pc \vdash P} \quad \frac{P[pc] = \text{ireturn} \quad T_{pc} = \alpha \cdot \epsilon \quad \alpha \leq R[m] \quad pc = \langle m, i \rangle}{A, R, B, F, T, pc \vdash P} \\
\frac{R[m] \leq \alpha}{A, R, B, F, T, pc \vdash P}
\end{array}$$

図 5: 束縛時解析のための型規則

スプログラム上の異なる局所変数が 1 つのフレーム変数に割り当てられることが許されていたり、(2) 代入による副作用があるため、これらの扱いが問題となる。

本論文で提案する手法は、命令/スタック/フレームの束縛時に対して、型規則によって基本的な制約を与え、制御フロー解析によって副作用に関する制約を加えることで束縛時解析を行う。

一般に部分計算のための束縛時解析の方法としては、型規則を用いるものと、抽象解釈を用いるものが知られているが、本研究では Stata と Abadi が提案した JVM のための型規則 [14] を発展させ、複数のメソッドが扱えるものを作る方法をとった<sup>5</sup>。基本的な方針は、メソッド中の各番地における命令/スタック/フレームに対して型を割り当て、命令の型を、その命令が操作するスタック要素/フレーム変数の型と一致させるような規則を与え、それを満たすような最小の型 (束縛時) の割り当てを計算する、というものである。

## 5.2 定義

命令・スタックの要素・フレーム変数の束縛時を表わす型として、 $S$  (静的) および  $D$  (動的) を用意し、 $S \leq D$  という順序関係を定義する。 $\alpha, \beta, \dots$  は  $S$  または  $D$  をとる変数とする。スタックの型は、スタック中の要素の型をリストにしたもので、空のスタックを  $\epsilon$ 、型  $\sigma$  のスタックに型  $\alpha$  の要素を積んだものを  $\alpha \cdot \sigma$  のように書く。フレームの型は、変数番号からフレーム変数の型への写像として表わす。写像  $f$  の拡張  $f[x \mapsto \alpha]$  を以下のように定義する：

$$(f[x \mapsto \alpha])[y] = \text{if } x = y \text{ then } \alpha \text{ else } f[y]$$

図 5 に型規則を示す。規則は、型環境  $A, R, B, F, T$  の下で、プログラムの実行番地  $pc$  におけるプログラム  $P$  に対する判定  $A, R, B, F, T, pc \vdash P$  に関する条件になっている。実行番地  $pc$  は、メソッド名  $m$  と番地  $i$  の対  $\langle m, i \rangle$  から成り、次番地を与える演算:  $\langle m, i \rangle + 1 \equiv \langle m, i + 1 \rangle$  が定義されている<sup>6</sup>。

$A, R$  はそれぞれ各メソッドの引数および返り値の型を与える。 $A[m][n]$  は、メソッド  $m$  が受け取る第  $n$  引数の型を表わし、 $R[m]$  はメソッド  $m$  の返り値の型を表わす。以下、 $A[m][n]$  のような表記は  $A_m[n]$  と略記する。 $B, F, T$  はメソッド中の各命令番地における、命令/フレーム/スタックの型を与える。 $F_{pc}, T_{pc}$  は、それぞれ実行番地  $pc$  の命令を実行する前のフレーム/スタックの状態を表わす型である。

型規則は、命令の種類ごとに定義されている。

<sup>5</sup> 抽象解釈を用いた場合でも、同様の解析が可能だと思われるが、型規則を用いたものの方が正当性などを証明する場合に有利になると予想している。

<sup>6</sup> JVM*Li* では簡単のため、メソッド中の命令は連続した番地に並んでいるものとしている。

- `iconst`, `iadd`, `iload`, `invoke` のようにスタック上に結果を残す命令は、次番地のスタックの先頭の要素と、命令の型を一致させる。ただし、後で説明するように、静的な命令による結果を動的な文脈で使う、いわゆる `lift` 操作を許すため、不等号の制約を作る。
- `iadd`, `istore`, `ifne` のように、スタック上の値を使用する命令は、その番地のスタックの上の要素の型と命令自身の型を一致させる。先頭要素については同様に `lift` 操作を許すため、不等号の制約を作る。
- フレーム変数を操作しない命令では、現在の実行番地と次の実行番地のフレーム変数の型を一致させる。ただし、制御フローが合流する場合は、次番地のフレームの定義域が大きくなる場合があるので、次のように定義される関係によって制約を与える:

$$F \subseteq F' \equiv \forall x \in \text{Dom}(F). F[x] = F'[x].$$

( $\text{Dom}(F)$  は  $F$  の定義域である。) この関係はまた、フレーム変数の `lift` 操作を許していないが、この点は将来拡張を行う予定である。

- `invoke` は、現在のスタック上の要素と起動されるメソッドの引数の型を一致させる。さらに、次番地のスタックの先頭要素と、メソッドの戻り値の型を一致させる。この規則は、一つのメソッドの引数に通りの束縛時を持つことを要請しているため、`monovariant` な解析に相当する。
- `ireturn` は、現在のスタックの先頭要素と現在のメソッドの戻り値の型を一致させる。

メソッド全体の制約: メソッド全体に対する制約は、メソッド中の全ての番地に対する制約と、メソッドの 0 番地におけるスタックおよびフレーム状態への制約として与えられる:

$$\frac{\forall pc \in \text{Dom}(P). pc = \langle m, i \rangle, \quad A, R, B, F, T, pc \vdash P \quad T_{\langle m, 0 \rangle} = \epsilon \quad F_{\langle m, 0 \rangle} = [j \mapsto A_m[j], \dots \forall j \in \text{Dom}(A_m)]}{A, R, B, F, T, m \vdash P}.$$

副作用の扱い: 上記以外の制約として、動的な条件分岐中の副作用を打ち消す制約が与えられる。例えば、次のような式において変数  $x$  は動的、 $y$  は静的であったとする。

```
if (x==0) { z = f(y); } else { z = g(y); }
```

このとき、 $f(y)$  や  $g(y)$  は静的に実行可能であるが、`if` 式終了後の  $z$  の値は  $x$  の値に依存するため動的とする必要がある。そこで、(詳細は略すが) 制御フロー解析を行い、各条件分岐命令  $P[pc] = \text{if } L$  について、`then/else` 節が合流する番地の集合  $M_{pc}$  を求め、各  $pc' \in M_{pc}$  について `then/else` 節中の `istore` の対象となっているフレーム変数のリスト  $l_{pc'} = [x_1, x_2, \dots]$ 、番地  $pc'$  におけるスタック要素のうち `then/else` 節の実行中に積まれた要素数  $n_{pc'}$  を求めておき、次のような制約を加える:

$$\forall pc' \in M_{pc}. B[pc] \leq \beta_{pc'}, \quad T_{pc'} = \overbrace{\beta_{pc'} \cdots \beta_{pc'}}^{n_{pc'}} \cdot \sigma, \quad \forall x \in l_{pc'}. F_{pc'}[x] = \beta_{pc'}.$$

このようにして作られた制約を満たすような束縛時の割り当てを求めることで、束縛時解析が完了する。制約の解き方については省略する。

### 5.3 束縛時解析の例

メソッド `power` の引数について、 $x$  が動的、 $n$  が静的である、つまり、 $A[\text{Power.power}] = [0 \mapsto D, 1 \mapsto S]$  としたときの束縛時解析の結果を図 6 に示す。ソース言語レベルで束縛時解析した場合と同様、 $n$ (フレーム変数 1) に関する計算が静的であるという結果を得ている。

$P$	$B$	$T$	$P$	$B$	$T$
iload 1	$S$	$\epsilon$	iconst 1	$S$	$S \cdot D \cdot D \cdot \epsilon$
ifne L2	$S$	$S \cdot \epsilon$	isub	$S$	$S \cdot S \cdot D \cdot D \cdot \epsilon$
L1: iconst 1	$D$	$\epsilon$	invoke Power.power 2	$S$	$S \cdot D \cdot D \cdot \epsilon$
goto L0	$S$	$D \cdot \epsilon$	imul	$D$	$D \cdot D \cdot \epsilon$
L2: iload 0	$D$	$\epsilon$	goto L0	$S$	$D \cdot \epsilon$
iload 0	$D$	$D \cdot \epsilon$	L0: ireturn	$S$	$D \cdot \epsilon$
iload 1	$S$	$D \cdot D \cdot \epsilon$			

$R[\text{Power.power}] = D \quad \forall pc \in \text{Dom}(F). F[pc] = [0 \mapsto D, 1 \mapsto S]$

図 6: 束縛時解析された power

## 6 特化器の作成

束縛時解析された命令列を変換して特化器が作成される。静的な命令はそのまま特化器の命令となる。動的な命令は、その命令に対応するバイトコードを出力する命令列に変換される。以下に、特化器を作る手順を、コード出力のための疑似命令を備えた命令セットを用いて示す。実際には、疑似命令は JVMML の命令列として実現され、特化器は Java 言語のメソッドとして利用できるが、本論文では省略する。

特化器を構成する命令セットは、JVMMLi の命令セットに、6 つの疑似命令を加えたものである：

$$\begin{aligned} \text{instruction}_g ::= & \text{instruction} \mid \text{GEN instruction} \mid \text{LIFT} \mid \text{LABEL } L \\ & \mid \text{SAVE } n [x_0, \dots] \mid \text{RESTORE} \mid \text{INVOKEGEN } m [x_0, \dots] \end{aligned}$$

以下に、束縛時解析された命令の変換および疑似命令の働きを説明する：

- 静的な命令  $i$  はそのまま特化器メソッドの命令  $i$  となる。
- 動的な命令  $i$  は、GEN  $i$  に変換される。GEN  $i$  が実行されると、命令  $i$  が生成される。つまり、命令  $i$  に対応するバイトコードが、特化されたメソッドの命令列に追加される。
- 命令と、その命令が操作するスタックの要素の束縛時が異なる場合には LIFT 命令が挿入される。具体的には、(1)  $P[pc]$  がスタックに結果を積むような命令で、 $B[pc] = S$  かつ  $T[pc+1] = D \cdot \sigma$  であるときと、(2)  $P[pc]$  がスタックから値をとり出すような命令で、 $B[pc] = D$  かつ  $T[pc] = S \cdot \sigma$  であるときである。LIFT が挿入される場所は (1) の場合はその命令の直後、(2) の場合はその命令の直前になる。LIFT が実行されると、スタックの先頭から値  $n$  がとり出され、命令 `iconst  $n$`  が生成される。

例えば、2 つの引数の積を返すメソッド `int mult(int, int)` を第 1 引数が静的、第 2 引数が動的であるとして束縛時解析すると図 7(a) のような束縛時の割り当てを得る。

<table border="1"> <thead> <tr> <th></th> <th><math>P</math></th> <th><math>B</math></th> <th><math>T</math></th> </tr> </thead> <tbody> <tr> <td>0</td> <td>iload 0</td> <td><math>S</math></td> <td><math>\epsilon</math></td> </tr> <tr> <td>1</td> <td>iload 1</td> <td><math>D</math></td> <td><math>D \cdot \epsilon</math></td> </tr> <tr> <td>2</td> <td>imul</td> <td><math>D</math></td> <td><math>D \cdot D \cdot \epsilon</math></td> </tr> <tr> <td>3</td> <td>ireturn</td> <td><math>S</math></td> <td><math>D \cdot \epsilon</math></td> </tr> </tbody> </table> <p><math>F_{pc} = [0 \mapsto S, 1 \mapsto D]</math></p>		$P$	$B$	$T$	0	iload 0	$S$	$\epsilon$	1	iload 1	$D$	$D \cdot \epsilon$	2	imul	$D$	$D \cdot D \cdot \epsilon$	3	ireturn	$S$	$D \cdot \epsilon$	<pre> mult_gen(int) 0  iload 0 1  LIFT 2  GEN iload 1 3  GEN imul 4  ireturn </pre>
	$P$	$B$	$T$																		
0	iload 0	$S$	$\epsilon$																		
1	iload 1	$D$	$D \cdot \epsilon$																		
2	imul	$D$	$D \cdot D \cdot \epsilon$																		
3	ireturn	$S$	$D \cdot \epsilon$																		
(a) <code>int mult(int, int)</code> の束縛時解析	(b) 作成される特化器																				

図 7: LIFT の例

このとき、フレーム変数 0 が  $S$  なので、0 番地の命令 `iload` の束縛時も  $S$  となる。この命令の結果、スタック上に積まれる値の束縛時は (`imul` 命令の規則によって)  $D$  なので、特化器には図 7(b) のように LIFT 疑似

命令が挿入される。このようにして作られた特化器を `mult_gen(3)` のようにして呼び出すと、LIFT 命令はスタックの先頭の値 (この場合は 3) を使って `iconst` 命令を生成するので、`iconst 3; iload 1; imul` のような命令列が得られる。

- 静的な `invoke m` は、`INVOKEGEN m [x0, x1, ...]` へと変換される。 $x_0, x_1, \dots$  は、その時点での動的なフレーム変数のリストである。INVOKEGEN が実行されると、(1)  $x_0, x_1, \dots$  を退避する命令列および  $m$  の動的な引数をスタックからフレームにコピーする命令列がそれぞれ生成され、(2)  $m$  に対応する特化器が呼び出される。最後に、(3) 退避していた  $x_0, x_1, \dots$  を復旧する命令列が生成される。
- 動的な分岐命令 `ifne L` は、(1) `GEN ifne L; SAVE n [x0, x1, ...]` という命令列に変換され、(2) ラベル  $L$  の位置に `LABEL L; RESTORE` という命令列が挿入される。ここで  $n$  と  $[x_0, x_1, \dots]$  は、`then` 節の実行によって消費される静的なスタックの深さと、破壊され得る静的なフレーム変数のリストである。SAVE が実行されると、スタックの先頭から  $n$  要素のコピーと、フレーム変数  $x_0, x_1, \dots$  の値が退避される。また、RESTORE が実行されると、SAVE によって退避されていたスタックおよびフレーム変数が復元される。

例として、メソッド `power` から作成される特化器 `power_gen` を図 8 に示す。

<pre>Method Power.power_gen(int)   iload_1   ifne L5 L2:GEN iconst_1 L4:ireturn L5:GEN iload_0   GEN iload_0</pre>		<pre>  iload_1   iconst_1   isub   INVOKEGEN Power.power_gen 2 [0]   GEN imul L13:ireturn</pre>
--	--	---

図 8: 疑似命令列を用いた特化器

実際の特化器は、静的な引数に加えて、(1) 命令列を書き込む配列 `byte[] code`、(2) `code` 上の添字 `int i`、(3) 定数を管理するオブジェクト<sup>7</sup> `ConstantPool cp` の 3 つの引数を受け取り、`code[i]` から特化された命令列のバイトコード表現を書き込み、更新された  $i$  の値を返すプログラムになる。手順の詳細は省略するが、例として `power` から作られた特化器の定義を付録 A に示す。

実際に  $n = 2$  として `power` を特化したときに生成される命令列を図 9 に示す。この命令列には明らかに無駄なフレーム変数の操作が含まれている。しかし、JIT コンパイラのレジスタ割当によって、このような命令列からでも最適な機械命令列が生成されることが期待できる。

<pre>Method int power(int)   0 iload_0   1 iload_0   2 istore_1   3 iload_0   4 iload_1</pre>		<pre>  5 istore_0   6 iload_0   7 iload_0   8 istore_1   9 iload_0  10 iload_1</pre>		<pre> 11 istore_0  12 iconst_1  13 istore_1  14 istore_0  15 iload_1  16 imul</pre>		<pre> 17 istore_1  18 istore_0  19 iload_1  20 imul  21 ireturn</pre>
---	--	--	--	---	--	---

図 9: 特化された `power` の命令列

## 7 処理系の現状と性能

プロトタイプとして、JVML のサブセットである `JVMLi` を対象とした処理系が完成している。この処理系では、生成された命令列は Java 言語のクラスローダを用いて読み込まれ、Java プログラム中からメソッド呼び出しができるようになる。

<sup>7</sup>JVM クラスファイルの `constant pool` 領域に対応する。

この処理系を用いて、特化されたプログラムの効率を測定した。メソッド `power(x,30)` の実行時間を (1) 特化しない場合、(2)BCS によって特化した場合、(3) オフラインで特化した、つまり、ソース言語レベルで特化された定義を作り、それをコンパイルして実行した場合、のそれぞれについて測定した。比較のため、従来の実行時特化を想定して、C 言語による実行速度も測定した。この場合の実行時特化は、図 2 に示したようなアセンブリ言語プログラムを手で作成・実行したものである。

実行は、PC 互換機 (PentiumII 300MHz・メモリ 256MB・MS-Windows NT) 上の Sun JDK 1.1.7 (+ Symantec JIT コンパイラ) および Cygnus GCC 2.7.2 を用いた。

	BCS		従来の実行時 特化 (C)		JIT あり	JIT なし
	JIT あり	JIT なし				
特化前	0.74	11.8	1.47	a. 特化に要した総時間	610	724
実行時特化	0.25	7.1	0.83	b. 特化器の実行時間	50	142
静的特化	0.21	1.6	0.26	c. それ以外	560	582

表 1: メソッド `power(x,30)` の実行と特化に要した時間 ( $\mu$  秒)

表 1 (左) から分かるように、JIT コンパイラを利用した場合、BCS によって特化されたプログラムは約 3 倍の速度向上になっており、ソース言語レベルで静的に特化されたプログラムと同程度の速度になっていると言える。一方 JIT コンパイラを用いずに BCS を用いた場合や、従来の (C 言語による) 実行時の特化によって得られたプログラムは、ソース言語レベルで静的に特化されたプログラムと比べてかなり遅い。これは、2.3.1 節で述べたように、特化されたプログラム中にメソッド / 関数呼出に相当するスタックやレジスタ操作が含まれてしまっているためであると考えられる。

表 1 (右) は BCS において特化に要した時間の総計 (a) と、そのうち、(b) 特化器の実行時間と (c) それ以外、つまり、クラスローダや JIT コンパイラの実行時間を示している。この表から分かるように、特化に要する時間の大部分は、特化器以外の実行に費されていることが分かる。JIT コンパイラがある場合とない場合で (c) の時間があまり変わらないことから、JIT コンパイラのオーバーヘッドはあまり大きくないと推定されるが、この点については今後より細かく調べてゆきたい。

## 8 関連研究

Wickline らの実行時特化処理系は、CAM スタック型仮想機械上で特化を行う [17] が、本論文と異なり、特化器はソース言語から作成される。また、我々の知る範囲では実機械語への変換は行われていない。

実行時特化処理系 FABIUS では、生成する実機械語中のレジスタをパラメタ化して、特化時にレジスタ割り当てを行うことができる [9, 10]。また、ICODE は特化器を記述するためのレジスタ型の仮想機械であるが、仮想的なレジスタを用いて命令列を生成した後に、大域的なレジスタ割り当てを行うことができる [12]。

Sperber と Thiemann は、部分計算器とコンパイラの定義をプログラム合成することで、特化器が得られることを示している [13]。ただし、ここで用いられたコンパイラは非常に単純なものであり、一般的なコンパイラに対する有効性や生成されたコードの効率は明らかでない。

## 9 まとめ

本論文では、スタック型の仮想機械上でプログラム特化を行うバイトコード特化を提案した。この方式の特徴は、(1) 従来の実行時特化と同様に、特化器による機械語レベルのプログラム生成によって効率的な特化が行えると同時に、(2) 生成された命令列を最適化することで、より効率的な命令列を生成できるという点である。また、従来の実行時特化技術はソースプログラムを解析することで特化器を作っていたのに対し、我々のものは JVMIL プログラムに対する束縛時解析を提案することで、ソース言語に依存しない技術になっている。解析の際は、ス

タック/フレーム変数の存在や、代入による副作用などが問題となるが、Stata と Abadi が提案した型体系 [14] の拡張と、制御フロー解析によって解決している。現在までに JVMML のサブセットから特化器を作る処理系が作成され、簡単な例については、オフラインで特化された場合と同程度の効率を得られることが確かめられている。

今後は、より実的なプログラムを扱えるよう、フルセットの JVMML を扱えるように枠組を拡張する予定である。基本型および配列に関しては、本論文で示した枠組の単純な拡張で扱えると思われる。オブジェクトについては、束縛時解析のために、メソッド呼出やフィールド操作の規則を適切に与える必要が生じるが、これについては、既存のオブジェクト指向言語を対象とした部分計算の技術を応用できると思われる。また、本論文で示した解析は monovariant なものであったが、これを polyvariant にすることも課題である。

謝辞 スタック機械上で特化を行うという着想は杉田祐也氏との議論から得ました。また、田浦健次朗、浅井健一、小林直樹の各氏および米澤研究室の諸氏からは草稿に対して有益なコメントを頂きました。ここに感謝します。

## 参考文献

- [1] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *PLDI'98*, pp. 280–290, 1998.
- [2] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *POPL'96*, pp. 145–170, 1996.
- [3] D. R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *PLDI'96*, pp. 160–170, 1996.
- [4] D. R. Engler and T. A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *ASPLoS'94*, pp. 263–272, 1994.
- [5] 藤波. オブジェクト指向言語を利用した自動的かつ効率的な実行時コード生成. *コンピュータソフトウェア*, 15(5):25–37, Sept. 1998.
- [6] Y. Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
- [7] J. C. Hardwick and J. Sipestein. Java as an intermediate language. Technical Report CMU-CS-96-161, School of Computer Science, Carnegie Mellon University, 1996.
- [8] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
- [9] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *PLDI'96*, pp. 137–148, 1996.
- [10] M. Leone and P. Lee. Lightweight run-time code generation. In *PEPM'94*, pp. 97–106, 1994.
- [11] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.
- [12] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In *PLDI'97*, pp. 109–121, 1997.
- [13] M. Sperber and P. Thiemann. Two for the price of one: Composing partial evaluation and compilation. In *PLDI'97*, pp. 215–225, 1997.
- [14] R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *POPL'98*, pp. 149–160, 1998.
- [15] 杉田, 増原, 原田. 動的コード生成を利用した自己反映言語の効率的な処理系の実装. プログラミングおよび応用のシステムに関するワークショップ (SPA'98), 1998.
- [16] Y. Sugita, H. Masuhara, K. Harada, and A. Yonezawa. On-the-fly specialization of reflective programs using dynamic code generation techniques. In *Workshop on Reflective Programming in C++ and Java*, pp. 21–25, 1998.
- [17] P. Wickline, P. Lee, and F. Pfenning. Run-time code generation and Modal-ML. In *PLDI'98*, pp. 224–235, 1998.

## A 特化器の定義

本文中の図 8 に示された、疑似命令による特化器は、実際には以下に示すような JVM 命令列に変換される。メソッド `power_gen` の第 1 引数は、もとの `power` の第 1 引数、第 2 引数は命令列を書き込む配列、第 3 引数は配列上の添字、第 4 引数は `constant pool` である。返り値は、更新された添字の値とスタックの深さを要素する

配列である。後者は、JVMでは、メソッドが使用するスタックの最大の深さを宣言する必要があるが、その量は静的な引数の値に依存する可能性があるためである。

```

method int[] Power.power_gen(int, byte[], int, ConstantPool)
// set stack depth
  iconst_2
  istore 4
L0: iload_0
L1: ifne L5
// GEN iconst_1
L2: aload_1
  iload_2
  iconst_4
  bastore
  iinc 2 1
// return index & depth
L3: iconst_2
  newarray int
  dup
  iconst_0
  iload_2
  iastore
  dup
  iconst_1
  iload 4
  iastore
  areturn
// GEN iload_0
L5: aload_1
  iload_2
  bipush 27
  bastore
  iinc 2 1
// GEN iload_0
L6: aload_1
  iload_2
  bipush 27
  bastore
  iinc 2 1
// GEN iconst_1
L8: iconst_1
L9: isub
// INVOKEGEN Power.
// power_gen 2 [0]
// GEN istore_2
L10: aload_1
  iload_2
  bipush 61
  bastore
  iinc 2 1
// GEN iload_1
  aload_1
  iload_2
  bipush 27
  bastore
  iinc 2 1
// GEN iload_2
  aload_1
  iload_2
  bipush 28
  bastore
  iinc 2 1
// GEN istore_1
  iload_2
  bipush 60
  bastore
  iinc 2 1
// extra arguments
  aload_1
  iload_2
  aload_3
  invoke
  Power.power_gen
// extract index
  dup
  iconst_0
  iaload
  istore_2
// extract&update depth
  iconst_1
  iaload
  iconst_2
  iadd
  dup
  iload 4
  if_icmpgt L14
  pop
  iload 4
L14: istore 4
// GEN istore_2
  aload_1
  iload_2
  bipush 61
  bastore
  iinc 2 1
// GEN istore_1
  aload_1
  iload_2
  bipush 60
  bastore
  iinc 2 1
  aload_1
  iload_2
  bipush 60
  bastore
  iinc 2 1
  // GEN iload_2
  aload_1
  iload_2
  bipush 104
  bastore
  iinc 2 1
// return index & depth
L12: iconst_2
  newarray int
  dup
  iconst_0
  iload_2
  iastore
  dup
  iconst_1
  iload 4
  iastore
  areturn

```