仮想機械の仕様記述に基づくバイトコードインタプリタ生成系

内山雄司[†] 緒方大介[††] 脇田建[†]

著者らが開発したバイトコードインタプリタ生成系である Virtual Machine Builder（VMB）について述べる．VMB は，仮想機械の仕様記述を入力として，仮想機械の中核をなすインタプリタの実装を生成するシステムである．VMB に与える仕様記述は，仮想機械を構成するレジスタやスタックの定義と，仮想機械で実行される各バイトコード命令の意味の定義からなる．バイトコード命令の意味は，仮想機械の状態遷移の形で表現される．以前の VMB には，データ型の宣言やオブジェクト間の参照の表現力に不十分な点があり，バイトコード命令の意味を記述する際に特別な工夫を必要とする場合があった．本研究では，仕様記述言語を再設計して，より自然な形で仮想機械の仕様を記述できるように工夫した．VMB を適用した処理系開発の事例として，簡単な手続き型言語に対するバイトコード言語を設計し，その処理系を実装した．さらに，VMB を利用して Objective Caml のインタプリタを実装し，仕様記述言語の表現力と生成されるインタプリタの性能という 2 つの観点から VMB の評価を行った．表現力については，146 命令のうち 133 命令を自然な形で記述できた．インタプリタの性能については，ベンチマークテストの結果，手書きのインタプリタで実行した場合の 97%の性能が得られた．

A Bytecode Interpreter Generator Based on Virtual Machine Specifications

Yuji Uchiyama,[†] Daisuke Ogata[††] and Ken Wakita[†]

The article proposes Virtual Machine Builder (VMB) which is a bytecode interpreter generator developed by authors group. VMB takes a specification of a virtual machine and generates implementation of the interpreter which composes a main part of the virtual machine. A specification of a virtual machine comprises a definition of a virtual hardware composed from registers and stacks, and a definition of the behavior of each virtual machine instruction. The behavior of an instruction is specified in terms of a machine state transition system. In the previous version of VMB, there were cases that we needed some tricks to specify the behavior of an instruction since the specification language lacked the way to declare data types of objects and to express references between objects. We redesigned the language so one could specify instructions more naturally. The article also shows an experience in developing a tiny imperative language system using VMB. Also the expressiveness of the specification language and the performance of the generated interpreter are evaluated by the use of a virtual machine of Objective Caml generated by VMB. On the former point, 133 of 146 bytecode instructions can be defined naturally. On the latter point, a benchmark test shows that the efficiency of the generated virtual machine is 97% of a carefully implemented human-crafted one.

1. はじめに

近年，ネットワーク接続環境の普及や計算機ハードウェアの性能向上を背景として，原始プログラムをバイトコードと呼ばれる中間形式にコンパイルし，仮想機械と呼ばれるソフトウェアを用いてこれを実行するという，バイトコード実行方式が広まっている．この方式で実行される言語の代表には Java[3] などがある．バイトコード実行方式には異機種間でのバイナリ互換性やプログラムの検証可能性といった利点があり，ネットワークを介したプログラム実行との親和性が高い方式であるといえる．その一方で，この方式でのプログラム実行には仮想機械というソフトウェアが介在するため，機械語を直接実行する方式に比べて処理速度が遅いという欠点がある．しかし，近年のハードウェア性能の著しい向上や最適化技法の発展により，応用の分野によっては，このことはもはや重大な欠点ではなくなりつつある．

† 東京工業大学大学院情報理工学研究科数理・計算科学専攻
Department of Mathematical and Computing Science, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
†† ソニー株式会社
Sony Corporation

一方，プログラミング言語処理系の開発者にとって，バイトコード処理系の開発は困難な作業である．バイトコード処理系はコンパイラと仮想機械から構成される．このうち，後者の仮想機械の設計にはハードウェア構成や内部データ構造の表現形式，命令セットの設計といった広い自由度があり，慎重に検討と実験を重ねながら開発を進めていかなければならない．このことに対して，さまざまなプログラミング言語から用いられる共通の仮想機械を基盤として提供する .NET[6] のような立場もある．しかし，プログラミング言語の特長を生かして効率の良い仮想機械を開発することを考えると，独自に仮想機械を開発する必要性は依然として大きい．

そこで本研究では，仮想機械の設計仕様を形式的に記述し，その記述をもとに生成系を利用して仮想機械の実装を自動生成する手法を提案する．我々が開発したバイトコードインタプリタ生成系 Virtual Machine Builder（VMB）は，そのような形式的記述から，仮想機械の実装の中心部分であるバイトコードインタプリタを生成するシステムである．開発者は，VMB を用いてバイトコードインタプリタを生成し，これにメモリ管理機能や入出力機能などの諸機能を追加することによって，仮想機械全体を構築できる．

このような開発手法には，仮想機械の開発時間の短縮や保守性の向上という利点がある．形式的な記述から仮想機械の実装を自動的に生成する手法によって，仮想機械の設計上の変更を実装に反映させることが容易となる．また，設計の変更を繰り返すうちに実装にバグが混入してしまう可能性も排除でき，保守性の向上にも役立つ．これらの利点は，言語処理系を開発する際に有用なものであると考えている．また，仮想機械の仕様を形式的に記述する方法を与えることには，その記述の上でさまざまな解析や最適化を行うという応用が考えられ，特定の言語に依存しない形で最適化アルゴリズムを記述するといった応用も期待できる．

本研究は，我々が過去に提案した枠組み[7),8),12)] を大幅に拡張したものである．過去の枠組みには，仮想機械の仕様を記述するための言語に柔軟性を欠く点があり，ユーザが自由にデータ型を定義できない，メモリ操作をともなうバイトコード命令の一部に記述できないものがあるという問題があった．本研究では記述言語を再設計し，これらの問題を解決した．また，インタプリタ生成アルゴリズムも新たなものとし，より効率の良いインタプリタの生成を可能とした．

本稿の次章以降の構成を示す．まず 2 章で，我々の開発した VMB が仮想機械の開発工程においてどのように利用されるシステムであるかを述べ，そのことによって VMB の位置付けを明らかにする．次に 3 章で，VMB の入力となる仮想機械の仕様記述の方法について述べ，4 章では，与えられた記述からインタプリタの実装を生成するためのアルゴリズムについて述べる．5 章では VMB の実装と評価について述べ，6 章では VMB の現在の問題点と今後の課題について議論する．7 章では，仮想機械の生成に関連する他の研究について，本研究との比較検討を行う．最後に 8 章で本稿をまとめる．

2. VMB のシステムの概要

次章以降で VMB の具体的な説明を行う前に，まず本章で，本研究が対象とする仮想機械について簡単な定義を与え（2.1 節），そのような仮想機械を開発する際に VMB がどのように利用されるかについて概略を述べる（2.2 節）．

2.1 仮想機械の定義

本研究が対象とする**仮想機械**とは，レジスタやスタックといった内部構造を持つ仮想的なハードウェア上で，**バイトコード命令**と呼ばれる仮想的な機械語命令を逐次的に実行することによって計算を進めていくソフトウェアとする．このような仮想機械は**図 1** のように表現できる．

バイトコードプログラムとは仮想機械によって実行されるプログラムのことであり，これはバイトコード命令の列として表現されるものとする．仮想機械は，次に実行する命令を指し示す**プログラムカウンタ**という特別なレジスタを 1 つ持ち，これを利用してバイトコード命令を読み取り，処理を進めていくものとする．

レジスタと**スタック**は，一般的な計算機で用いられるそれらと同種の構造とし，仮想機械によって計算を進めていく際に利用されるものとする．図 1 ではレジ

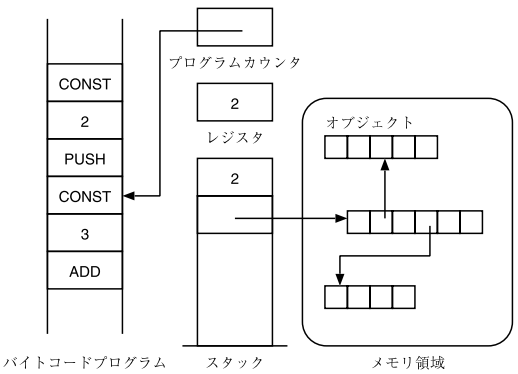

図 1 仮想機械のモデル
Fig. 1 The model of virtual machines.

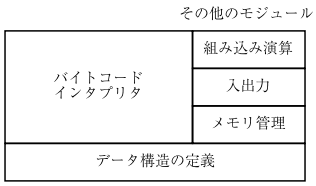

図 2 仮想機械の一般的な実装
Fig. 2 An ordinary implementation of virtual machines.

スタとスタックを 1 つずつ示してあるが，これらの本数は仮想機械の設計上の必要に応じて任意であるものとする．

本研究では，これらの構造以外のメモリ領域に格納されるデータをすべて**オブジェクト**と総称する．オブジェクトには特別な内部構造を仮定せず，配列のような単純な要素列と考える．これらのオブジェクトには，レジスタやスタックから参照をたどってアクセスするものとする．また，メモリ領域の具体的な管理方法についても特に仮定を置かず，あたかも無限の大きさのメモリ領域があるかのように扱えるものとする．

2.2 VMB を用いる処理系開発

2.1 節に定めたような仮想機械の具体的な実装は，**図 2** に示すようないくつかのモジュールからなるのが一般的である．まず，基礎となる部分として，仮想機械による操作の対象となるデータの表現形式の定義がある．その上に，仮想機械の実装の中心となる**バイトコードインタプリタ**がある．これは，プログラム中のバイトコード命令を読み込んで処理を行うものである．このほかに，組み込み演算やメモリ管理機能，入出力機能などを担う補佐的なモジュールがいくつか存在し，これらを組み合わせることによって，仮想機械というソフトウェアの全体が構築される．

VMB は，仮想機械の実装の中心部分をなすバイトコードインタプリタを生成するシステムである．処理系開発者は，本稿の 3 章で定める記述言語を用いて仮想機械の設計を形式的に記述し，それを VMB に入力として与える．VMB は与えられた記述を解析し，バイトコードインタプリタの実装を C 言語のプログラム断片の形で出力する☆．

仮想機械全体を構築するためには，VMB によって生成されたインタプリタに，その他の必要なモジュールを実装して組み合わせればよい．VMB はインタプリタ以外の各モジュールを生成しないが，開発者によるモジュール実装を容易にするため，抽象性の高いインタフェースを用いてインタプリタと他のモジュールとの独立性を保つ．また，基本的な構造の仮想機械を構築するためのモジュール集合をテンプレートとして提供することもできる．テンプレートを利用すれば，開発者はインタプリタの仕様記述のみから仮想機械を簡単に構築できる．

仮想機械を用いてバイトコードプログラムを実行するためには，原始プログラムをバイトコードプログラムに変換するコンパイラが別途必要となる．VMB はコンパイラを生成するシステムではないので，コンパイラは開発者が独自に実装する必要がある．コンパイラの実装もまた困難な作業ではあるが，コンパイラ生成系については古くからさかんに研究がなされており，現在ではさまざまなツールが広く利用できるようになっている．

3. 仮想機械の仕様記述

本章では，仮想機械の設計仕様を記述するための言語を定める．VMB は，ここで定める形式による仮想機械の記述を入力とし，次章で述べるアルゴリズムによってインタプリタの実装を生成する．

我々は，仮想機械の仕様は，その仮想機械の内部構造と各バイトコード命令の意味によって定められるものと考え，それらを簡単に記述できるように記述言語を設計した．内部構造は，仮想機械が持つレジスタやスタックを 1 つずつ宣言することで定める．バイトコード命令の意味は，命令の実行前後での仮想機械の状態を並べ記すことによって定める．すなわち，実行前後での状態の差が，その命令の意味であるものとする．我々は，これが命令の意味を表現する直観的な方法であると考えている．たとえば Java 仮想機械の各バイトコード命令の意味は，このような表現方法で定められている[5]．

以下では，本研究で定める記述言語について述べる．3.1 節では仮想機械の内部構造の記述方法について述べ，3.2 節でバイトコード命令の意味の記述方法について述べる．命令の意味は文法規則だけでは不十分な点があり，付加的な条件が必要となる．3.3 節でこのことについて述べる．

3.1 仮想機械の構造の定義

仕様記述の前半では，生成する仮想機械の構造を定義する．この記述は，基底データ型の宣言，定数と演算の宣言，仮想ハードウェアの構成の定義という 3 つの部分から成り立つ．それぞれの部分について以下で述べる．以下の説明のため，仮想機械の構造を定義する部分の記述例を**図 3** に示す．

☆ 出力言語として C 言語を選択した理由は，それが広く用いられている言語だからである．このことは，本稿において本質的なことではない．

```
%type       Bool

%const      true
%const      false
%prim       not(Bool):  Bool
%prim       and(Bool, Bool):  Bool
%prim       or(Bool, Bool):  Bool

%register   reg:  Bool
%fstack     stack:  Bool
%pc         prog:  int
```

図 3 仮想機械の構造の定義例
Fig. 3 The example of definition of virtual machine structures.

基底データ型の宣言

まずはじめに，これから定義する仮想機械が扱うデータの型を宣言する．これは，

`%type` *name*

という記述によってなされる．たとえば図 3 では，`%type Bool` という記述によって，この仮想機械が `Bool` 型のデータを扱うことを宣言している．

本仕様記述では，この記述によって宣言されたデータ型のほかに，組み込みの整数型として `int` を利用できる．`int` 型は，VMB の出力言語である C 言語での `int` を意味するものとする．

定数と演算の宣言

定数や演算の宣言は，それぞれ `%const`，`%prim` を用いて記述する．

`%const` *name* `:` *type*

は定数を宣言する記述であり，*name* を *type* 型の定数と宣言するものである．一方，

`%prim` *name* `(` *type* `,` *type* `,` `···` `)` `:` *type*

は演算を宣言する記述であり，括弧内の仮引数型と末尾の返値型を持つ演算 *name* を宣言するものである．図 3 では，`true`，`false` を `Bool` 型の定数として，`not`，`and`，`or` を `Bool` 上の演算として，それぞれ宣言している．

本仕様記述では，ここで宣言された定数や演算のほかに，`int` 型の定数として整数値を，組み込み演算として四則演算や比較演算を利用できる．これらはそれぞれ，VMB の出力言語である C 言語での対応する演算を意味するものとする．

仮想ハードウェアの構成の定義

仮想機械を構成するレジスタやスタック，プログラムカウンタといった内部構造の定義は，

`%`*kind* *name* `:` *type*

という記述によってなされる．先頭の `%`*kind* は構造の種類を示すものであり，これには **表 1** のいずれかを指定する．*name* は構造の名前を定め，*type* はその構造に格納されるデータの型を定める．図 3 では，この仮想機械を構成する内部構造として，`Bool` 値を格納するレジスタ `reg`，`Bool` 値を各要素とするメモリ順方向に伸びるスタック `stack`，および，`int` 値を各要素とするバイトコードプログラムを指し示すプログラムカウンタ `prog` を定めている．この記述によって，**図 4** に示す構造の仮想機械が定められる．

表 1 仮想機械を構成する内部構造の一覧
Table 1 The list of internal structures which compose virtual machines.

宣言方法	定義される内部構造
`%register`	一般的なレジスタ
`%fstack`	メモリの順方向に伸びるスタック
`%bstack`	メモリの逆方向に伸びるスタック
`%pc`	プログラムカウンタ

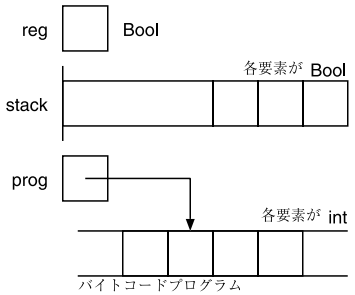

図 4 図 3 によって定められる仮想ハードウェア構成
Fig. 4 The virtual hardware structure defined by Fig. 3.

一般に，仮想機械の設計においてはメモリ領域に大域データ領域，ヒープ領域といった構造を考えるが，本仕様記述では，メモリ領域内部の構造は宣言しない．これは，本仕様記述が仮想機械全体の中のインタプリタ部分を生成するためのもの記述であり，その目的のためにはメモリ領域の内部構造を特に定める必要はないためである．

3.2 バイトコード命令の定義

仕様記述の後半では，前節での記述によって定められた仮想機械で実行されるバイトコード命令集合を定義する．命令集合の定義は，個々の命令の定義を列挙することによってなされる．

本章の冒頭で述べたように，本記述言語では，実行前後での仮想機械の状態を並べ記す方法でバイトコード命令の意味を定める．まずはじめに，簡単なバイトコード命令の記述を例にして，この記述方法をおおまかに説明する．なお，以下で例に用いる各バイトコード命令は，特に断りのない限り，図 3 で定義された仮想機械の命令であるものとする．

図 5 の `POP n` 命令は，バイトコードプログラムか

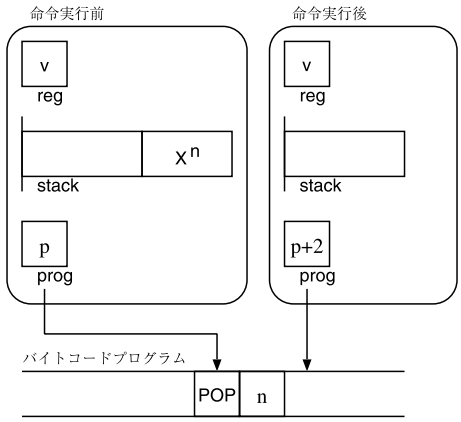

図 5 POP n 命令の意味
Fig. 5 The semantics of POP n instruction.

$$
\begin{array}{rcl}
Instr & ::= & instr\_name\ x_1\ x_2\ \cdots : Rule^{+} \\
Rule & ::= & State_{\mathrm{L}}\ \texttt{->}\ State_{\mathrm{R}}\ [\ \texttt{when}\ Expr_{\mathrm{R}}\ ] \\
State_{\mathrm{L}} & ::= & (\ (Expr_{\mathrm{L}} \mid Seq_{\mathrm{L}}^{*})^{+}\ )\ [\ \{\ Obj_{\mathrm{L}}^{*}\ \}\ ] \\
State_{\mathrm{R}} & ::= & (\ (Expr_{\mathrm{R}} \mid Seq_{\mathrm{R}}^{*})^{+}\ )\ [\ \{\ Obj_{\mathrm{R}}^{*}\ \}\ ] \\
Expr_{\mathrm{L}} & ::= & x \\
Expr_{\mathrm{R}} & ::= & x \mid c \mid f(Expr_{\mathrm{R}}^{*}) \mid l \\
Seq_{\mathrm{L}} & ::= & X^{Expr_{\mathrm{R}}} \mid \texttt{[}Expr_{\mathrm{L}}\texttt{]} \mid \texttt{@}x \mid \texttt{\#}l \\
Seq_{\mathrm{R}} & ::= & X \mid \texttt{[}Expr_{\mathrm{R}}\texttt{]} \mid \texttt{\#}l \\
Obj_{\mathrm{L}} & ::= & o : type = Seq_{\mathrm{L}}^{*} \\
Obj_{\mathrm{R}} & ::= & o \texttt{ <- } Seq_{\mathrm{R}}^{*} \mid o : type = Seq_{\mathrm{R}}^{*}
\end{array}
$$

図 6 バイトコード命令の意味記述文法
Fig. 6 The syntax for specification of the semantics of bytecode instructions.

ら引数を 1 つ読み込み，スタックの先頭からその引数分の要素を取り除く命令である．本記述言語では，この命令の意味を

```
POP n: ( v, X(n), p ) -> ( v, , p + 2 )
```

のように記述する．先頭の POP n は，ここで定義するバイトコード命令に POP という名前を付けるとともに，この命令が引数を 1 つとる命令であり，その引数の値を n で表すことを表現している．これに続く部分が POP n 命令の意味を表現する部分である．まず，矢印（->）の左側にある 3 つ組の記述によって，命令実行前のレジスタ（reg）の値を v と，スタック（stack）の先頭から n 個分の要素を X と，プログラムカウンタ（prog）の値を p と，それぞれ表す．3 つ組の順序は図 3 での定義の順番によって定められる．矢印の右側には，POP n 命令の実行後に仮想機械がどのような状態となるかを記述する．ここでは，reg の値は v のまま変化しないこと，stack から X にあたる要素が取り除かれること，および，prog の値が実行前の値から 2 だけ増加することが表現されている．

ここで，バイトコード命令の意味記述のための文法を**図 6** に示す．先頭の *Instr* 文法規則によって，1 つの命令の意味が定められる．先の POP n 命令の例で述べたように，これは命令の名前の定義，引数の記述，および，仮想機械の状態遷移規則によって構成される．ここで，$Rule^{+}$ とあるように，1 つの命令に対して複数の状態遷移規則を定めることができる．たとえば，レジスタ reg の値が true であれば n 番地先にジャンプするという条件分岐命令 BRANCH n は，本記述言語では以下のように簡単に記述できる．

```
BRANCH n:
  ( v, , p ) ->
      ( v, , p + 2 + n ) when (v == true)
  | ( v, , p ) -> ( v, , p + 2 )
```

BRANCH n 命令の最初の状態遷移規則における when 以降の部分は，その状態遷移規則が実行される条件を表す．つまり，実行前の reg の値を表す v が true であるときに限り，この規則に従う処理を実行する．

命令の実行前後の状態は，$State_{\mathrm{L}}$，$State_{\mathrm{R}}$ の文法規則に従って，それぞれ定められる．丸括弧の中には，これまでの例で示したように，レジスタやスタック，プログラムカウンタの状態をそれらが定義された順番に記述する．後半の波括弧の中には，レジスタやスタックから参照されるオブジェクトの状態を必要に応じて記述する．オブジェクトの記述については後述することとし，まず先にレジスタとスタックの状態を記述するための文法を説明する．プログラムカウンタの状態の記述方法は，レジスタのそれと同じである．以下の説明では，バイトコード命令の実行前後における仮想機械の状態のことを，それぞれ**前状態**，**後状態**と略記する．

レジスタの状態の記述

レジスタの状態は $Expr_{\mathrm{L}}$，$Expr_{\mathrm{R}}$ の文法によって定める．前状態では，識別子 x の記述によって，そのレジスタの値を x と表す．後状態には，識別子 x，定数 c，演算 $f(e_1, e_2, \cdots)$，および，参照値 l を記述できる．これらはそれぞれ，命令の実行後には，記述した値がレジスタに代入されることを意味する．たとえば，レジスタ reg の値の否定をとる命令は，図 3 で宣言された演算 not を用いて

```
NOT: ( v, , p ) -> ( not(v), , p + 1 )
```

と記述できる．

参照値 l は，オブジェクトへの参照を表すものである．この記述については，オブジェクトの記述方法とあわせて説明する．

スタックの状態の記述

スタックの状態の記述には，Seq_{L}，Seq_{R} の文法規則を利用する．これらの文法規則の右辺には $X^{Expr_{\mathrm{R}}}$，X，$[Expr_{\mathrm{L,R}}]$，`@`x，`#`l がある．これらのうち主要なものは $X^{Expr_{\mathrm{R}}}$ と X であるから，まずこの 2 つについて説明する．

$X^{Expr_{\mathrm{R}}}$ と X は，複数の要素からなる一連のデータ列をまとめて扱うための記述である．$X^{Expr_{\mathrm{R}}}$ は前状態に記述し，その位置にある長さ $Expr_{\mathrm{R}}$ の要素列を X で表すことを意味する．添字のない X は後状態に記述し，X が表す要素列がその位置に代入されることを意味する．たとえば，最初の例に示した `POP n` 命令では，前状態の記述によって `stack` の先頭から `n` 個の要素を `X` で表し，後状態の記述によって，これが `stack` から取り除かれることを表現している．なお，X の要素数を示す $Expr_{\mathrm{R}}$ は実際には `POP n` 命令に示したように丸括弧で囲む形で記述するが，本稿では見やすさのために上付き文字で表記する．

スタックの状態は，$Seq_{\mathrm{L,R}}$ の要素を複数個並べることで簡単に記述できる．例として，`stack` の先頭から `n` 番目に格納されているデータを取り除いて `reg` に代入する `GETSTACK n` 命令を考える．この命令の実行による仮想機械の状態変化を図示すると**図 7** のようになる．この命令は以下のように記述できる．

```
GETSTACK n:
    ( x, [y].S^n, p ) -> ( y, S, p + 2 )
```

前状態の `stack` 部分の記述では，`stack` の先頭から `n` 要素をまとめて `S` と置き，その次の要素を `y` と置いている．`[y]` というのは，$Seq_{\mathrm{L,R}}$ の文法規則の右辺にある $[Expr_{\mathrm{L,R}}]$ の規則を用いた表現である．これは，スタックの一要素を表す記述形式である．

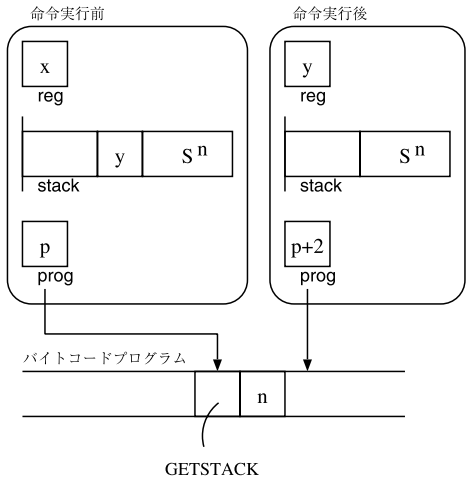

図 7 `GETSTACK n` 命令による仮想機械の状態変化
Fig. 7 The state transition in the virtual machine by the execution of `GETSTACK n` instruction.

この記述のように $Seq_{\mathrm{L,R}}$ を複数並べて記述したときには，右側がメモリの順方向を表すものとする．上の例における `stack` は仮想ハードウェアの定義の際に `%fstack` と宣言されていたので，$\mathtt{S}^{\mathtt{n}}$ がスタックの先頭を意味することになる．もし `stack` が `%bstack` と宣言されていたならば，上の記述は `stack` の先頭要素を `y` とし，続く `n` 要素を `S` とするという意味になり，単に `stack` の先頭要素を取り除いて `reg` に代入するという意味を表すことになる．

$Seq_{\mathrm{L,R}}$ の文法のうち，`@`x と `#`l はオブジェクトの参照を表現するために用いるものである．これらは，次のオブジェクトの記述の中で述べる．

オブジェクトの状態の記述

メモリ領域に存在するオブジェクトの状態は，Obj_{L}，Obj_{R} の文法規則によって表現する．前状態の記述 o : $type$ = Seq^{*}_{L} は，$type$ 型の要素列 Seq^{*}_{L} を持つオブジェクトを o と表すことを意味する．後状態の記述 o <- Seq^{*}_{R} は，オブジェクト o の各要素が Seq^{*}_{R} と変化することを表す．また，後状態における o : $type$ = Seq^{*}_{R} の記述は，要素型 $type$ を持つ新たなオブジェクト o を生成し，Seq^{*}_{R} で初期化することを表現する．

オブジェクトの状態の記述例として，メモリ領域に存在するオブジェクトに値を代入する `SETFIELD n` という命令の記述を考える．この命令の意味を**図 8** に示す．すなわち，`SETFIELD n` 命令は，`stack` の先頭要素から参照されているオブジェクトの `n` 番目のフィールドに，実行前の `reg` の値を代入する命令とする☆．

本仕様記述では，この命令の意味を以下のように記述できる．

```
SETFIELD n:
  (v,[ptr],p) { obj:Bool = @ptr.A^n.[x] }
    -> ( v, [ptr], next ) { obj <- A.[v] }
```

前状態の `@ptr` はオブジェクトの参照を定めるものであり，スタックの先頭の値である `ptr` がその位置への参照であるということを示す．したがってこの記述は，スタックの先頭から指されているオブジェクトの，指されている位置から `n` 要素を `A` とおき，その次の要素を `x` とおくことを表現している．左辺の `obj:Bool` は，このオブジェクトの名前を `obj` とすること，このオブジェクトの各要素が `Bool` 型の値であることを

☆ `stack` の要素型は `Bool` であるから，これがオブジェクトへの参照であるというのは型が合わないが，ここでの説明には影響しないので，このことは無視する．

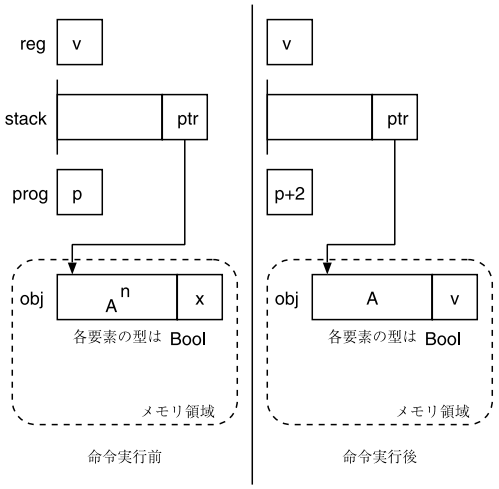

図 8 SETFIELD n 命令による仮想機械の状態変化
Fig. 8 The state transition in the virtual machine by the execution of SET n instruction.

表す．後状態の obj <- A.[v] によって，前状態で定めた obj への代入を表現する．

#*l* は，オブジェクトへの参照を取得する記述である．たとえば，

```
OFFSET n:
  (v,[ptr],p) { obj:Bool=@ptr.A^n.#l }
   -> ( v, [l], next )
```

という命令記述は，#l によって A の次の位置への参照を l と置き，それを実行後の stack の先頭に代入する．

3.3 バイトコード命令の記述における制約

前節に述べたバイトコード命令の意味記述では，文法に従った記述であっても，命令の定義として意味が定まらない場合がある．ここでは，命令の定義における意味的な制約について述べる．4 章に述べるインタプリタ生成アルゴリズムでは，ここに述べる制約を満たす記述のみを対象とする．

命令の定義において満たさなければならない制約は，実行前状態で定義されるスタックやオブジェクトの各要素が，すべて一意に定められるというものである．たとえば，以下のような命令記述はこの制約を満たさない（仮想ハードウェアは図 3 のものとする）．

```
X1:( v, [y].x^y, p ) -> ···
X2:(v, , p){o:Bool = @y.[y]} -> ···
```

X1 では，X の長さを定めるために y が必要となり，その一方で y の場所を定めるためには X の長さが必要となる．X2 では，o の先頭要素 y によって参照されるオブジェクトとして *o* 自身を定義している．いずれの命令も y を定められないため，前状態として意味をなさない記述である．

命令の記述がこの制約を満たしているかどうかを判断する判定式はここでは省略するが，これは，レジスタやスタックから再帰的に調べていくことで容易に判定可能である．

```
while (true) {
  switch (次の命令) {
    case 命令 1: {
      状態変化 1 の前提条件を調べる
      if (前提条件が成立) {
        状態変化 1 を実行; break; }
      状態変化 2 の前提条件を調べる
      if (前提条件が成立) {
        状態変化 2 を実行; break; }
      ···
    }
    case 命令 2: {
      ···
    }
    ···
  }
}
```

図 9 VMB が生成するインタプリタの構造
Fig. 9 The structure of an interpreter generated by VMB.

4. インタプリタ生成アルゴリズム

VMB は，前章で述べた形式の仕様記述を解析し，本章で述べるアルゴリズムに従って**図 9** のような典型的な構造のインタプリタを生成する．インタプリタ全体は，図のような巨大な switch 文から構成される☆．この switch 文によって 1 命令分の処理を実行し，これを外側の while ループによってプログラムの実行が終了するまで繰り返す．仕様記述では，バイトコード命令はそれぞれが前提条件を持つ複数の状態変化規則として表現されているが，記述の順に前提条件の成立を調べ，成立したものを実行する形で実現できる．

本章では，1 つの命令中の，1 つの状態変化規則に対応するコード断片を生成するアルゴリズムについて述べる．本章のアルゴリズムによって各状態変化規則に対応するコード断片を生成できれば，それらを複合してインタプリタ全体のコードを得るのは，たやすい作業である．

VMB のインタプリタ生成アルゴリズムは，

- 状態の記述をより単純な正規形に変換する，
- 前状態と後状態の差分から，必要な操作を求める，
- 求められた操作を適切な順序に並べる，

☆ 簡単のため switch 文として説明しているが，もちろん threaded code によるインタプリタを生成することも容易である．

- 各操作を実現する命令列を求める，
- データ型に応じて適切な型変換操作を追加する，
- コード断片を出力する，

という 6 つの段階からなる．以下の各節では，それぞれの段階で行われる処理について述べる．以下の説明では，前章で図 7 に示した `GETSTACK n` 命令を例として用いる．

4.1 正規形への変換

インタプリタ生成の最初の段階に行う処理は，命令の記述におけるスタックやバイトコードプログラムの表現を，一般的なオブジェクトの表現に変換することである．このように変換しておくことで，仮想機械のすべての状態がレジスタとオブジェクトという表現形式のみで定められることになり，後続の各段階の処理を単純化できる．

スタックというデータ構造は一般に，スタックポインタと呼ばれるレジスタと，そこから参照されるデータ列として実現でき，本研究の仕様記述でも，通常のレジスタとオブジェクトからなる組としてスタックを表現することが可能である．つまり，`%fstack` と宣言されたスタックが命令の実行によって $S = S_1.S_2.\cdots.S_i$ から $T = T_1.T_2.\cdots.T_j$ と変化するとき，これはレジスタとオブジェクトの組を用いて，

`(sp){o:`*type*`=`S`.@sp} -> (sp')){o <-` T`.#sp'}`

と表現しても同じである．`%bstack` の場合には，`@sp` と `#sp'` をそれぞれ S，T の先頭に付ければよい．また，バイトコード命令の引数についても，同様の方法でオブジェクトの表現に変換できる．

この変換によって，例題の `GETSTACK` 命令の状態変化規則の部分は以下のような記述に変換される．

```
(x, s, p) {
    stack: t = [y].S^n.@s,
    prog: t' = @p.I^1.[n] } ->
(y, s', p') {
    stack <- S.#s',
    prog <- I.[n].#p' }
```

ところが，単純にこの変換を施すと，スタックの長さが命令実行前後で変化する場合に左右のどちらが基準であるかが分からなくなる．たとえばスタックが `[x].[y] -> [x]` と変化するとき，このスタックが `%fstack` であるか `%bstack` であるかで命令の意味が異なるが，単純にレジスタとオブジェクトとして表現し直してしまうと，この情報が失われてしまう．

VMB では，この変換を行う際に，オブジェクトの変化の基準となる位置を情報として加えることで問題を解決した．各オブジェクトに対して，それが `%bstack` から変換されたオブジェクトであるならば右端を基準とし，それ以外のオブジェクトであれば左端を基準とする．

これ以降の各段階のアルゴリズムは，すべてこの情報を考慮して作られているが，本稿では説明を簡単にするためオブジェクトの先頭が基準であるものとして（つまり，スタックはすべて `%fstack` であるものとして），アルゴリズムを解説する．

4.2 操作集合の抽出

状態記述をレジスタとオブジェクトの表現に正規化した次の段階として，前状態と後状態の記述を比較して状態間の差分を調べ，仮想機械の状態を変更するために必要な操作を求める．

まず準備として，仮想機械の記憶空間における**位置**を表す概念を導入する．これは，計算機のアドレスに対応する概念である．すでに前段階で正規化がなされているので，すべてのデータはレジスタかオブジェクトのどちらかに存在すると考えてよい．これらをそれぞれ以下の記法で表す．

- R_r：名前 r のレジスタを表す．
- $\mathrm{O}_{o[e]}$：名前 o のオブジェクトの，先頭から e 番目の要素を表す．

これを用いて，バイトコード命令の実行による状態変化を以下の 3 種類の操作からなる集合として表す．ただし，l_c は，上で定義した位置を表すものとする．

- $\mathcal{A}[l_c \leftarrow e]$：$e$ を計算して，結果を l_c の位置に**代入**（$\mathcal{A}ssign$）する．
- $\mathcal{C}[l_c \leftarrow X]$：$X$ を l_c を左端とする位置に**複写**（$\mathcal{C}opy$）する．
- $\mathcal{N}[o(e)]$：e を計算して得られる長さのオブジェクトを新規（$\mathcal{N}ew$）に**生成**する．このオブジェクトの名前を o とする．

これらの操作は，命令の後状態を構成する各要素を調べることで簡単に求まる．たとえば `GETSTACK` 命令の記述から求まる操作集合は以下の 4 つの要素からなる．

$$\begin{aligned} \mathcal{A}[\mathrm{R}_{\mathtt{reg}} \leftarrow \mathtt{y}] \quad & (= A_{\mathtt{r}}) \\ \mathcal{A}[\mathrm{R}_{\mathtt{sp}} \leftarrow \mathtt{s'}] \quad & (= A_{\mathtt{s}}) \\ \mathcal{A}[\mathrm{R}_{\mathtt{pp}} \leftarrow \mathtt{p'}] \quad & (= A_{\mathtt{p}}) \\ \mathcal{C}[\mathrm{O}_{\mathtt{stack}}[0] \leftarrow \mathtt{S}] \quad & (= C_{\mathtt{s}}) \end{aligned}$$

ただし，`reg`，`sp`，`pp` をそれぞれレジスタ，スタックポインタ，プログラムポインタに対応するレジスタの名前とした．また，次節以降での説明のため，それぞれの操作を括弧内のように名付ける．

4.3 直 列 化

次に，前節で求めた各操作について互いの依存関係を

解析し，必要に応じて一時変数を導入しながら，各操作を正しく順序付ける処理を行う．たとえば，GETSTACK 命令から求まった 4 つの操作では，もし最初に $C_{\mathtt{s}}$ を実行してしまうと y の値が破壊され，$A_{\mathtt{r}}$ を正しく実行することができない．以下では，このような依存関係を解析し，正しく順序付けるアルゴリズムについて述べる．

生成操作 $\mathcal{N}$ を実行する

まず最初に，操作 $\mathcal{N}$ をすべて実行する．$\mathcal{N}$ は新しいオブジェクトの領域を確保する操作であり，実行前状態の値を何も破壊しないので，最初に実行してしまっても問題を生じない．また，$\mathcal{N}$ を最初に実行することには，解析対象となる集合を小さくできること，代入対象のオブジェクトが生成済みであるかを解析に含める必要がなくなり，解析を簡単にできることという利点もある．操作 $\mathcal{N}$ が複数ある場合の実行順序は自由に決めることができる．

各操作が使用する値と破壊する値を求める

操作 $\mathcal{A}$ と $\mathcal{C}$ からなる集合の各要素に適切な順序付けを行うため，まず，各操作が使用する値の集合と破壊する値の集合を求める．以降の説明では，操作 O が使用する値の集合を $\mathrm{U}(O)$，破壊する値の集合を $\mathrm{D}(O)$ とそれぞれ表記する．

操作 O が使用する値の集合 $\mathrm{U}(O)$ は，O の右辺を計算するために必要な値をすべて求めることで計算できる．$\mathrm{U}(O)$ を求める方法の説明として，操作が $\mathcal{A}[l_c \leftarrow x]$ である場合について述べる．この場合の手順は以下のとおりである．

- 右辺の識別子 x を $\mathrm{U}(O)$ に加える．
- x の定義された位置を調べる．それがオブジェクト $\mathrm{O}_{o[e]}$ ならば，x に到達するためには，$o : t = \cdots \mathtt{@}y \cdots$ なる y，および，$\mathtt{@}y$ の位置を $\mathrm{O}_{o[e']}$ として $e - e'$ を求めるために必要なすべての値を使用する．これらを $\mathrm{U}(O)$ に加える．
- 上の操作を，新たに加えられる要素がなくなるまで繰り返す．

例として，GETSTACK 命令から求まった代入操作 $A_{\mathtt{r}}$ について $\mathrm{U}(A_{\mathtt{r}})$ を求める．$A_{\mathtt{r}}$ の右辺は y であるら，まず $\mathrm{U}(A_{\mathtt{r}}) = \{\mathtt{y}\}$ とする．y が定義されている位置は $\mathrm{O}_{\mathtt{stack}}[0]$ であるから，y の値を得るためには，まず stack オブジェクトに到達するために s が必要である．さらに，@s の位置は $\mathrm{O}_{\mathtt{stack}}[\mathtt{n+1}]$ であるから，@s から y に到達するために 0 - (n+1) を計算する必要がある．このため n も必要となる．同様にして，n は prog オブジェクトで定義されているので p も必要となる．@p から n に到達する式は 1 - 0 であるから，ここで必要となる値はない．以上で計算は止まり，$\mathrm{U}(A_{\mathtt{r}})$ は $\{\mathtt{y}, \mathtt{s}, \mathtt{n}, \mathtt{p}\}$ と求まる．

他の種類の操作の場合も，手順はこれとほぼ同じである．右辺が演算であれば，引数に現れるすべての識別子について上の手順を実行し，全体の和集合をとる．右辺が参照 l の場合は $\mathtt{\#}l$ の位置について上の手順を用いる．右辺が列 X の場合は，上の手順に加えて X の長さについても求める．

一方，操作 O が破壊する値の集合 $\mathrm{D}(O)$ は，O の左辺と重なる可能性のある識別子の集合を考えることで求められる．操作が $\mathcal{A}$ の場合について $\mathrm{D}(O)$ を求める手順を述べる．以下の説明で，loc は引数に与えられた識別子が定義されている位置を返す関数とする．

- 操作 O の左辺 l_c がレジスタ R_r ならば，$\mathrm{loc}(x) = l_c$ となる x がただ 1 つ存在する．これを $\mathrm{D}(O)$ の唯一の要素とする．
- l_c がオブジェクト $\mathrm{O}_{o[e]}$ ならば，$\mathrm{loc}(x) = \mathrm{O}_{o[e']}$ であるすべての x について，$e \neq e'$ がつねに成り立つかどうかを調べる．この不等式が恒真でない x を $\mathrm{D}(O)$ の要素とする．X についても同様に調べるが，X の場合は長さを持つので，判定式は (X の右端 $< e \vee e <$ X の左端) となる．これが恒真でない X を $\mathrm{D}(O)$ に加える．

操作が $\mathcal{C}[l_c \leftarrow X]$ の場合は，複写先の位置が l_c から l_c に X の長さを加えた位置までの幅を持つものとして，上と同様にして重なる可能性を考えればよい．

ここまでの説明に対する例として，GETSTACK 命令で求められた 4 つの操作の使用集合 U と破壊集合 D とを**表 2** にまとめる．

操作の順序付け

各操作に対してこれらの集合を求めた後，以下の方法で操作の実行順序を決定する．

- 操作 O の $\mathrm{D}(O)$ が，他のどの操作 O' の $\mathrm{U}(O')$ とも共通部分を持たないならば，O をここで実行して何の問題も生じない．O を実行することとし，これを操作集合から取り除く．
- そのような O が存在しなければ，操作集合から任意の 1 つを選び，その操作を退避する．

表 2 GETSTACK 命令における各操作の使用集合と破壊集合
Table 2 Use set and destroy set of each action in GETSTACK instruction.

操作	使用集合	破壊集合
$A_{\mathtt{r}}$	{y, s, n, p}	{x}
$A_{\mathtt{s}}$	{s}	{s}
$A_{\mathtt{p}}$	{p}	{p}
$C_{\mathtt{s}}$	{S, s, n, p}	{y, S}

ここで，操作 O の退避とは，O の右辺を計算した結果を一時的な領域に格納することとする．操作 O を退避することで，（その時点で O の右辺の計算は終了しているため）$\mathrm{U}(O)$ は空集合となり，順序付けの処理を先に進めることができる．この処理で使われる一時的領域を表現するため，前節で導入した位置の概念に以下の 2 つの分類を追加する．

- $\mathrm{R}^{\mathrm{T}}_r$：名前 r の一時レジスタを表す．
- $\mathrm{O}^{\mathrm{T}}_{o[e]}$：名前 o の一時オブジェクトの，先頭から e 番目の要素を表す．

これを用いて，操作 O を退避する処理が定められる．操作が $\mathcal{A}[l_c \leftarrow e]$ の場合について示す．

- 新たな一時レジスタ r を導入して，左辺を $\mathrm{R}^{\mathrm{T}}_r$ に置き換えた操作を実行する．
- 操作集合から O を取り除き，$\mathcal{A}[l_c \leftarrow x]$ を新たに加える．x は新たな識別子であり，先に代入した一時レジスタ r の値を表すものとする．

同様に，操作 $\mathcal{C}[l_c \leftarrow X]$ を退避するには，

- 新たな一時オブジェクト o を導入して，操作 $\mathcal{N}[o(e)]$ を実行する．ここで e は X の長さとする．
- 元の操作の左辺を $\mathrm{O}^{\mathrm{T}}_{o[0]}$ に置き換えた操作を実行する．
- O を操作集合から取り除き，$\mathcal{C}[l_c \leftarrow Y]$ を新たに加える．Y は新たな識別子であり，先に代入した一時オブジェクトの先頭から e 要素分の列を表すものとする．

これらの手順に従って，実行する順に操作を取り除いていき，操作集合が空になるまで続けることで操作の順序けがなされる．

例として，GETSTACK の操作集合の直列化を行う．表 2 より，最初に実行できる操作は $A_{\mathtt{r}}$ のみであるから，まずこれを実行することとして操作集合から取り除く．$A_{\mathtt{r}}$ が取り除かれたことによって，$C_{\mathtt{s}}$ が実行可能になる．$C_{\mathtt{s}}$ も取り除かれると，残った $A_{\mathtt{s}}$ と $A_{\mathtt{p}}$ は任意の順番で実行可能である．結果として，GETSTACK の 4 つの操作はいずれも退避の必要なく実行することが可能であった．

4.4 原始的操作列の生成

直列化の次に続く段階として，求められた操作列に含まれる各操作を，実際の計算機命令に近い形式である**原始的操作**の列に変換する．ここで求められる原始的操作列とは**表 3** に示す各操作からなる列である．

それぞれの原始的操作は，計算機の機械語命令に近い単純なものであるから，各バイトコード命令を原始的操作列として表現することができれば，そこから実際のプログラミング言語のコードを生成することは容易である．表 3 の最初にあるのは，右辺の定数値を読み込み左辺の一時レジスタに代入するものである．次の 4 つは，仮想レジスタやオブジェクトの要素の読み書きを行う操作である．それぞれ，右辺の値を左辺に代入することを意味する．$\mathrm{R}^{\mathrm{T}}_{\mathrm{d}} \leftarrow \&(\mathrm{R}^{\mathrm{T}}_{\mathrm{o}}[\mathrm{R}^{\mathrm{T}}_{\mathrm{i}}])$ は，右辺が示す要素への参照を左辺に代入する操作である．$\mathrm{R}^{\mathrm{T}}_{\mathrm{d}} \leftarrow \mathrm{new}_{\mathtt{o}}(\mathrm{R}^{\mathrm{T}}_{\mathrm{n}})$ は新規にオブジェクトを生成するための領域を確保する操作であり，$\mathrm{R}^{\mathrm{T}}_{\mathrm{n}}$ の大きさの領域を確保して，先頭要素への参照を $\mathrm{R}^{\mathrm{T}}_{\mathrm{d}}$ に代入する．添字の o はオブジェクトの名前を表す．$\mathrm{R}^{\mathrm{T}}_{\mathrm{d}} \leftarrow f(\mathrm{R}^{\mathrm{T}}_1, \cdots, \mathrm{R}^{\mathrm{T}}_{\mathrm{n}})$ は関数適用である．最後の $\mathrm{for}(\mathrm{R}^{\mathrm{T}}_{\mathrm{i}} < \mathrm{R}^{\mathrm{T}}_{\mathrm{e}}, I^*)$ は列の複写のために用意されている操作であり，$0 \le \mathrm{R}^{\mathrm{T}}_{\mathrm{i}} < \mathrm{R}^{\mathrm{T}}_{\mathrm{e}}$ の範囲の $\mathrm{R}^{\mathrm{T}}_{\mathrm{i}}$ に対して内部の原始的操作列 I^* を実行する．

表 3 原始的操作の一覧
Table 3 The list of primitive actions.

原始的操作の形式	原始的操作の意味
$\mathrm{R}^{\mathrm{T}}_{\mathrm{d}} \leftarrow c$	定数値の取得
$\mathrm{R}^{\mathrm{T}}_{\mathrm{d}} \leftarrow \mathrm{R}_{\mathtt{s}}$	レジスタ値の取得
$\mathrm{R}_{\mathtt{d}} \leftarrow \mathrm{R}^{\mathrm{T}}_{\mathrm{s}}$	レジスタへの代入
$\mathrm{R}^{\mathrm{T}}_{\mathrm{d}} \leftarrow \mathrm{R}^{\mathrm{T}}_{\mathrm{o}}[\mathrm{R}^{\mathrm{T}}_{\mathrm{i}}]$	オブジェクトの要素値の取得
$\mathrm{R}^{\mathrm{T}}_{\mathrm{o}}[\mathrm{R}^{\mathrm{T}}_{\mathrm{i}}] \leftarrow \mathrm{R}^{\mathrm{T}}_{\mathrm{s}}$	オブジェクトの要素への代入
$\mathrm{R}^{\mathrm{T}}_{\mathrm{d}} \leftarrow \&(\mathrm{R}^{\mathrm{T}}_{\mathrm{o}}[\mathrm{R}^{\mathrm{T}}_{\mathrm{i}}])$	オブジェクトの要素の参照の取得
$\mathrm{R}^{\mathrm{T}}_{\mathrm{d}} \leftarrow \mathrm{new}_{\mathtt{o}}(\mathrm{R}^{\mathrm{T}}_{\mathrm{n}})$	新規オブジェクトの領域確保
$\mathrm{R}^{\mathrm{T}}_{\mathrm{d}} \leftarrow f(\mathrm{R}^{\mathrm{T}}_1, \cdots, \mathrm{R}^{\mathrm{T}}_{\mathrm{n}})$	関数適用
$\mathrm{for}(\mathrm{R}^{\mathrm{T}}_{\mathrm{i}} < \mathrm{R}^{\mathrm{T}}_{\mathrm{e}}, I^*)$	繰返し処理

原始的操作列への変換は，代入（$\mathcal{A}$），複写（$\mathcal{C}$），生成（$\mathcal{N}$）で表されている各操作に対して，次の 2 つの処理を行うことで実現できる．

- 操作の表現の中に残存する式形式や列形式を，位置を用いた表現に変換する．
- オブジェクトの位置表現 $\mathrm{O}_{\mathtt{o}}[\mathtt{e}]$ を，仮想レジスタから参照をたどってその位置に到達するような原始的操作列に具体化する．

これらの処理は，前節で使用集合 U を求めた方法とほぼ同じようにして実現できる．

たとえば，GETSTACK 命令での操作 $A_{\mathtt{r}}$ を実現する原始的操作列は以下のようになる．

$$\mathrm{R}^{\mathrm{T}}_2 \leftarrow \mathrm{R}_{\mathtt{sp}}$$
$$\mathrm{R}^{\mathrm{T}}_7 \leftarrow \mathrm{R}_{\mathtt{pp}}$$
$$\mathrm{R}^{\mathrm{T}}_8 \leftarrow \mathtt{1}$$
$$\mathrm{R}^{\mathrm{T}}_5 \leftarrow \mathrm{R}^{\mathrm{T}}_7[\mathrm{R}^{\mathrm{T}}_8]$$
$$\mathrm{R}^{\mathrm{T}}_6 \leftarrow \mathtt{1}$$
$$\mathrm{R}^{\mathrm{T}}_4 \leftarrow \mathrm{R}^{\mathrm{T}}_5 \mathtt{+} \mathrm{R}^{\mathrm{T}}_6$$
$$\mathrm{R}^{\mathrm{T}}_3 \leftarrow \mathtt{-}\, \mathrm{R}^{\mathrm{T}}_4$$
$$\mathrm{R}^{\mathrm{T}}_1 \leftarrow \mathrm{R}^{\mathrm{T}}_2[\mathrm{R}^{\mathrm{T}}_3]$$

$\mathrm{R}_{\texttt{reg}} \leftarrow \mathrm{R}_1^{\mathrm{T}}$

この操作はレジスタ reg への代入であるから，操作を実現するための最後の原始的操作は $\mathrm{R}_{\texttt{reg}} \leftarrow \mathrm{R}_1^{\mathrm{T}}$ の形となる．右辺の $\mathrm{R}_1^{\mathrm{T}}$ には $\mathrm{O}_{\texttt{stack}}[0]$ の値である y が格納されていなければならないので，$\mathrm{R}_1^{\mathrm{T}} \leftarrow \mathrm{R}_2^{\mathrm{T}}[\mathrm{R}_3^{\mathrm{T}}]$ の形の原始的操作によってオブジェクトの要素を取得しておく必要がある．さらにこれに先だって，$\mathrm{R}_2^{\mathrm{T}}$ にはスタックポインタである $\mathrm{R}_{\texttt{sp}}$ を，$\mathrm{R}_3^{\mathrm{T}}$ には，-(n+1) の計算結果をそれぞれ格納しておく必要がある．このように，U を求める方法と同様にしてオブジェクトをたどっていくことで，原始的操作の列が求められる．

4.5 原始的操作列への型付け

解析の最後の段階として，求まった原始的操作列に対する型付けを行う．前節で述べた原始的操作には型情報が含まれていないため，このままでは正しいコードを生成することができない．ここでは，原始的操作列を解析してコード生成のために必要な型情報を求める処理について述べる．

原始的操作列に対する型付けは，仕様記述に宣言された型をもとに一時レジスタの型を順に定めていき，矛盾を生じた時点で必要な型変換操作を挿入することによって実現する．

GETSTACK 命令の操作 $A_{\texttt{r}}$ から前節で求まった原始的操作列を例として，本節での解析の概要を述べる．ここでは，説明のために，それぞれのレジスタの型は reg:int，sp:Bool*，pp:int* であるものとする．型の後のアスタリスクは，その型への参照型を表すものとする．このとき，定められた型情報は

$$\begin{aligned}
\mathrm{R}_2^{\mathrm{T}} &\leftarrow \mathrm{R}_{\texttt{sp:Bool*}} \\
\mathrm{R}_7^{\mathrm{T}} &\leftarrow \mathrm{R}_{\texttt{pp:int*}} \\
\mathrm{R}_8^{\mathrm{T}} &\leftarrow \texttt{1:int} \\
\mathrm{R}_5^{\mathrm{T}} &\leftarrow \mathrm{R}_7^{\mathrm{T}}[\mathrm{R}_8^{\mathrm{T}}] \\
\mathrm{R}_6^{\mathrm{T}} &\leftarrow \texttt{1:int} \\
\mathrm{R}_4^{\mathrm{T}} &\leftarrow \mathrm{R}_5^{\mathrm{T}}\ (\texttt{+}{:}\forall t_1, t_2.t_1 \times t_2 \to t_1)\ \mathrm{R}_6^{\mathrm{T}} \\
\mathrm{R}_3^{\mathrm{T}} &\leftarrow (\texttt{-}{:}\forall t_1 \to t_1)\ \mathrm{R}_4^{\mathrm{T}} \\
\mathrm{R}_1^{\mathrm{T}} &\leftarrow \mathrm{R}_2^{\mathrm{T}}[\mathrm{R}_3^{\mathrm{T}}] \\
\mathrm{R}_{\texttt{reg:int}} &\leftarrow \mathrm{R}_1^{\mathrm{T}}
\end{aligned}$$

のようになる．これに従って一時レジスタの型を定めていくと，最終的に $\mathrm{R}_1^{\mathrm{T}}$ の型が Bool となり，最後の $\mathrm{R}_{\texttt{reg}}$ への代入部分でデータ型を変換する必要が生じる．

ここで，一時レジスタの値の型を変換する処理を原始的操作に加える．この操作を

$$\mathrm{R}_{\mathrm{d}}^{\mathrm{T}} \leftarrow \mathcal{T}[t_d \leftarrow t_s](\mathrm{R}_{\mathrm{s}}^{\mathrm{T}})$$

と記述することにする．ここで $\mathcal{T}[t_d \leftarrow t_s]$ は，t_s から t_d への型変換を実現する関数とする．上の例では，

表 4 型付け規則
Table 4 Typing rules.

原始的操作	制約	定義
$\mathrm{R}_{\mathrm{d}}^{\mathrm{T}} \leftarrow c$		$\mathrm{t}(\mathrm{R}_{\mathrm{d}}^{\mathrm{T}}) = \mathrm{t}(c)$
$\mathrm{R}_{\mathrm{d}}^{\mathrm{T}} \leftarrow \mathrm{R}_{\mathtt{s}}$		$\mathrm{t}(\mathrm{R}_{\mathrm{d}}^{\mathrm{T}}) = \mathrm{t}(\mathrm{R}_{\mathtt{s}})$
$\mathrm{R}_{\mathtt{d}} \leftarrow \mathrm{R}_{\mathrm{s}}^{\mathrm{T}}$	$\mathrm{t}(\mathrm{R}_{\mathrm{s}}^{\mathrm{T}}) = \mathrm{t}(\mathrm{R}_{\mathtt{d}})$	
$\mathrm{R}_{\mathrm{d}}^{\mathrm{T}} \leftarrow \mathrm{R}_{\mathrm{o}}^{\mathrm{T}}[\mathrm{R}_{\mathrm{i}}^{\mathrm{T}}]$	$\mathrm{t}(\mathrm{R}_{\mathrm{o}}^{\mathrm{T}}) = \mathrm{t}(\mathtt{o})*$ $\mathrm{t}(\mathrm{R}_{\mathrm{i}}^{\mathrm{T}}) = \mathtt{int}$	$\mathrm{t}(\mathrm{R}_{\mathrm{d}}^{\mathrm{T}}) = \mathrm{t}(\mathtt{o})$
$\mathrm{R}_{\mathrm{o}}^{\mathrm{T}}[\mathrm{R}_{\mathrm{i}}^{\mathrm{T}}] \leftarrow \mathrm{R}_{\mathrm{s}}^{\mathrm{T}}$	$\mathrm{t}(\mathrm{R}_{\mathrm{o}}^{\mathrm{T}}) = \mathrm{t}(\mathtt{o})*$ $\mathrm{t}(\mathrm{R}_{\mathrm{i}}^{\mathrm{T}}) = \mathtt{int}$ $\mathrm{t}(\mathrm{R}_{\mathrm{s}}^{\mathrm{T}}) = \mathrm{t}(\mathtt{o})$	
$\mathrm{R}_{\mathrm{d}}^{\mathrm{T}} \leftarrow \&(\mathrm{R}_{\mathrm{o}}^{\mathrm{T}}[\mathrm{R}_{\mathrm{i}}^{\mathrm{T}}])$	$\mathrm{t}(\mathrm{R}_{\mathrm{o}}^{\mathrm{T}}) = \mathrm{t}(\mathtt{o})*$ $\mathrm{t}(\mathrm{R}_{\mathrm{i}}^{\mathrm{T}}) = \mathtt{int}$	$\mathrm{t}(\mathrm{R}_{\mathrm{d}}^{\mathrm{T}}) = \mathrm{t}(\mathtt{o})*$
$\mathrm{R}_{\mathrm{d}}^{\mathrm{T}} \leftarrow \mathrm{new}_{\mathrm{o}}(\mathrm{R}_{\mathrm{n}}^{\mathrm{T}})$	$\mathrm{t}(\mathrm{R}_{\mathrm{n}}^{\mathrm{T}}) = \mathtt{int}$	$\mathrm{t}(\mathrm{R}_{\mathrm{d}}^{\mathrm{T}}) = \mathrm{t}(\mathtt{o})*$
$\mathrm{R}_{\mathrm{d}}^{\mathrm{T}} \leftarrow f(\mathrm{R}_0^{\mathrm{T}}, \cdots)$	$\mathrm{t}(\mathrm{R}_{\mathrm{i}}^{\mathrm{T}}) = \mathrm{t}(f_i^a)$	$\mathrm{t}(\mathrm{R}_{\mathrm{d}}^{\mathrm{T}}) = \mathrm{t}(f^r)$
$\mathrm{for}(\mathrm{R}_{\mathrm{i}}^{\mathrm{T}} < \mathrm{R}_{\mathrm{e}}^{\mathrm{T}}, I^*)$	$\mathrm{t}(\mathrm{R}_{\mathrm{e}}^{\mathrm{T}}) = \mathtt{int}$	$\mathrm{t}(\mathrm{R}_{\mathrm{i}}^{\mathrm{T}}) = \mathtt{int}$

関数適用の項では，f の型を $f_0^a \times \cdots \times f_n^a \to f^r$ とする．

最後の原始的操作である $\mathrm{R}_{\texttt{reg}} \leftarrow \mathrm{R}_1^{\mathrm{T}}$ を

$$\begin{aligned}
\mathrm{R}_9^{\mathrm{T}} &\leftarrow \mathcal{T}[\texttt{int} \leftarrow \texttt{Bool}](\mathrm{R}_1^{\mathrm{T}}) \\
\mathrm{R}_{\texttt{reg}} &\leftarrow \mathrm{R}_9^{\mathrm{T}}
\end{aligned}$$

のように分割し，ここで型変換を行うことを明示化する．

原始的操作への型付けは，**表 4** の規則に従って原始的操作列を上から順番に調べていくことで実現できる．表中の関数 t は引数に与えられたレジスタの型を返す関数とする．型付けを行うには，各々の操作に対して，制約の項目に示されている条件を満たしているかどうかを調べる．制約が満たされていなければ，それを満たすように型変換操作を挿入する．また，一時変数への代入が発生する操作では，定義の項目に従って一時変数の型を定める．

ただし，表 4 の規則のうち，関数適用の引数については型変換操作を挿入することができない．なぜならば，本仕様記述では引数の型が異なる同名関数の定義を許しているため，変換が一意に定まらないからである．関数適用の引数の型が合わない場合には VMB はバイトコード命令を生成することができない．

4.6 コード生成

以上の解析を通して求まった原子的操作列を，C 言語のコードの形式にして出力する．原子的操作列は，列の複写に用いられる for 形式を除けば簡単な代入命令であり，それらについてのコード生成は単純な作業である．

列の複写については，複写元と複写先が同一オブジェクトであるときには，両者の位置関係に応じて左右のどちらから複写を行うかを決めなければならない．代入先が元の位置よりも左側であれば左から順に，元の位置よりも右側であれば右から順に複写する．なぜならば，同一オブジェクトへの複写では，複写元と複

写先の位置の一部が重なっていることがあり，正しい順序で複写しなければデータを破壊してしまうからである．たとえば，GETSTACK 命令の $C_{\mathtt{s}}$ 操作は長さ n の列 $\mathtt{S}^{\mathtt{n}}$ を 1 つ左にずらす操作であり，これは当然，列の左端から順に複写していかなければならない．

VMB では，列の複写に関して，複写元と複写先が同一オブジェクトである場合にはあらかじめ両者の位置を比較しておき，それによってどちらから複写するかを情報として保持しておく．位置関係が実行時まで定まらない場合には，実行時に比較を行って複写順序を決定するようなコードを出力する．

それぞれの原子的操作からコードを生成する際には，代入文などの直接的な形で出力するのではなく，C 言語の備えるマクロ機能をインタフェースとして利用することで，データ操作の具体的な実装方法からインタプリタを独立させる☆．たとえば，原子的操作 $\mathrm{R}_{\mathrm{d}}^{\mathrm{T}} \leftarrow \mathrm{R}_{\mathtt{reg}}$ によってレジスタ reg の値を一時変数 $\mathrm{R}_{\mathrm{d}}^{\mathrm{T}}$ に取得するには，d = reg; という直接的な代入文を出力するのではなく，GET_reg(d); というようなマクロを使う形にして出力する．

5. 実装と評価

本研究では，前章までに述べたインタプリタ生成系 VMB を実装した．VMB は関数型言語 Objective Caml（O'Caml）で実装されており，全体のコード量は約 5,300 行である．

本章では，実装した VMB を利用して本研究の評価を行う．まず 5.1 節では，VMB を用いて実際に処理系を開発した経験から，言語処理系の開発工程で VMB を利用することの有用性を確認する．次に 5.2 節では，既存のバイトコード言語の仮想機械を VMB を用いて実装し，それらを比較することで VMB が生成するインタプリタの性能を評価する．

5.1 VMB を用いたバイトコード処理系開発

本研究の主張である，仮想機械の開発工程においてインタプリタ生成系を用いる手法について，その有効性を実際に確かめるため，VMB を用いて簡単な手続き型言語の処理系を実装した．本節ではこのことについて述べ，VMB を言語処理系開発の場面に適用することの有用性を示す．

ソース言語 $\mathrm{W_H}$

まず，言語処理系の開発における最初の段階として，ソース言語の設計を行った．ここでは，ソース言語として C 言語風の文法を持つ簡単な手続き型言語を設計した．以下，この言語を $\mathrm{W_H}$ とする．$\mathrm{W_H}$ の主要な言語機能を以下にまとめる．

- **データ型**：$\mathrm{W_H}$ は基底データ型として整数型（int），実数型（double），文字型（char），論理型（bool）を持つ．また，これらの型をもとにした配列を利用できる．
- **文**：$\mathrm{W_H}$ の文は，代入文，if 文，while 文，print 文の 4 種類とする．print 文は任意の型の値を受け取り，それを端末に表示する．
- **組み込み演算**：組み込み演算として，通常の四則演算，論理演算，比較演算を提供する．そのほかに，配列の長さを取得する演算を提供する．
- **関数**：$\mathrm{W_H}$ では，引数と返り値を持つ関数を定義できる．関数の中では，局所変数を宣言できる．なお，トップレベルで宣言された変数は大域変数とする．

```
fun int fib(int n) {
  int n1, n2;
  if (n <= 1) return 1;
  else return fib(n - 1) + fib(n - 2);
}
print fib(30);
```

図 10 $\mathrm{W_H}$ によるフィボナッチ数の計算
Fig. 10 A $\mathrm{W_H}$ program which calculates fib(30).

$\mathrm{W_H}$ のプログラム例として，フィボナッチ数を計算するプログラムを**図 10** に示す．

仮想機械の設計

次に，$\mathrm{W_H}$ のプログラムを実行する仮想機械として，**図 11** に示す簡単なスタック機械を設計した．仮想機械によって読み込まれたバイトコードプログラムは，int 型の配列に格納されるものとする．仮想機械のスタックは関数ごとに作られるフレームから構成される．各フレームは，引数の数，動的リンク，戻り番地といった制御のために必要な情報，引数領域，局所変数領域，および，計算途中の値を格納するための領域からなる．大域変数は専用の領域に格納される．$\mathrm{W_H}$ の配列と実数値はヒープ領域に割り当てられ，これらは自動メモリ管理機能によって管理されるようにする．実行時に必要となる実数型の定数値は，あらかじめ配列に保持されているものとする．

この仮想機械上で計算を進めるためのバイトコード言語 $\mathrm{W_B}$ の命令を，**表 5** にまとめる．$\mathrm{W_B}$ のプログラムの例として，図 10 のプログラムをコンパイルして得られる $\mathrm{W_B}$ の命令列を**図 12** に示す．

☆ 出力言語が C 言語であるから，C 言語のマクロ機能を利用する．関数としてもかまわないが，実行性能のためにマクロとした．

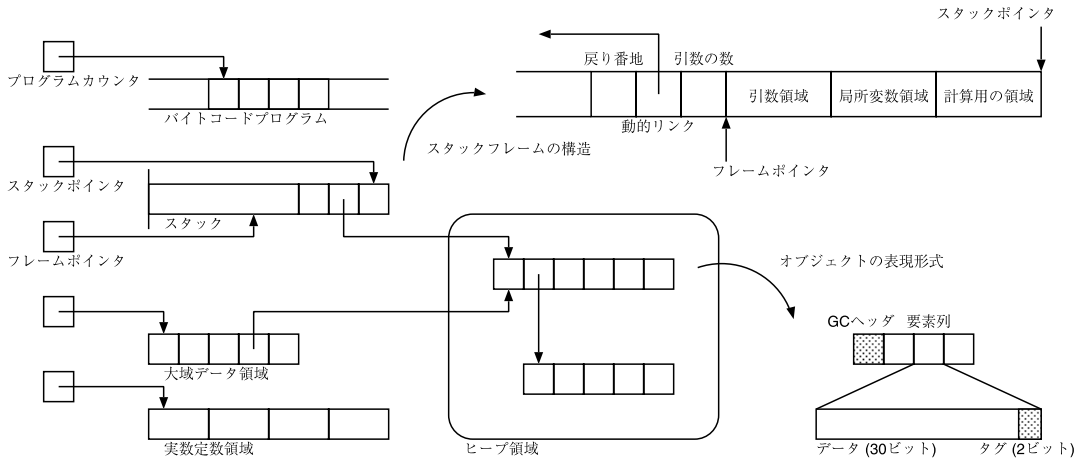

図 11 W_H プログラムを実行する仮想機械の内部構造
Fig. 11 The internal structure of the virtual machine which executes W_H programs.

表 5 W_B のバイトコード命令
Table 5 Bytecode instructions defined in W_B.

命令の分類	数	命令の簡単な内訳
メモリ読み書き	8	大域変数，局所変数，引数，配列
定数の読み込み	5	真，偽，`int`，`char`，`double`
関数呼び出し	3	呼出，復帰，局所変数領域確保
組み込み演算	14	算術，論理，比較，同値，配列長
制御命令	2	条件分岐，無条件ジャンプ
配列の領域確保	1	
`print`	1	
実行終了命令	1	

```
 0  CALL 0 36          23  CALL 1 4
 3  STOP               26  CONSTINT 1
 4  FRAME 2            28  GETARG 0
 6  CONSTINT 1         30  SUB
 8  GETARG 0           31  CALL 1 4
10  LE                 34  ADD
11  BRANCHIFNOT 5      35  RETURN
13  CONSTINT 1         36  FRAME 0
15  RETURN             38  CONSTINT 30
16  JUMP 18            40  CALL 1 4
18  CONSTINT 2         43  PRINT
20  GETARG 0           44  CONSTINT 0
22  SUB                46  RETURN
```

図 12 フィボナッチ数を計算する W_B 命令列
Fig. 12 A W_B bytecode program which calculates fib(30).

処理系の構築作業

VMB を利用して上に述べた設計の言語処理系を実際に構築するために要した作業量は，以下のとおりであった．

- **コンパイラの実装**：W_H から W_B へのコンパイラを実装した．コンパイラの実装には O'Caml を用いた☆．コンパイラの実装は 2,725 行であった．
- **インタプリタの生成**：VMB を利用してインタプリタを生成した．VMB に与える仕様記述は 191 行であり，VMB が出力したインタプリタのコード断片は 1,194 行であった．
- **インタプリタのインタフェースの実装**：データアクセスや型変換，組み込み演算など，VMB が生成したインタプリタから利用されるマクロの実装を行った．これらの実装は 226 行となった．
- **仮想機械を構成するその他の部分の実装**：メモリ管理機能やプログラムのロード部分など，インタプリタ以外の部分の実装を行った．メモリ管理機能はコピー方式 GC を実装した．これには 194 行を要した．メモリ管理機能以外の諸機能の実装に，さらに 301 行を要した．

評価

記述量の比較のために同一の仮想機械を VMB を用いずに実装した結果，上記のインタプリタ部分とマクロ実装に対応する部分とを合わせて，499 行の記述となった（その他の部分は VMB を用いたものと同一の実装を共有できたので記述量は変わらない）．このことから，記述量という単純な観点からの限定的な評価ではあるが，本研究が提案する仕様記述からのインタプリタ生成手法に一定の効果があることを確認した．

VMB を用いる処理系開発が特に有用な場面として，バイトコード命令セットを頻繁に変更して実験を行うような場面があげられる．その理由は，VMB を用い

☆ 字句解析部，構文解析部は ocamllex，ocamlyacc を利用して生成した．コンパイラの 2,725 行には lex 記述，yacc 記述を含むが，これらが生成したファイルは含まない．

た開発では仕様記述よりもデータアクセス操作や組み込み演算などのインタフェースの実装に多くの記述を要するという経験が得られたためである．これらのインタフェースは仮想機械のデータ構造によって決まるものであって個々のバイトコード命令には依存しないため，一度インタフェースを実装してしまえば，バイトコード命令の追加や変更は仕様記述上での変更のみで容易に対応できる．

5.2 生成されるインタプリタの性能評価

VMB によって生成されるインタプリタの性能を評価するため，既存の言語処理系の仮想機械を VMB を用いて再構築し，それぞれを用いてベンチマークプログラムの実行速度を比較した．ここでは，この実験を行うための言語処理系として O'Caml のバイトコード処理系を用いた．O'Caml 仮想機械は，レジスタとスタックを用いて計算を進めるスタック機械であり，VMB が生成対象とする仮想機械のモデルに合致する．レジスタは，スタックトップのキャッシュのような役割を果たすものであり，このレジスタを計算機の実レジスタに割り付けることにより，実行時のメモリ操作を削減して高速な処理を可能にしている．

O'Caml 仮想機械の生成

まずはじめに，仮想機械の生成のために O'Caml の仮想機械の構造と各バイトコード命令の意味を調べ，仮想機械の仕様記述を与えてインタプリタを生成した．バイトコード命令の意味を記述した結果は，**表 6** のとおりである☆．表の括弧内の数字は，その分類の命令のうち意味を記述できなかったものの数を示す．オブジェクト生成に分類される `CLOSUREREC` 命令の記述には，列の各要素に演算を適用するような記法が必要となり，現在の仕様記述言語にはそのような記法がないため，これを記述できなかった．その他の各命令は，仮想機械の具体的な実装に密接に依存した命令であり，命令の意味を形式的に表現すること自体が難しいように思われる．これらの記述できなかった命令については，インタプリタを生成した後に手作業で O'Caml の本来のコード断片に差し替えて対応した．

表 6 O'Caml のバイトコード命令の分類
Table 6 Bytecode instructions of O'Caml.

分類	命令数（記述不可）	
スタック操作	21	
オブジェクト読み書き	40	
オブジェクト生成	11	(1)
関数適用	12	
制御命令	13	
演算命令	33	
例外処理	4	(1)
外部関数呼び出し	6	(6)
メソッド操作	3	(2)
仮想機械制御	3	(3)
合計	146	(13)

☆ 命令の無用な細分化を避けるため，表 6 ではヒープ領域に割り付けられる通常のデータのほかに，環境，大域変数などを含むすべてのデータをオブジェクトと総称した．

ベンチマーク結果

VMB を用いて構築した仮想機械で，O'Caml のベンチマークプログラムを実行した結果を**表 7** に示す．表中の VMB と Orig の項目はプログラムの実行時間（秒）である☆☆．末尾の項目には，VMB の実行時間を Orig のそれで割ったものの逆数を速度比として示した．速度比の項目の値が大きいほど，VMB を用いて生成された仮想機械の実行性能が良いことを意味する．それぞれの仮想機械は gcc（バージョン 2.95.3）を用いてコンパイルし，その際，gcc の最適化オプションには -O を指定した☆☆☆．

表 7 が示すように，この実験では VMB を用いて構築した仮想機械が本来の O'Caml 仮想機械の 97%の実行速度を達成した．十分な性能評価のためには他のさまざまな仮想機械を広く調査して同様の実験を行う必要があるが，今回の実験結果は，本稿で提案したインタプリタ生成アルゴリズムによって効率的なインタプリタを生成できることを示唆している．

表 7 ベンチマーク結果
Table 7 Benchmark results.

プログラム	VMB	Orig	速度比
Boyer	5.98	5.72	0.95
FFT	5.65	5.46	0.97
Fibonacci	7.07	7.00	0.99
GenLex	6.38	6.04	0.95
KB	6.13	6.08	0.99
LifeGame	5.51	5.88	1.07
Logic	9.52	8.85	0.93
Mandelbrot	7.09	6.93	0.98
Nucleic	5.61	5.45	0.97
Quicksort	6.08	6.17	1.01
RatioRegions	5.97	5.37	0.90
Sieve	9.44	8.71	0.92
Simple	7.87	7.63	0.97
Soli	5.14	5.67	1.10
TakC	5.90	5.65	0.96
TakU	7.43	6.28	0.85
TSP	5.94	5.65	0.95

Intel Pentium4 2.4 GHz，512 MB memory，Linux 2.4.22

☆☆ 各プログラムの実行時間が比較的揃っているのは，実行時間が 5 秒を超える程度に問題のサイズを調節したためである．

☆☆☆ O'Caml の Makefile では -O が指定されており，これを -O2，-O3 に変更して構築した仮想機械においても有意な速度向上が見られなかったため，-O のまま比較を行った．

VMB を用いた仮想機械で，TakU をはじめとするいくつかのプログラムで大幅な性能低下が見られる原因は，オブジェクト生成命令において非効率なコードを出力してしまうためである．O'Caml 仮想機械では世代別コピー方式の GC によってメモリ管理を行う．世代別 GC ではオブジェクトに値を代入する際に特別な処理が必要となるが，新たなオブジェクトを初期化する際には，その処理の必要がない．本来の O'Caml 仮想機械ではその性質を利用しているが，VMB の生成するインタプリタでは，オブジェクトの初期化の際にも GC のための処理を無駄に行ってしまう．このことが性能低下の原因である．VMB によって生成されたインタプリタを修正してこの問題を取り除いた場合，Sieve を除くすべてのプログラムで速度比は 0.98 以上となる結果が得られた．Sieve ではこの問題を取り除いても速度比が 0.95 にとどまっており，この原因は現在調査中である．

6. 考　　察

本章では，本稿で提案した VMB に存在するいくつかの問題点について議論し，将来の改良の方向性について述べる．

インタプリタ生成アルゴリズムの主要な部分である直列化アルゴリズムについては，まださまざまな改良の余地がある．特に主要なものは，現在のアルゴリズムは，操作の実行によって値が他の場所に複製されることを利用していない点である．簡単な例として，3 つのレジスタの値が

```
( x, y, z ) -> ( y, x, x )
```

と変化するような場合を考える．これは本来，まず `x` を `z` の位置に代入することで，一時レジスタを使用することなく 3 つの操作を実行できるはずである．つまり，1 つの操作を実行することはそれ自体が操作を退避したのと同じ効果を持ち，残りの操作の依存関係を減らすことにつながる．しかし現在の VMB ではこれを考慮していないために，`x` と `y` のいずれか一方を一時レジスタに退避してしまう．

直列化アルゴリズムを拡張してこれを考慮するように変更するのは容易であるが，単純な拡張ではかえって実行速度を低下させることもある．たとえば上記の例で，`z` にあたるものがレジスタではなく参照をいくつもたどった先のオブジェクトであったとすれば，そこから `x` の値を利用するコストは大きなものとなる．このことから，この問題を考慮する賢いアルゴリズムを作るためには，メモリアクセスのコストを加味したモデルを導入しなければならないだろう．

直列化アルゴリズム以外の部分の問題として，現在の VMB は 4.4 節で原子的操作列を求める際にもとの操作 1 つ 1 つを独立に処理しているため，同じ処理を何度も繰り返すコードを生成してしまうというものがある．たとえば，

```
(v, [x].[y], p)->(x+y, [x+y], next)
```

というような規則では，2 つの `x+y` が別個の操作として扱われるため，VMB が生成するコードではそれぞれの代入操作で加算を行ってしまう．現在の VMB は C 言語のコードを生成しているため，C コンパイラの最適化によって除去できる場合も多いが，これが単純な加算命令ではなくユーザによって定義された演算であれば VMB がインタプリタを生成した後にそのような最適化を行うことは難しくなる．5 章で評価に用いた O'Caml の仮想機械ではこの問題は顕在化していないが，これは生成されるインタプリタにおいて大幅なオーバヘッドとなる可能性がある．

この問題を解決するためには，原子的操作列を求めた後に VMB が独自にコピー伝搬を行えばよいであろう．原子的操作列の形式は通常のプログラミング言語の代入命令とほとんど変わらないので，このような最適化を VMB に実装することに特別な困難はないと思われる．

7. 関 連 研 究

本研究と同様に，命令の意味を抽象的に記述してインタプリタの実装を生成する研究として，Vmgen [2] がある．Vmgen では，

```
sub ( i1 i2 -- i ) i = i1 - i2;
```

という形で命令実行前後のスタックの要素の関係を記述し，この記述からインタプリタのコードを生成できる（`--` の前後がそれぞれ実行前後のスタックトップを表し，`i = i1 - i2` の部分で実行前後の値の関係を表現する）．しかし，記述の表現力は実質的にスタックの各要素への名前付けにとどまっており，実際の操作は `i = i1 - i2` のように C 言語で記述しなければならない．本研究は，記述言語と生成アルゴリズムを工夫して，より広範な操作に対するインタプリタ生成を可能にした．Vmgen ではインタプリタ生成のほかに，プロファイラやデバッガなどの生成，複合命令の生成，計算機のハードウェア的な特性を利用した効率化などが可能であるが，これらは本研究にも導入可能であると思われる．特に複合命令の生成では，著者らの文献 11) の解析手法を用いることで，Vmgen での方法よりも効率の良いコード生成が可能であると考えている．

Ovm[9] では，仮想機械を構成するモジュール群を処理系開発者が利用しやすい形でフレームワークとして提供している．Ovm の特徴的な点は，バイトコード命令を Java のクラスとして定義することである．命令の内容によってクラス階層が作られており，既存のクラスを拡張することで新たな命令を定義できる．文献で著者らも述べているように，命令がクラスとして実装されていることにより，バイトコード命令の解析などを見通し良く実装できるように思われる．このことは本研究においても参考にすべき点である．一方，命令の記述力については Vmgen と同様に限定的なものであり，本研究のようにデータ列を簡潔に表現することや，依存を解析して効率良い命令を生成することなどは行っていない．

仮想機械における重要な機能の 1 つであるメモリ管理システムをインタプリタと独立に実装する手法として，文献 1)，4)，10) がある．メモリ管理システムは提供する機能の性格上，仮想機械のオブジェクトの内部表現と密接な依存関係を持つことになり，メモリ管理システムに影響される形で仮想機械の実装全体がモジュラリティを失いがちである．著者らの文献 10) では，インタプリタがデータ構造を直接操作せずに適切なインタフェースを介して操作することで，メモリ管理機能との独立性を保つ手法を提案した．VMB が生成するインタプリタがデータ操作の際に用いるインタフェースは文献 10) で提案されたものと対応付けることができ，このインタフェースを利用してメモリ管理システムを独立に実装できるように考慮されている．

8. おわりに

本稿では，仮想機械の形式的な記述からのインタプリタ生成手法を提案した．提案手法に基づく生成系 VMB を用いて簡単な手続き型言語の処理系，および，O'Caml の仮想機械を構築し，提案手法の有効性を示した．O'Caml 仮想機械を用いたベンチマークテストでは，VMB の生成する仮想機械が O'Caml 本来の仮想機械の 97%の性能を達成した．

本稿で提案した記述言語は，言語非依存なバイトコード解析のための基盤として利用できるのではないかと考えている．文献 11) では 1 つの応用として合成命令の生成手法を提案しているが，このほかにもさまざまな応用がないか探っていきたい．

また，VMB は，実際に言語処理系を開発する際に有用であるほかに，コンパイラ教育の場面でフレームワークとして利用することも有望であると考える．この用途のためには，仕様記述のための GUI ツールの開発などが課題であると考えている．

参考文献

1) Attardi, G. and Flagella, T.: A Customisable Memory Management Framework, *Proc. 1994 Usenix C++ Conference*, pp.123–142, Usenix Association (1994).
2) Ertl, M.A., Gregg, D., Krall, A. and Paysan, B.: vmgen — A Generator of Efficient Virtual Machine Interpreters, *Software—Practice and Experience*, Vol.32, No.3, pp.265–294 (2002).
3) Gosling, J., Joy, B., Steele, G. and Bracha, G.: *The Java Language Specification*, 2nd Edition, Addison-Wesley (2000).
4) Hudson, R., Moss, E., Diwan, A. and Weight, C.: A Language-Independent Garbage Collector Toolkit, Technical Report 91-47, Object Oriented Systems Laboratory, Department of comp. and Info. Science, Amherst, MA, 01003 (1991).
5) Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification*, Second Edition, Addison-Wesley (1999).
6) Microsoft Corporation: .NET Framework Developer Center. http://msdn.microsoft.com/netframework/
7) 緒方大介：バイトコードインタプリタ作成のためのツールキットの実現，修士論文，東京工業大学大学院情報理工学研究科 (2003).
8) 緒方大介，脇田　建，内山雄司，佐々政孝：バイトコード命令の操作的意味記述を用いた仮想機械核生成系，第 18 回大会論文集，日本ソフトウェア科学会 (2001).
9) Palacz, K., Baker, J., Flack, C., Yamauchi, C.G.H. and Vitek, J.: Engineering a customizable intermediate representation, *Proc. 2003 Workshop on Interpreters, Virtual Machines and Emulators*, pp.67–76, ACM Press (2003).
10) 内山雄司，脇田　建：メモリ管理機能のモジュラーかつ効率的な実装手法，情報処理学会論文誌：プログラミング，Vol.43, No.SIG 1 (PRO 13), pp.10–24 (2002).
11) 内山雄司，脇田　建，緒方大介：意味記述を用いたバイトコード命令連結手法，第 19 回大会論文集，日本ソフトウェア科学会 (2002).
12) Wakita, K., Uchiyama, Y. and Ogata, D.: Generation of efficient virtual machine, *Conference Record of International Lisp Conference 2002* (2003).

(平成 16 年 9 月 30 日受付)
(平成 16 年 12 月 27 日採録)

内山 雄司

1975 年生．1999 年東京工業大学理学部情報科学科卒業．2001 年同大学大学院情報理工学研究科目数理・計算科学専攻博士前期課程修了．同大学院博士後期課程在学中．プログラミング言語，言語処理系の実装方式等に興味を持つ．ACM 学生会員．

緒方 大介

1977 年生．2001 年東京工業大学理学部情報科学科卒業．2003 年同大学大学院情報理工学研究科目数理・計算科学専攻博士前期課程修了．同年ソニー（株）入社．プログラミング言語処理系等に興味を持つ．

脇田 建（正会員）

1965 年生．1989 年東京大学理学部情報科学科卒業．1991 年同大学大学院理学系研究科情報科学専攻修了．東京工業大学大学院情報理工学研究科数理・計算科学専攻助教授．博士（理学）．プログラミング言語，分散処理等に興味を持つ．日本ソフトウェア科学会，ACM，IEEE 各会員．