操作的意味記述を用いた仮想機械の生成に向けて

脇田 建　　緒方 大介　　佐々 政孝

東京工業大学情報理工学研究科
{wakita,ogata8,sassa}@is.titech.ac.jp

概 要

本システムは、バイトコード命令の操作的な意味記述からインタプリタを自動生成するシステムである。記述実験として133個のバイトコード命令を解釈するObjective Camlのバイトコードインタプリタの記述を試み、15個の命令以外の118個の命令の記述とそれらの命令を処理するコードを得た。いくつかのプログラムを用いて測定した実験では、Objective Camlの手書のインタプリタの9割程度の性能値が得られた。

1　はじめに

プログラミング言語の処理系を一から作ることには大変な労力を伴なう。この労力を少しても低減することを目的にプログラミング言語のバイトコード解釈系を形式的な仕様から自動生成することを目標に研究をはじめた。形式化の体系として処理系の実行時の振舞いを記述する目的から操作的な意味記述を選んだ。

これまでに、多くのプログラミング言語に対して操作的な意味が与えられてきた。また、言語処理系の中間コード[2]やCPUの仕様[4]に操作的な意味が与えることも多かった。近年、Javaなどのバイトコード実行に関心が集まるとともに、研究の題材としてバイトコードの意味が与えられることが増してきた[7, 8]。操作的な意味記述と言語処理系の実装との関連は古く、たとえば[2]では、MLの処理系で中間コードからアセンブリ命令列を出力するためのアルゴリズムの記述に操作的な意味記述が利用されている。

しかしながら、バイトコードに対して与えられた意味記述からバイトコード処理系を自動生成することは、著者らの知る限りあまり試みられていないようである。バイトコードに対して形式的な記述を利用する利点としては、バイトコード解釈系だけでなくバイトコード命令列を操作するさまざまなツール群、たとえばアセンブラや逆アセンブラ、プロファイラ、デバッガなどを自動合成できる点にある。また、意味記述を利用して、与えられたバイトコードプログラムを解析することによって、データや制御の流れの解析にも役立つものと思われる。さらに、[6, 3]などの高速化手法の応用が可能である。さらに、その延長としてのJust-in-Timeコンパイラの生成などの応用が期待できる。

本研究では、バイトコード命令に対する操作的な意味記述から効率的なバイトコード解釈系を生成することを目指す。記述の体系としては、実装に近くコード生成が容易と考えられる操作的意味論を用いた。本稿では、この枠組でインタプリタの仕様を記述し、コードを生成した経験をもとに本方式の可能性と問題点について論じる。記述実験では、Objective Camlのインタプリタを題材にし、記述面と生成されたコードの性能について評価を行った。記述面では、ほとんどの命令の記述が可能となったことから一定の有効性が認められたが、大域脱出などの記述などについてなお問題を残している。性能面では、実行時チェックの除去などの問題を除けば、手書きのインタプリタと同等の性能が得られることが分かった。

本稿の構成は以下の通りである。2節 と3節で仮想機械の構成と命令を記述する方法を述べ、4節でこれらの記述からバイトコード解釈系を生成する方法を述べ、5節で実行性能を評価し、6節で本枠組の可能性などについて論じ、最後に7節でまとめる。

2　仮想機械

仮想機械は、与えられた命令列を順次解釈し、実行するもので、インタプリタの核にあたる。この節では、仮想機械のアーキテクチャがどのように記述されるかについて述べる。

2.1　記法

nのワード $a_1, a_2, \ldots, a_n$ を要素とするワード列$\mathcal{A}$を$\mathcal{A} = a_1 \bullet a_2 \bullet \cdots \bullet a_n$と表す。ここで、$\bullet$は二つの列の連結である。$\mathcal{A}, \mathcal{B}, \mathcal{S}$のような大文字アルファベットは、要素数が0

以上の列を表わし、a, b, x のような小文字は、ワード値、および、そのワード値を唯一の要素とする長さ 1 の列をともに表わすものとする。列について、以下の記法を定義する。

- $\mathcal{S}^i$: i 個の要素を持つワード列
- $\mathcal{S}[n]$: $\mathcal{S}$ の左から n 番目の要素 $((\mathcal{S}^{n-1} \bullet v \bullet \mathcal{S}')[n] = v)$
- $\mathcal{S}\{n\}$: $\mathcal{S}$ の右から n 番目の要素 $((\mathcal{S}' \bullet v \bullet \mathcal{S}^{n-1})[n] = v)$
- $\mathcal{S}[v/n]$: $\mathcal{S}$ の左から n 番目の要素を v で置換したもの。$((\mathcal{S}^{n-1} \bullet v' \bullet \mathcal{S}')[v/n] = \mathcal{S}^{n-1} \bullet v \bullet \mathcal{S}')$
- $\mathcal{S}\{v/n\}$: $\mathcal{S}$ の右から n 番目の要素を v で置換したもの。$((\mathcal{S}' \bullet v' \bullet \mathcal{S}^{n-1})[v/n] = \mathcal{S}' \bullet v \bullet \mathcal{S}^{n-1})$

2.2 仮想機械の構成

インタプリタの記憶空間の基本的な構成は、仮想機械の**レジスタ** と関数呼出しなどに利用する**スタック**、そして動的な記憶領域の割当てを行うための**ヒープ**に分けられる。レジスタには、実機械のワードを保存することができる。スタックはワード列を用いれ表わす。ヒープ上の**オブジェクト**には、ワード境界に配置されたワード列とする。オブジェクトの先頭は、**ヘッダ**と呼ばれオブジェクトの種類を表わす**タグ**とオブジェクトの大きさ (オブジェクトのヘッダ以外の部分が何ワードか) を表わす。n 個の要素 $d_1, d_2, \ldots, d_n$ と、1 バイトのタグ T を持つヒープ上のオブジェクトを以下のように表現する。

$$\mathsf{block}(T, n; d_1 \bullet d_2 \bullet \cdots \bullet d_n)$$

このオブジェクトへの参照には、d_1 を指す 1 ワードのポインタを用いる。Objective Caml のバイトコード解釈系が解するタグには文字列 (String = 252)、浮動小数点数 (Double = 253)、浮動小数点数の配列 (Double_array = 254)、関数閉包に関わるもの (Closure = 250 と Infix = 249)、オブジェクト指向プログラミングのオブジェクト (Object = 248)、ユーザ定義型 (0 . . . 247)、その他 (Abstract = 251、Custom = 255) がある。

ここでは、ハードウェアとその上で実装される仮想機械について、いくつかの制約を与えたが、これらは一般の処理系でも仮定されていることが多い。このため、これらがわれわれの枠組で実現される仮想機械を大きく制限することはないものと考えられる。

図1には、Objective Caml[5] が採用している仮想機械の構成を定義している。それぞれの構成部品には、名前がつけられている。これらの名前は、4節において生成される実装コードに現れる C 言語の変数名として使われる。図1で定義される仮想機械は、プログラムカウンタ pcと、spと名付けられたスタックポインタを持つスタックを有し、5 つのレジスタを持つ。accuレジスタは汎用レジスタであり、この仮想機械の上に実現される Objective Caml の値 (*value*) を持つ。extra_args、global_data、trapspはそれぞれ、Curry 化された関数に適用された引数の数、大域変数を表わす配列、例外処理に関する情報を保存するレジスタである。このうち、global_dataと trapsp は **global**と宣言されている。この宣言は、これらの変数が他の変数のようにインタプリタを実現する関数の中で宣言される局所変数ではなく、大域変数であることを表している。これらはともに本システムの実行時ライブラリの中で定義されており、**global**宣言はそれらとリンクをする働きがある。

```
VM = {
  program_counter = pc;
  stacks = [ sp ];
  registers = [ accu: value;
                env: value;
                extra_args: long;
                global_data: global value;
                trapsp: global pointer ] }
```

図 1: 仮想機械の構成の定義例

2.3 仮想機械での命令実行

仮想機械に与えられるプログラム P は仮想機械のバイトコード命令列である。バイトコードとは、呼ぶが実際にはそれぞれの命令が 1 ワード長であり、正確にはワードコードと呼ぶべきかもしれない。

命令によっては、命令自体で完結しているものの他に、命令に続くワードから整数値の引数を得るものもある。例えば、Objective Caml の MakeBlock 命令は、タグ T を持ち、長さ n のオブジェクトを生成するが、この命令はふたつの整数 T と n をそれぞれ 1 ワードの引数として取る。この命令は、「MakeBlock n T」という 3 ワード命令である。

仮想機械は、与えられたプログラムの先頭の命令から順次、実行していく。P の中の次に実行する命令を指し示すレジスタがプログラムカウンタである。制御命令でない、多くの命令では命令は順番に実行されていく。この場合、プログラムカウンタは、現在、実行している命令の長さだけ増加される。たとえば、MakeBlock命令ならば、プログラムカウ

ンタは 3 だけ増加される。このように命令長に依存したプログラムカウンタの振舞いを考慮するために、$next(\mathsf{pc}) = \mathsf{pc} + \langle$ 命令長 $\rangle$ というものを導入する。

3 仮想機械の意味記述

前節によって、定まる仮想機械の状態についての遷移を用いることによって、仮想機械の命令の意味を定めることができる。

3.1 仮想機械の実行状態

仮想機械の構成に表れるプログラムカウンタ、スタック、レジスタへの値を定めることによって、仮想機械の実行状態が決まる。たとえば、図1が定義する仮想機械が実行状態は、以下のように与えることができる。

$$P \vdash \langle \mathsf{pc},\ \mathcal{S},\ v,\ \varepsilon,\ \delta,\ \chi,\ \mathsf{tsp} \rangle$$

ここで、P は実行中のバイトコードプログラム、pc は現在のプログラムカウンタである。すなわち、$P[\mathsf{pc}]$ が次に実行すべきバイトコード命令を表わす。$\mathcal{S}$ は、スタックに積まれている全てのワード値を列で表わしたものである。$\mathcal{S}$ が空でなければ、スタックトップの値は $\mathcal{S}[0]$、スタックの底の値は $\mathcal{S}\{0\}$ で参照できる。v、ε、δ、χ、tsp はそれぞれ accu、env、extra_args、global_data、trapspに記憶されている値を表わす。

3.2 バイトコード命令の定義

バイトコード命令の定義は、命令の名前、引数、そしてその実行時の振舞いを与えることで定まる。名前と引数についてはすでに述べた。実行時の振舞いは、仮想機械の実行状態の遷移を記述した操作的意味を与えることにする。以下の式は、2 引数を取る命令の操作的意味の記述である。上段が実行状態についての条件を、下段が遷移状態を表現している。

$$\frac{P \vdash \langle \mathsf{pc},\ \mathcal{S},\ v,\ \varepsilon,\ \chi,\ \xi,\ \mathsf{tsp} \rangle \quad P[\mathsf{pc}] = \mathsf{Command}\ a_1 a_2}{P \vdash \langle \mathsf{pc}',\ \mathcal{S}',\ v',\ \varepsilon,\ \chi',\ \xi',\ \mathsf{tsp}' \rangle}$$

ような意味記述を使って、仮想機械のさまざまな命令を記述した。Objective Caml の仮想機械が扱う、133 個のバイトコード命令の記述を試みたところ、15 個の命令を除いたすべての記述ができた。

$$\frac{P \vdash \langle \mathsf{pc},\ \mathcal{S},\ v,\ \varepsilon \rangle \quad P[\mathsf{pc}] = \mathsf{Push}}{P \vdash \langle next(\mathsf{pc}),\ v \bullet \mathcal{S},\ v,\ \varepsilon \rangle}$$

$$\frac{P \vdash \langle \mathsf{pc},\ \mathcal{T}^n \bullet \mathcal{S},\ v,\ \varepsilon \rangle \quad P[\mathsf{pc}] = \mathsf{Pop}\ n}{P \vdash \langle next(\mathsf{pc}),\ \mathcal{S},\ v,\ \varepsilon \rangle}$$

$$\frac{P \vdash \langle \mathsf{pc},\ \mathcal{S},\ v,\ \varepsilon \rangle \quad P[\mathsf{pc}] = \mathsf{Assign}\ n}{P \vdash \langle next(\mathsf{pc}),\ \mathcal{S}[v/n],\ (),\ \varepsilon \rangle}$$

Push命令は accuレジスタ上の値をスタックに載せ、Pop n 命令は、スタックの上から n 個を取り除き、Assign n命令は、accuレジスタの値をスタックの n 番目のスロットに代入する[1]

$$\frac{P \vdash \langle \mathsf{pc},\ \mathcal{S},\ v,\ \varepsilon \rangle \quad P[\mathsf{pc}] = \mathsf{ConstInt}\ i}{P \vdash \langle next(\mathsf{pc}),\ \mathcal{S},\ i,\ \varepsilon \rangle}$$

$$\frac{P \vdash \langle \mathsf{pc},\ v' \bullet \mathcal{S},\ v,\ \varepsilon \rangle \quad P[\mathsf{pc}] = \odot\_\mathsf{Int}}{P \vdash \langle next(\mathsf{pc}),\ \mathcal{S},\ v \odot v',\ \varepsilon \rangle}$$

$$\forall \circ \in \{\mathsf{Add}\ \mathsf{Sub}\ \mathsf{Mul}\ \mathsf{And}\ \mathsf{Or}\ \dots\}$$

整数を扱う場合には、ConstInt命令が、引数に与えられた整数値 i を accuレジスタにロードする。また、整数を扱うさまざまな比較演算子に対応した命令（EQ_INT、NEQ_INT、...）は、accuレジスタとスタックトップから得た値を用いて演算し、その結果を accuレジスタに保存する。計算に使われたスタックトップの値は取り除かれる。なお、レジスタに保存する値を表す式としては、条件節に与えられたワード値のほか、それらを使った計算の結果を許し、それを表現するのに任意の C 言語の式を用いるものとする。

$$\frac{P \vdash \langle \mathsf{pc},\ \mathcal{S},\ v,\ \varepsilon \rangle \quad P[\mathsf{pc}] = \mathsf{Branch}\ \mathsf{pc}_\delta}{P \vdash \langle \mathsf{pc} + \mathsf{pc}_\delta,\ \mathcal{S},\ v,\ \varepsilon \rangle}$$

$$\frac{P \vdash \langle \mathsf{pc},\ \mathcal{S},\ v,\ \varepsilon \rangle \quad P[\mathsf{pc}] = \mathsf{BranchIf}\ \mathsf{pc}_\delta}{P \vdash \langle v?(\mathsf{pc} + \mathsf{pc}_\delta) : next(\mathsf{pc}),\ \mathcal{S},\ v,\ \varepsilon \rangle}$$

分岐命令のひとつである Branch pc_δ命令は引数に与えられたオフセット整数値 pc_δ が指定する命令を次のプログラムカウンタとして与える。条件分岐命令の BranchIf pc_δは accu レジスタに保存された値が「真」である場合には、オフセット整数値 pc_δ が指す命令に制御を与える。ここで「_ ? _ : _」は C 言語の条件演算子である。

$$\frac{P \vdash \langle \mathsf{pc},\ \mathcal{V}^n \bullet \mathcal{S},\ v,\ \varepsilon \rangle \quad P[\mathsf{pc}] = \mathsf{MakeBlock}\ n\ T}{P \vdash \langle next(\mathsf{pc}),\ \mathcal{S},\ \mathsf{block}(T, n; \mathcal{V}^n),\ \varepsilon \rangle}$$

$$\frac{P \vdash \langle \mathsf{pc},\ v \bullet \mathcal{S},\ \mathsf{block}(T, n; \mathcal{V}^n),\ \varepsilon \rangle \quad P[\mathsf{pc}] = \mathsf{SetFld}\ i}{P \vdash \langle next(\mathsf{pc}),\ \mathcal{S},\ \mathsf{block}(T, n; \mathcal{V}[v/i]),\ \varepsilon \rangle}$$

MakeBlock T n命令は、新しいデータ構造の記憶割当てと初期化をする。ヒープ上に記憶領域 ($\mathsf{block}(T, n; \dots)$) を確保し、そこにスタックから取り除いたデータ $\mathcal{V}^n$ を収納する。また、このように作成されたオブジェクトの要素を操

[1]紙面の都合上、特に必要な場合を除いて命令の規則から、実行状態の中から xargs、global_data、trapspの値を省略して表記する。また、いくつかの命令の名前を規則の中で省略して表記する。

作するための命令として、GetFld i命令や SetFld i命令などがある。たとえば、SetFld i命令では、accuレジスタの中のポインタが指すヒープ上のオブジェクト block$(T, n; \mathcal{V}^n)$ の第 i 番目のフィールドにスタックトップの値を代入する。

SetFld i命令の意味規則に表れる block$(T, n; \mathcal{V}^n)$ のように、レジスタの値としてオブジェクトが現れる場合には、そのレジスタがヒープ上に割当てられたオブジェクトを指す参照を持っているものとする。また、MakeBlock n T命令に現れるオブジェクトのように、条件節になかったオブジェクトが状態遷移を経て現れる場合には、このオブジェクトが新たに生成されたものとして取り扱う。さらに、SetFld i 命令のように対応するオブジェクトが状態遷移の前後にともに現れる場合には、それらは同じオブジェクトとして扱う。二つのオブジェクトがひとつの規則に表われる場合には、block2(_, _; _) のように名前を変えることで表現する[2]。

GetFld i命令や SetFld iは accuレジスタに保存されたオブジェクトを操作するための命令であったが、envレジスタや global_dataレジスタに保存されたオブジェクトを操作するための、GetGlobal i、SetGlobal i命令や現在の実行環境から自由変数の値を取り出すための EnvAcc i命令などが定義できる。

関数呼び出しには、関数閉包を作成する Closure命令、その相互再帰版の ClosureRec命令、呼出し直前にリターンアドレスを記憶するための PushRetAddr pc_δ命令、関数適用の Apply n、その末尾再帰版の ApplyTerm n、カリー化関数の実装に使われる Restart命令と Grab n命令などがある。このうち、相互再帰的な関数閉包を作成する ClosureRecは、ひとつの環境を作成し、それを相互再帰的ないくつかの関数閉包で共有させるような構造を作るやや複雑な命令であり、これまでに述べた枠組みのみを用いて簡潔に記述することができなかった。このほかの関数呼び出しに関わる命令の記述は容易であった。

このようにして、Objective Caml の提供する命令すべてを記述することを試みたところ、ほとんどの命令を記述できた。記述が不可能であった命令は、外部関数呼出し、オブジェクト指向機能のためのメソッド探索、デバッガやプロファイラへのインタフェイス、再帰的な関数閉包のデータ構造を生成する命令、大域脱出、パターンマッチに用いられる複雑な分岐命令である。これらについては、6節で詳しく議論する。

[2]Objective Caml のバイトコード命令では、異るオブジェクトを扱う命令はなかった。

```
extern value global_data;
extern value *trapsp;
extern value *stack_sp;

value interprete(code_t prog, asize_t prog_size)
{
  register code_t pc = prog;
  register value *sp = stack_sp;
  value accu = Val_unit;
  value env = Val_unit;
  long extra_args 0;

  インタプリタの状態の初期化コード;

  while (1) {
    switch (*pc++) {
      lbl_Instruction1:
        ⟨Instruction1 命令を処理するコード ⟩;
        goto *(void *)(jumptbl + *pc++);

      lbl_Instruction2:
        ⟨Instruction2 命令を処理するコード ⟩;
        goto *(void *)(jumptbl + *pc++);

        ⋮
  } } }
```

図 2: インタプリタのコードの雛型

4 仮想機械のコード生成

仮想機械のコード、すなわちインタプリタのコードの大部分は前節で述べた命令の振舞いを実装したコードで占められる。そこで、インタプリタを生成するためには、命令の操作的意味記述から効率的なコードを生成すればよい。本研究では、出力する言語として C 言語を用いる。

4.1 インタプリタのコードの雛型

このコードの雛型は図2に示した。ここで、*interpret*関数がインタプリタの本体をなす。各命令は **switch** 文中のラベル付けられたコード断片で実装されている。概念的には、命令実行の各ステップごとにプログラムカウンタで指される命令に対応したコードが実行される。この場合分けは費用がかかるために、GNU C コンパイラが提供する一級ラベルの機能を利用して、threaded code[1] を実現している。

図1で、**global**宣言された要素は大域変数として宣言される。これらは、前述の通り、実行時ライブラリの中の定義とリンクされる。**global**宣言されなかった要素は、*interpret* 関数の局所変数として宣言される。

コードの雛型では、まず宣言された局所変数を初期化する。プログラムカウンタは、*interpret*関数に引数として与えられたバイトコードプログラム *prog*の先頭番地に、スタッ

クポインタは対応するスタックの底に初期化される。スタックは低位番地に向って成長するものとする。

各命令の処理コードを生成するためには、以下の手順を踏む。(1) 命令の引数を読み込むコードの生成、(2) 実行状態の差分の計算、(3) 差分の合成コードの生成。(1) では、引数を取る命令に対して、命令列の中から引数に対応する値を読み込み、引数に対応した変数に代入する。(2) では、命令実行の前後の状態を比較して、実行状態が変化する部分を調べる。(3) では、(2) で抽出された差分を生成するようなコードを生成する。

4.2 引数の処理

switch文の中で pcが増加されているので、命令処理のコードが実行される時点では pcは命令の次のワードを指している。当該命令が引数を取らない場合には、これは次の命令を指している。引数がないので、引数の処理は不要である。引数を取る関数の場合、たとえば Command $a_1\ a_2\ \ldots a_n$命令の場合、バイトコード列の中には、Commandに続いて n 個の引数が記録されている。そして、pcは命令の第一引数を指している。すべての引数を読み込むためのコードは以下のようになる。

```
a1 = *pc++;
a2 = *pc++;
   ⋮
an = *pc++;
```

このコードが実行されたあとでは、pcが当該命令のつぎの命令の番地、すなわち $next$(pc) を指している。このため、当該命令が制御命令であればプログラムカウンタの調整は不要になる。

4.3 差分コードの生成

状態変化は、大きく分けて、プログラムカウンタの変化、スタックの変化、レジスタの変化、オブジェクトの生成、レジスタが指すオブジェクトの変化に分類できる。この中で、スタックとオブジェクトは列構造を取り、特にスタックは列の伸び縮みを考慮する必要があるために複雑である。

プログラムカウンタの変化　すでに伸べたように、引数の処理が済めばプログラムカウンタは次の命令を指している。したがって、通常の命令に対してはプログラムカウンタの処理は不要である。制御命令の場合はプログラムカウンタの変更を伴なうので、プログラムカウンタに新しい値を指定する。たとえば、3 ページに定義された BranchIf命令の場合、以下のようなコードが生成される。

```
case BranchIf: {
    pcδ = *pc++;
    value v = accu;
    { pc' = (v ? pc + pcδ : pc);
      pc = pc'; } }
  break;
```

レジスタの変化　レジスタの値が変化は、レジスタに対応した変数への代入で実現する。たとえば、整数の加算命令 AddIntの場合、accuレジスタには $v + v'$ の結果が代入される。ここで、v と v' は、遷移前の状態を調べることから、それぞれ accuにもともと記憶されていた値とスタックトップの値 (sp[0]) であることが分かる。このため、以下のようなコードが生成される。

```
case AddInt: {
    value v1 = unpack(accu);
    value v2 = unpack(sp[0]);
    { value accu' = v1 + v2;
      accu = pack(accu');
      sp = sp + 1; } }
  break;
```

オブジェクトの生成　遷移後の状態に遷移前のパターンになかったオブジェクトが現れる場合には、オブジェクトを生成するコードを出力する。MakeBlock $T\ n$ の場合は、以下のようなコードが出力される。

```
case MakeBlock: {
    value T = *pc++;
    value n = *pc++;
    { value t1;
      Allocate(t1, T, n);
      オブジェクト t1の初期化
      accu = t1; } }
  break;
```

オブジェクトを生成する命令はこの MakeBlock命令と同様に、生成したオブジェクトを初期化するものが多い。これは、初期化していないフィールドに既に死んだオブジェクトへの参照が残る可能性をなくし、メモリ管理を単純にする目的である。オブジェクトの初期化については、後述のスタックの変化のところで扱う。

レジスタが指すオブジェクトの変化　レジスタが指すオブジェクトの変化は、オブジェクトのフィールドの変化である。それぞれのフィールドを別個のレジスタと見なせば、フィールドの変化はレジスタの変化と同様の扱いが可能である。例えば、SetFld i命令の場合、i 番目のフィールドに代入されているので以下のようなコードを得る。

```
case SetFld: {
    value i = *pc++;
    { value v = sp[0];
      value t1 = accu;
      t1[i] = v;
      sp = sp - 1; } }
  break;
```

スタックの変化　スタックの変化が、レジスタの変化などに比べて複雑な点は、部分列の入れ替えが含まれる点である。例えば、以下に定義される Shift n 命令があったとしよう。

$$\frac{P \vdash \langle \mathsf{pc},\ \mathcal{S}^n \bullet v \bullet \mathcal{S}',\ v,\ \varepsilon \rangle \quad P[\mathsf{pc}] = \mathsf{Shift}\ n}{P \vdash \langle next(\mathsf{pc}),\ v \bullet \mathcal{S} \bullet \mathcal{S}',\ v,\ \varepsilon \rangle}$$

この場合、以下のコードのようにひとつの作業用の変数を利用した効率的なコードを作成することができる。

```
case Shift: {
    n = *pc++;
    { value v = sp[n];
      { int i;
        for (i = n; i >= 0; i--)
          sp[i+1] = sp[i]; }
      sp[0] = v; } }
  break;
```

Shift n命令では、要素数が 1 と n の部分列が位置を替えていたが、次に位置を替える要素の数を m と n に一般化してみよう。

$$\frac{P \vdash \langle \mathsf{pc},\ \mathcal{S}^m \bullet \mathcal{T}^n \bullet \mathcal{S}',\ v,\ \varepsilon \rangle \quad P[\mathsf{pc}] = \mathsf{Swap}\ m\ n}{P \vdash \langle next(\mathsf{pc}),\ \mathcal{T} \bullet \mathcal{S} \bullet \mathcal{S}',\ v,\ \varepsilon \rangle}$$

この場合、コピー元とコピー先という依存関係にしたがって、作業用の変数をひとつだけ使って実現することができるが、かなり複雑なコードになってしまう。

このように、静的に長さが確定しない部分列が複数含まれる命令に対するコードに対応した列の処理コードを生成することは困難である。そこで、本研究では一般的だが性能の劣る単純なアルゴリズムとほとんどの命令に対応でき、効率のよいやや特殊なアルゴリズムを併用することを考えている。それらを使い分ける条件は以下の通りである。

- 条件節のスタックに出現する各部分列のうち、状態遷移後にもスタックに残るものが、すべて同一方向に移動する場合。[3]
- 上記の条件を満たさない命令では、一般的だが非効率的なアルゴリズム1 を利用する。

[3]この条件はほとんどの命令について成り立つものと思われる。例外としては、Scheme 言語の call/cc プリミティブのような一級継続の機能をスタックの複製によって実現する場合がある。

まず、単純なアルゴリズム1から述べる。

アルゴリズム 1 大域変数に十分な大きさの作業領域を割当て、スタックが変化する部分をすべて作業領域に複製し、次に複製された要素毎に意味記述にしたがったスタックの個所に複製する。最後に、スタックポインタをあるべき場所に設定する。

このアルゴリズムを Swap m nに適用すると以下のようなコードが生成される。

```
case Swap: {
    long m = *pc++;
    long n = *pc++;
    if (tmp_size < m+n) {
      tmp_size = (m + n) * 2;
      free(tmp_stack);
      tmp_stack = realloc(tmp_size); }
    { int i;
      for (i = 0; i < m+n; i++) {
        tmp_stack[i] = sp[i]; }
      for (i = 0; i < m; i++) {
        sp[n+i] = tmp_stack[i]; }
      for (i = 0; i < n; i++) {
        sp[i] = tmp_stack[m+i]; } } }
  break;
```

アルゴリズム 2 部分列の動く方向に対して先頭の部分列から順に複製する。部分列の複製に際しては、やはり、部分列が動く方向に対して先頭の要素から順に複製する。つぎに、その他の要素、すなわち、状態遷移前には存在せず、状態遷移後にスタックに加わる要素を複製する。

例えば、以下に定義される末尾再帰呼出しに用いられる AppTerm n s 命令を考えてみよう。

$$\frac{P \vdash \langle \mathsf{pc},\ \mathcal{S}^n \bullet \mathcal{T}^s \bullet \mathcal{U},\ c,\ \varepsilon,\ \chi \rangle \quad P[\mathsf{pc}] = \mathsf{AppTerm}\ n\ s}{P \vdash \langle code(c),\ \mathcal{S} \bullet \mathcal{U},\ c,\ c,\ \chi + n - 1 \rangle}$$

この命令では、$\mathcal{S}$ はスタックの底の方向に移動し、$\mathcal{U}$ は不動、$\mathcal{T}$ は消滅するために、アルゴリズム2の条件を充す。これらのなかの移動する部分列、すなわち $\mathcal{S}$ のみについて、移動方向、すなわちスタックの底に近い要素から順番に複製するようなコードを出せばよい。

```
case AppTerm: {
    long n = *pc++;
    long s = *pc++;
    { int i;
      for (i = n-1; n>0; n--) {
        sp[s+i] = sp[i]; }
      sp = sp + s; } }
  break;
```

4.4 実装

与えられた操作的意味記述から、命令の実装を出力するバイトコードインタプリタの生成器を実装した。入力に与える意味記述は、下記のような記述である。

```
push =
  [pc, S, v, e, k,,] -> [, v @ S, v, e, k,,]

apply n =
  [pc, S, closure, env, x,,] ->
  [closure[0], S, closure, closure, {n - 1},,]

makeblock wosize tag =
  [pc, S^(wosize - 1) @ T, v, e, x,,] ->
  [, T, Block(tag, v @ S^(wosize - 1)), e, x,,]
```

生成器は 800 行余りの Objective Caml のプログラムである。本節で述べたアルゴリズムを素朴に実装したものである。

5 評価

前節までに述べた記述系、および、C 言語への変換系を実装し、その性能を既存のインタプリタと比較した。比較にあたって、Objective Caml 3.00 のバイトコード命令群に対して意味記述を与え、それからインタプリタのコードを合成した。Objective Caml の既存のインタプリタコードをこのように合成されたコードと入れ替えることによって、われわれのバイトコード処理系を用いて任意の Objective Caml のプログラムを実行できるようになった。ただし、われわれが十分に記述できなかった 15 個の命令については、オリジナルのコード断片を借用した。また、われわれのシステムが生成するコードはスタックオーバーフローの検査を行わないので、Objective Caml のインタプリタのスタックオーバーフローのコードは除去して比較した。

まず、われわれのシステムが既存のシステムと等価であることを確認するために、5 万行以上の ML のコードが含まれる Objective Caml システムをわれわれのシステムを用いて再構成できることを確認した。

また、性能を比較するために 13 のベンチマークプログラムを用いて性能を比較したものが表1である。プラットフォームは Athlon 800MHz、256MB、L1 Cache 128K、L2 Cache 256K、Linux 2.2.18 であり、いずれの結果も 5 回の試行の平均である。T_{caml} が Objective Caml の性能であり、T_{ours} はわれわれのインタプリタの性能である。最悪の場合 (KB) でも 85%の性能を維持しており、このアプローチの十分な実用性を示唆しているようである。この実行時間をインタプリタと各種の実行時システムの実行時間に分類して計測したデータを図3にまとめた。

表 1: 実行速度の比較：第二、第三列は実行時間 (秒)

Program	T_{ocaml}	T_{ours}	T_{ours}/T_{ocaml}
Boyer	0.280	0.290	96.55%
FFT	2.676	2.792	95.84%
Fib	0.256	0.272	94.11%
KB	1.404	1.638	85.71%
Life	0.734	0.704	104.2%
Mandelbrot	6.728	6.816	98.70%
Nucleic	1.170	1.236	94.66%
QuickSort	0.876	0.892	98.20%
Simple	7.002	7.860	89.08%
Soli	0.474	0.516	91.86%
Takc	0.394	0.412	95.63%
Taku	0.710	0.776	91.49%
TSP	5.164	5.626	91.78%

以上のベンチマークは、実行時間による比較である。生成されたインタプリタの性能を比較するためには、もう少し厳密な計測が必要となる。表**??**は Objective Caml のインタプリタにプロファイラを適用して、各ベンチマークを実行し、言語実装のいずれの部分で実行時間が消費されているかを調べたものである。これによれば、多くのベンチマークでは実行時間の多くをインタプリタの実行が占めるために、インタプリタのチューニングが全実行時間に与える潜在的な影響が大きいことが分かる。また、浮動小数点演算、配列のアクセス、比較演算などに性能向上の余地があまりないことを考えると、自動生成されたコードの効率の高さが、全実行性能の劣化を最小限にとどめていることが分かる。

最後に、インタプリタの実行に占める記述できなかった命令の実行の占める割合を調べた。Objective Caml のコンパイラにおいて Switch命令の実行回数が全体の 3%であるのが目立つほか、PushTrap命令が 0.51%に過ぎないことがわかった。このことから、図3に得られた結果が自動生成されたコードの性能を表すものと考えてもよい。

以上を総合すると、意味規則から生成されたコードの性能が実用的水準に達していることがわかる。さらに、これまでに記述できなかった命令についても同程度の性能を持つコードを自動生成できた場合に、インタプリタの性能が手書きのインタプリタの性能と同程度のものになることが予想される。この可能性については、6.1で論じる。

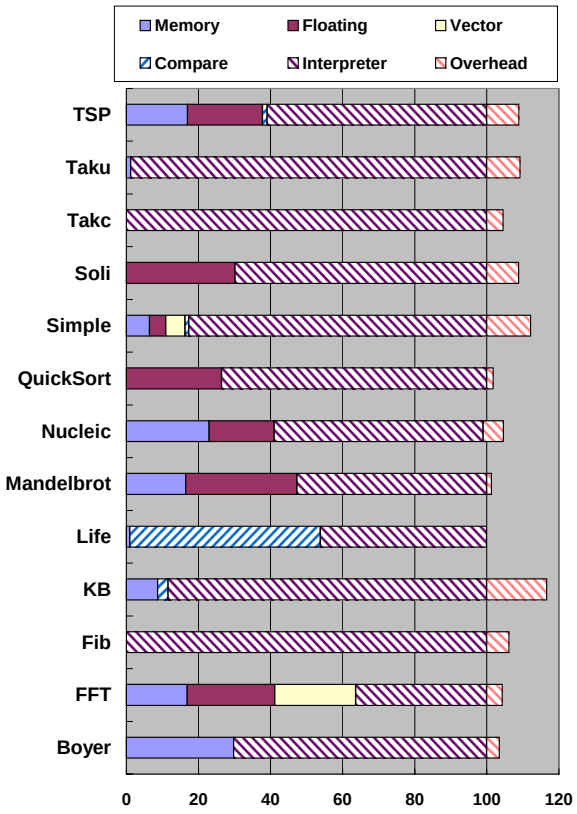

図 3: 実行時間に占めるインタプリタ実行の割合

6 議論

6.1 記述性

表2に本枠組で試みた Objective Caml のバイトコード命令を機能によって分類し、それぞれについて記述に成功した命令数を列挙した。ここで記述に成功したもの (OK) は、われわれのシステムで記述をし、そこから C 言語の実装を合成できたものの数であり、記述に失敗したもの (NG) はそれ以外を指す。表から、一割あまりの命令については記述できていないことが分かる。これらのうち、表の中で四角で囲った、C 言語のライブラリ関数を呼出す外部関数呼び出しとメタ機能はプログラミング言語の意味から逸脱したものであり、本質的に完全に記述することができない。

6 つの外部関数呼び出し命令は、C の関数が受け取る引数の数の違いだけで同じ振舞いをする。これらについては、コードを生成することはできないが、以下のような記述をすることで命令の実行前後で変化しうるレジスタなどを明記することはできる。この情報は、バイトコード上でのデータフロー解析に役立つ可能性がある。

$$\frac{P \vdash \langle \mathsf{pc},\ a^i \bullet S,\ v,\ \varepsilon, \chi \rangle \quad P[\mathsf{pc}] = \mathsf{C\_Call}_i\ \mathsf{f}}{P \vdash \langle \mathit{next}(\mathsf{pc}),\ S,\ v',\ \varepsilon, \chi \rangle}$$

3 つのメタ機能は、Stop命令, Event命令, Break命令である。Stop命令は、プログラムの最後の実行命令であり、インタプリタの実行が終了させる。Event命令と Break命令はデバッガとのインタフェイスであり挿入されたブレイクポイントなどに相当するものである。

表 2: バイトコード命令の分類とその記述性

命令の種類	Total	OK	NG
スタック操作	21	21	
環境操作	10	10	
関数適用	22	21	1
大域変数	5	5	
記憶領域確保	9	9	
オブジェクト	12	12	
配列	3	3	
文字列	2	2	
条件分岐	5	4	1
例外処理	3	0	3
外部関数呼出し	6	0	6
スカラ	31	31	
メソッド探索	1	0	1
メタ機能	3	0	3
合計	133	118	15

外部関数呼び出しとメタ機能はいずれもバイトコードに固有の機能というより、言語処理系の構成に依存したものである。このため、さまざまなバイトコード命令セットで共通に利用できるインタフェイス命令としてデフォルトで提供するのが自然である。

GetMethodは、オブジェクト指向プログラミングのために提供されている命令である。これは、オブジェクトのクラスのメソッドテーブルから該当するメソッドを取り出すものである。この記述をするためには、ネストしたオブジェクトから与えられたパス表現に相当するフィールドを取り出すような表現力が必要である。

関数適用のための 22 個の命令には関数閉包を作成する Closure命令、Curry 化関数の呼出しに用いる Grab命令、Restart命令、関数適用の Apply、末尾再帰呼び出しの Appterm命令、動的リンクを確保する PushRetaddr命令などが含まれる。これらのうち、ClosureRec命令以外のものはすべて記述が可能であった。ClosureRecは、相互再帰的な関数群について、環境を共有するいわゆる shared closure を生成する命令である。関数型言語の *map*のような記法を利用して、object representation を操作すれば記述することは可能である図4。しかし、*map*のような非標準的な演算子を導入するためらいから、現時点では保守的な立場から、記

$$\frac{P \vdash \langle \mathsf{pc},\ \mathcal{S}^n,\ v,\ \varepsilon,\ \chi,\ \xi,\ \mathsf{tsp}\rangle \quad P[\mathsf{pc}] = \mathsf{PushTrap}\ \mathsf{pc}_\delta}{P \vdash \langle \mathit{next}(\mathsf{pc}),\ (\mathsf{pc} + \mathsf{pc}_\delta) \bullet \mathsf{tsp} \bullet \varepsilon \bullet \chi \bullet S,\ v,\ \varepsilon,\ \chi,\ \xi,\ n+4\rangle}$$

$$\frac{P \vdash \langle \mathsf{pc},\ \mathcal{T} \bullet \mathcal{S}^{\mathsf{tsp}'},\ v,\ \varepsilon,\ \chi,\ \xi,\ \mathsf{tsp}\rangle \quad P[\mathsf{pc}] = \mathsf{PopTrap}}{P \vdash \langle \mathit{next}(\mathsf{pc}),\ \mathcal{S},\ v,\ \varepsilon,\ \chi,\ \xi,\ \mathsf{tsp}'\rangle}$$

$$\frac{P \vdash \langle \mathsf{pc},\ \mathcal{T}^\star \bullet \mathsf{pc}' \bullet \mathsf{tsp}' \bullet \varepsilon' \bullet \chi' \bullet \mathcal{S}^{\mathsf{tsp}'},\ v,\ \varepsilon,\ \chi,\ \xi,\ \mathsf{tsp}\rangle \quad P[\mathsf{pc}] = \mathsf{Raise}}{P \vdash \langle \mathsf{pc}',\ \mathcal{S},\ v,\ \varepsilon',\ \chi',\ \xi,\ \mathsf{tsp}'\rangle}$$

$$\frac{P \vdash \langle \mathsf{pc},\ S,\ \mathsf{block}(T; v^p),\ \varepsilon, \chi\rangle \quad P[\mathsf{pc}] = \mathsf{Switch}\ L^m\ l^n}{P \vdash \langle \mathsf{pc} + L[T],\ S,\ \mathsf{block}(T; v^p),\ \varepsilon, \chi\rangle}$$

$$\frac{P \vdash \langle \mathsf{pc},\ S,\ v,\ \varepsilon, \chi\rangle \quad P[\mathsf{pc}] = \mathsf{Switch}\ L^m\ l^n\ v < n}{P \vdash \langle \mathsf{pc} + l[v],\ S,\ v,\ \varepsilon, \chi\rangle}$$

$$\frac{P \vdash \langle \mathsf{pc},\ S,\ v,\ \varepsilon, \chi\rangle \quad P[\mathsf{pc}] = \mathsf{Switch}\ L^m\ l^n\ v \geq n}{P \vdash \langle \mathsf{pc} + m + n,\ S,\ v,\ \varepsilon, \chi\rangle}$$

$$\frac{P \vdash \langle \mathsf{pc},\ S,\ v,\ \varepsilon,\ \chi,\ \xi,\ \mathsf{tsp}\rangle \quad P[\mathsf{pc}] = \mathsf{ClosureRec}\ m\ 0\ \mathsf{pc}_\delta\ f^m}{P \vdash \langle \mathit{next}(\mathsf{pc}),\ S,\ \mathit{closure}(\mathit{next}(\mathsf{pc}) + \mathsf{pc}_\delta, \mathit{map}(\mathit{infix}, f^m), ()),\ \varepsilon,\ \chi,\ \xi,\ \mathsf{tsp}\rangle}$$

図 4: 記述が難しい命令

述不可能とした。

条件分岐には、BranchIfと BranchIfNotのほか、ML のパターンマッチの実装に利用される Switch命令が含まれる。この命令のやや複雑な振舞いは図4のように記述することができる。この記述からコードを生成することは原理的には可能と思われるが、われわれの実装が不十分なために現時点ではコードの自動合成はできていない。

PushTrap命令、PopTrap命令、Raise命令は大域脱出を実現する。PushTrap命令と PopTrap 命令はそれぞれ例外処理ハンドラの登録と削除、Raise命令は例外処理ハンドラの起動にあたる。Objective Caml の実装では、例外ハンドラを表現する関数閉包はスタックにプッシュされ、PopTrap命令はスタック中の例外ハンドラのアドレスを tsp レジスタに記憶する。本枠組では、ポインタを明示的にあつかっていないために、これを直接的に表現することができない。この代わりに例外ハンドラが保存されている場所のスタックの底からのオフセットを表現することで、同等の意味を記述することができる図4。ただし、この記述は2節で $\mathcal{S}^n$ の n が静的に定まる定数とした仮定について拡張がなされている。このため、アルゴリズムを拡張しなければコードの生成ができず、現段階で記述できないものに分類した。今後、上述のより一般的な場合に対応することで、大域脱出命令の扱いを可能にしたい。また、ポインタでなくオフセットを利用することの性能に対する影響も調査を要する。ただし、例外処理であれば性能全体に与える影響は小さいものと予測される。

以上をまとめると、記述ができなかった 15 の命令のうち、9 命令は言語の意味を越えた機能であるので、デフォルトのメタ命令として提供する。ClosureRec命令と GetMethod 命令については、記述面の不足があるため、今後、検討しなくてはならない。そのほかの条件分岐と例外処理については、われわれの枠組のわずかな修正によって対応できる感触を得ている。

われわれのシステムを用いて記述された Objective Caml のバイトコードの意味記述は 354 行で、オリジナルの 984 行の C 言語のコードよりもややコンパクトである。記述量に著しい優位は認められない。しかし、形式的な記述を利用することの利点は、むしろ、記述の直観性、それを利用した解析、最適化などの応用性、関連した言語処理ツール生成の補助などである。このため、単純な定量的な比較はあまり意味を持たない。

そうはいっても、実際のインタプリタのコードには本研究が生成するバイトコードの解釈部だけでなく、基本データ構造などを操作する実行時ライブラリ、メモリ管理やシグナル操作などの実行時処理部、I/O 部などから構成されており、それらのコード量の方がはるかに多い。例えば、Objective Caml の場合、これらのコードの量は 1 万 1 千行余りに達している。今後は、インタプリタ核だけでなく、これらの部分のコードの共通化、部品化、形式的記述に基づいた自動合成などを試みたい。これに関連して、[9] では記憶管理機構の部品化と部品化に伴なう性能劣化を防ぐ処理系特化の手法について述べた。本研究で行ったインタプリタ核の自動生成技術と合わせると、インタプリタ実装の 25%のコードの部品化に成功したことになる。

6.2 実装上の問題

インタプリタの核には、バイトコード命令の処理の記述のほかに、実行時システムとのやりとりをするコードが含まれる。たとえば、スタックオーバーフローのチェックやシグナル処理などである。これらは、現実装では、明示的に記述されていない。このため、これらのコードを生成することはできない。スタックオーバーフローは、スタックを述ばす命令についてそれぞれ検査を行ってもよい。この方法を試みたところ全実行時間に対して 20%以上ものオーバーヘッドとなることがわかった。

多くのインタプリタは、関数にスタックフレームを割当てる時点でこの関数が消費するスタックフレームの大きさの情報などを利用してまとめて検査を済ますことで、この検査に要する費用を小さく抑えている。Objective Caml では、各関数がに割当てられるスタックフレームの最大値がコンパイラによって決められているおり、その値を利用してオーバーフローの検査をしているようである。このような最適化を実現するためには、バイトコードプログラムの静的な解析によって各スタックフレームが消費するスタック量を調べる方法、あるいは、仮想機械の定義にスタックフレームの最大サイズを与えることが考えられる。

6.3 応用

バイトコードの意味記述を利用することでインタプリタとその周辺のいくつかの言語処理系が生成できるものと思われる。まず、定義されたバイトコード命令のフォーマットが定義されていることから、アセンブラとディスアセンブラを作成することができる。これらのツールを与えられた命令セットの仕様にしたがって実装することは容易である。しかし、命令セットの設計段階でインタプリタの命令セットと整合を取りながらこれらを実装することは甚だ面倒であるため、インタプリタの仕様からシステマティックに生成できることは有益である。

インタプリタ核にいくつかのメタ命令を追加することでバイトコードレベルのデバッガやプロファイラを作成することが可能であろう。これは、すでに多くのバイトコード処理系処理系で実践されていることである。

インタプリタの高速化の単純な技法には、スタックトップのいくつかの値をレジスタにキャッシュすることで無駄なスタック操作を避ける手法 [3] と頻繁に出現するバイトコード命令列をインライン化してインタプリタの命令フェッチの費用を節約する手法 [6] が提案されている。これらの手法で利用されているコードの解析技術は単純なものであり、本研究で与えられる形式仕様に対して適用することによって、インタプリタの性能を飛躍的に向上できるものと考えられる。

Java の処理系の多くが備えているような性能のよい JIT コンパイラでは、スタックフレームの構造を解析することでレジスタ割り付けをし、従来のコンパイラの内部形式と同様のレジスタ機械に変換した上で RTL の最適化を行われることが多い。このような解析は、Java が JVM に課しているスタックフレームのサイズや型についての制限が可能にしている。同様の制限を意味記述に設ければ、少なくとも局所的な最適化に関して Java JIT と同程度のことが可能になるものと期待できる。クラス階層解析のような言語に特化した大域的な解析手法は困難であろう。

7 まとめ

バイトコードインタプリタの解釈系の実装に対して、操作的意味を与えられること、そして操作的意味記述から比較的効率的な実行系を合成できることを述べた。

記述力については、Objective Caml のバイトコード命令をほとんど記述でき、そこから正しいと思われる処理系が生成できたことから一定の有効性を確認した。今後は、Smalltalk, Scheme, Java などのバイトコード処理系の記述も試みたい。

Objective Caml の命令のうち扱いが不可能であった命令には、デバッガへのインタフェイス (3)、外部関数呼出し (1) などのように本質的に記述が困難なものに加えて、例外処理の機能の一部 (3)、メソッド探索 (6)、相互再帰関数閉包の生成 (1)、パターンマッチ (1) など高度な言語機能を内包した命令があった。これらの扱いは今後の課題に残されている。

ベンチマーク性能については、本システムが不完全なプロトタイプであることを考慮すると良好と思われる。今後は、一部のベンチマークが遅い原因を探し、性能改善に役立てたい。

謝辞

PPL 査読者の方々からは投稿原稿を丁寧に読んでいただいた上で、多くの有用なコメントを頂きました。深く感謝いたします。議論につきあって下さった本研究のアイデアについて議論してくださった、数理・計算科学専攻の仲間に感謝します。本研究は文部科学省科学研究費特定領域研究 (B)「社会基盤としてのセキュアコンピューティングの実現方式の研究」から援助を受けています。

参考文献

[1] J. Bell. Threaded code. *Communication of ACM*, Vol. 16, No. 6, pp. 370–372, June 1973.

[2] L. Cardelli. Compiling a functional language. In *Proc. ACM Symposium on Lisp and Functional Programming*, 1984.

[3] M. Anton Ertl. Stack caching for interpreters. In *Proc. SIGPLAN conference on Programming Language Design and Implementation*, pp. 315–327, 1995.

[4] Intel. IA-64 アセンブリ言語リファレンス・ガイド , 2000. 資料番号：245363J-001.

[5] X. Leroy. *Objective Caml*, 1997. Available from http://pauillac.inria.fr/ocaml/.

[6] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. In *Proc. SIGPLAN conference on Programming Language Design and Implementation*, pp. 291–300, 1998.

[7] R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proc. symposium on Principles of Programming Languages*, 1998.

[8] 増原英彦, 米澤明憲. Java バイトコード上での実行時プログラム特化. ソフトウェア科学会 SPA '99 予稿集, 6 月 1999.

[9] 内山雄司, 脇田建. メモリ管理の性能評価基盤. 情報処理学会第 29 回プログラミング研究会予稿集, 6 月 2000.