

JVM バイトコードへの低水準操作を 簡潔に記述可能なマクロシステム

馬谷 誠二^{1,a)}

概要: 本稿では, JVM バイトコードへの低水準な操作を簡潔に記述するための Clojure ライブラリ `bc-macro` を紹介する. `bc-macro` ライブラリでは, 木構造に対する操作を, Java ベースのバイトコード操作ツールによく見られるようなビジターパターンではなく, パターンにマッチした部分木を別の木に置き換える操作 (マクロ展開) の自動適用によって行う.

キーワード: JVM, バイトコード操作, Lisp

A Macro System for Low-level and Concise JVM Bytecode Manipulation

SEIJI UMATANI^{1,a)}

Abstract: In this paper, we introduce `bc-macro`, a Clojure library for writing low-level manipulations of JVM bytecode in a concise manner. In `bc-macro`, instead of using the visitor pattern adopted in many Java-based tools, each bytecode manipulation is represented as a substitution (macro-expansion) of another tree for the subtree matched with a specific tree pattern.

1. はじめに

現在主流の JVM バイトコード操作ツールは, およそ次の 2 種類に分類できる. 1 つは, クラスファイル構造を表す構文木を対象とするビジターパターンによって低水準な操作を記述するツールであり, 事実上の標準ツールと言える ASM[2] がその代表である. もう 1 つは, アスペクト指向プログラミング (AOP) を用いて, バイトコードの特定箇所へのコードの織り込みを可能にするツールである. 前者のビジターパターンによる記述においては, 複数ノードにまたがる構文木の操作が複数メソッドにまたがるため可読性が良くない, 静的文脈情報の手動管理が必要等, しばしば煩雑な記述となってしまう. また, 後者については, AspectJ 言語を直接利用するツール [1] に始まり, 柔軟性

を改善するための先進的な機能を備えた DSL 言語を利用するもの [5] まで幅広い開発・研究が行われているが, 現時点では前者と同程度の柔軟性を備えているものではなく, 適用可能な応用は限定的と言わざるを得ない.

一方, 近年, JVM はオブジェクト指向言語に限らず様々な高水準プログラミング言語の共通プラットフォームとして利用されており, クラスロード時のバイトコード操作を実現するのに, Java 言語あるいはオブジェクト指向パラダイムに縛られる必要はなくなっている. そこで, 本発表では, ASM と同程度に柔軟な表現能力を備えながら, より簡潔な記述を可能とするため, Lisp マクロのアイデアを応用したバイトコード操作ツールを提案する. マクロによる構文木の書換えは, ビジターの巡回による構文木への一連の操作と比較し, より宣言的なメタ操作記述を可能にする. また, 実用面においても, Lisp 対話環境による段階的な操作コードの開発は, ともすれば煩雑になりがちな低水準バイトコード操作を伴うソフトウェアの開発効率を向上するものと考えられる.

¹ 京都大学大学院情報学研究科
Graduate School of Informatics, Kyoto University, Kyoto
606-8501, Japan

^{a)} umatani@kuis.kyoto-u.ac.jp

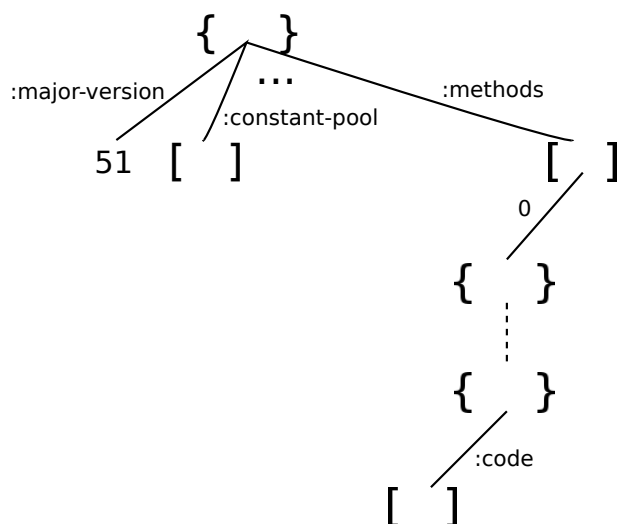


図 1 JVM バイトコードの抽象構文木

Fig. 1 AST of JVM bytecode.

提案ツールのマクロ展開の仕組みは、拡張構文を実現することが主目的の Lisp マクロとは次の点で大きく異なる。まず、提案ツールは、生のバイトコードへの書換えを指定できるよう、フォームの先頭要素ではなく、構文木の一部と木パターンをマッチさせながら順次マクロ展開を行う。これは、AOP ベースのツールにおけるポイントカットに相当する機能と言えるが、より柔軟に任意のコードフラグメントを切り出すことが可能である。さらに、木パターンとのマッチングに伴い静的文脈情報を自然に取り出せるため、ビジターパターンのように静的文脈情報を手動で管理することによる煩雑さを避けることが可能である。

論文の構成は次のとおりである。まず、2 節では提案するマクロシステム `bc-macro` の設計について説明する。次に、3 節実際に `bc-macro` を使った JVM バイトコード変換の例をいくつか示す。その後、4 節で木変換マクロの実装について簡単に触れ、最後に 5 節でまとめを行う。

2. 設計

2.1 システムの概要

提案する Clojure 用バイトコード操作ライブラリ `bc-macro` は、ASM などのオブジェクト指向言語ベースのツールとは異なり、ホスト言語である Clojure が直接サポートするデータ型のみを用いて、クラスファイルのデータ構造（木）を表現する（図 1）。特に、木の中間ノードを構築するためのデータ型としては、ベクタとマップのみを用い、リストやその他のシーケンス型は使用しない。JVM バイトコード仕様を木で表現するにはそれら 2 つのデータ型があれば十分である。また、マップで表現する中間ノードのキーの名前は JVM 仕様のレコード定義中に出てくる名前と完全に一致しており、操作ツール独自のネーミング規則を覚える必要はない。

`bc-macro` ライブラリを使って定義されたマクロ展開関

数（定義の仕方について後述）は、典型的には、クラスロード時に前述のと通りの内部表現へと変換されたクラスファイルに対し、それ以上展開が行われなくなるまで繰り返し適用される。その後、JVM の通常のバイトコード検証器等を経、実行される。また、それ以外に、木の一部に対し特定のマクロ展開関数（集合）を手動で起動する方法も提供している。

2.2 木パターン

本節では、`bc-macro` システムが用いる部分木指定のための記法（木パターン）について説明する。前節で簡単に触れたとおり、木パターンは、置換対象（部分）木に対応する部分（対象パターン）とそれを取り囲む静的文脈（以降、単に文脈と呼ぶ）に対応する部分（文脈パターン）とに分かれる。木パターン全体と対象木の照合は：

- (1) 対象パターンと対象木の照合を、トップダウンに行う。
 - (2) 文脈パターンと文脈の照合を、ボトムアップに行う。
- の手順が両方とも成功した場合に限り、成功したものと見なされる。（トップダウン、ボトムアップの意味については 2.2.1 節、2.2.2 参照。）

対象パターンと文脈パターンでは、パターン中に指定できる内容や記述方法が異なるため、以下では、別々に順を追って説明する。両方を含めた完全な木パターン文法の BNF 定義は、付録 A.1 に載せてある。

2.2.1 対象パターン

対象パターンには、変数、定数、アンクォート、ベクタ、列、マップの 6 種類があり、それぞれ簡潔な記法が用意されている。以下、簡単な例を示しながら、それぞれの意味について順に説明する。

変数: Lisp シンボルは変数を表し、任意の部分木とマッチする。変数はマクロ本体や、この変数を含むパターンより後ろのパターン中で参照可能である（「後ろ」の意味については後述）。たとえば：

```
(defbcmacro dup (x) [x x])
```

と定義されているマクロ展開関数 `dup` の呼出し (`dup [1 2 3]`) は `[[1 2 3] [1 2 3]]` へと評価される。**定数:** Clojure の定数リテラルは木パターン中で、そのまま使用でき、その値を持つノードとのみマッチする。たとえば：

```
(defbcmacro three (3) ...)
```

は整数リテラル 3 のみからなる木とマッチする。

アンクォート: パターン中で、Clojure の変数を参照したい場合には `~form` と書く。form の評価結果がパターン中に埋め込まれる。たとえば、先程の `three` を：

```
(def two 2)
```

```
(defbcmacro three (~(inc two)) ...)
```

と定義することもできる。

ベクタ: パターン `[p0 p1 ... pn-1]` は、要素数 n のベク

タのうち、各要素がパターン中の対応する位置の部分パターン p_i とマッチするものとマッチする。たとえば：

```
(defbmacro zero-and-two ([x _ y _])
  [x y])
```

では要素数 4 のベクタを、0 番目と 2 番目の要素だけからなるベクタへ変換する。

また、Clojure の分配束縛と同様に、 $\dots \& p]$ によりある位置から残りの要素全体をベクタにし、パターン p とマッチさせることができる。

ベクタパターンの照合は、木構造をトップダウンに（つまり、深さ優先で左から右に）巡回しながら行われる。この巡回順序に従ってパターンの前後関係が一意に決まっており、あるパターン中で、それより前のパターンによって束縛された変数を参照することも可能である。たとえば、パターン $([x \sim x])$ は、要素数 2 のベクタで、なおかつ、2 つの要素が同じものにマッチする。

列：変換を記述する際、ベクタ全体ではなく部分列を置換対象としたい場合がある。そのような場合は、単純にパターンの列で置換対象を表す。列の先頭のパターンが対象部分木とマッチし、かつ、残りすべての後続パターンが対象部分木の右側兄弟木と順番にマッチする場合にマッチする。たとえば：

```
(defbmacro from-one (1 2 3) ...)
```

のマクロを、木 $[0 \ 1 \ 2 \ 3 \ 4]$ の 1 を対象にマッチさせることが可能である。

マップ：パターン $\{p_0 \ k_0 \ \dots \ p_{n-1} \ k_{n-1}\}$ は、パターン中の全てのキー・パターンペア $[p_i \ k_i]$ ($i = 0 \dots n-1$) とマッチするキー・バリューペアを含むマップとマッチする。たとえば、 $\{51 :major-version \ x :minor-version\}$ は、 $:major-version$ キーに対応するバリューが 51 で、かつ、 $:minor-version$ キーを含む（それ以外を含んでも良い）任意のマップとマッチする。

また、ベクタとは異なり、マップのキー・バリュー要素間には順序関係がないため、マップパターンには巡回順序の前後関係は存在しない。

対象パターンには、上記 6 種類に加え、ベクタパターンとマップパターンに対し適用な *as* パターンとガードパターンが存在する。*as* パターンは $[\dots :as \ x]$ 、 $\{\dots :as \ x\}$ のように書き、変数 x はこのパターンとマッチした対象木全体に束縛される。

ガードパターンは、*#when pred* のように書き、ベクタやマップの任意の要素位置に埋め込むことができる。ガードパターンの埋め込まれたパターンは、*pred* が真（Clojure では *nil*, *false* 以外は全て真）に評価される場合にのみマッチする。なお、*pred* の評価されるタイミングはベクタパターンとマップでパターンでは異なる。ベクタパターンでは、埋め込まれた位置により通常の要素パターンと同じ前後関係に従って評価される。マップパターンの場合、前

述のとおりキー・バリューペアの間に巡回の前後関係は存在しないが、ガードパターンを記述する際にそれらの中で束縛された変数を参照できると便利のため、ガードパターンは必ず最後に評価されることとしている。

as パターン、ガードパターンの有効な利用法については、3 節で具体例により示す。

2.2.2 文脈パターン

文脈パターンは対象木を部分木に含む親ノード（ベクタあるいはマップ）を表現するための記法である。それらの記法は、*as* パターンが書けることなどを含め、基本的には対象パターンのベクタ、マップと同じである。たとえば、マクロ：

```
(defbmacro sum [0 (x y z) 4] (+ x y z))
```

は、ベクタ $[0 \ 1 \ 2 \ 3 \ 4]$ を $[0 \ 6 \ 4]$ に展開する。ここで、文脈パターン中の対象パターン全体は丸括弧で囲まれているが、丸括弧は静的文脈と置換対象の間の境界指定の役割を貼たしている。^{*1}*bc-macro* では 1 回のマクロ展開における置換対象は木全体において 1 箇所だけのため、丸括弧に囲まれた対象パターンはパターン全体中に必ず 1 つだけ存在する。また、対象パターンについては、Lisp マクロのスプライシング・アンクォートのように新しい部分木を継ぎ木して埋め込みたい場合がある。そのような場合には、丸括弧の前に *@* をつける。たとえば：

```
(defbmacro ->roman [0 @ (1 2 3) 4] '[I II III])
```

は、ベクタ $[0 \ 1 \ 2 \ 3 \ 4]$ を $[0 \ [I \ II \ III] \ 4]$ ではなく $[0 \ I \ II \ III \ 4]$ に展開する。

文脈パターンに対するマッチングの巡回順序は、対象パターンとは逆にボトムアップかつ幅優先に行われる。さらに、文脈がベクタの場合、巡回は対象パターンを起点にまず左兄弟のマッチングが右から左へと進み、その後、対象パターンの右兄弟を左から右へと進む。たとえば、文脈パターン：

```
{x :foo
 [b a (t) c] :bar
 y :baz}
```

では、まず *a*, *b*, *c* の順にマッチングが試され、その後 *x* と *y* のマッチングが試される。*x* と *y* の間には巡回の前後関係はない。

さらに上記に加え、文脈パターンでは、対象パターンには存在しない 2 種類の省略パターンを記述できる：

列省略パターン：ベクタパターンの任意の位置にトークン $\dots$ を記述することで、任意個の要素木をスキップすることができる。スキップ処理は前述の巡回順序に従い $\dots$ の直後に書かれているパターンとマッチする要素が見つかるまで行われる。たとえば：

```
[y ... { _ :foo } ... (t) ...]
```

^{*1} 前節の全ての対象パターンが丸括弧で囲まれていたのは、空の文脈パターンを意味する。

のようなパターンの場合、対象パターンのマッチング後、まず左兄弟のうち{_:foo}とマッチする一番右側の要素の探索が行われる。その後、(任意の木とマッチする)変数 *y* にベクタの先頭要素が束縛される。また、対象パターンの右には後続のない...があるが、これはそこから右に向かって任意個の兄弟が並んでいてもいいことを意味している。

階層省略パターン: 文脈パターン中の任意の部分木パターン(対象パターンと文脈パターンのどちらでも可)の前に`#nest`指示子をつけることで、その部分木が直接の子ではなく、任意の深さに位置する子孫であることを指定できる。たとえば、パターン{ `#nest[1 (x) y] :foo` }は、{`:foo [1 2 3]` }だけでなく {`:foo [100 {bar [1 2 3]} 200]` }ともマッチする。

また、列省略パターンを同じパターン中に何個でも書けるのと同じように、階層省略パターンは、入れ子して多段に用いることが可能である。

これらの簡略記法が可能だと、図1のような木パターンを：

```
{ 51                                :major-version
  [& cs]                             :constant-pool
  [#nest{(c) :code} ...] :methods
}
```

と素直に表現できる。図における省略部分(...)とパターンにおける...および`#nest`の対応を含め、図示すれば直観的に分かりやすい木構造を、木パターンでもそのまま表現できていることが分かる。

2.3 マクロ定義

前節までの説明で、`defbcmacro` は既に用いているが、`bc-macro` では、用途に合わせ他にもマクロ定義の方法が用意されている。

(1) `(bcmacro pattern body)`

(2) `(defbcmacro name pattern body)`

(3) `(letbcmacro [(name pattern mbody)+] body)`

(1)は匿名のマクロ展開関数の生成、(2)は大域マクロ定義、(3)は局所マクロ定義である。マクロ本体 *body*, *mbody* には任意の Clojure コードを記述でき、*pattern* にマッチした対象部分木は、マクロ展開によりそれらの評価結果に置き換えられる。匿名マクロ展開関数も含め、上記3つにより定義したマクロの使い方については次節で説明する。

2.4 マクロ展開

前節で述べたマクロ定義方法によりつくられた(匿名も含め)マクロ展開関数は、通常関数呼出しによりマクロ展開に使用できる。引数には置換対象を指し示す zipper のロケーション(`zipper` 関数についての簡単な説明は4.1節参照)を引数に受け取り、マクロ中のパターンがその位

置の部分木とマッチした場合には、マクロ展開によりその部分木を置換した結果のロケーションを返す。マッチしなかった場合には例外が投げられる。

上記の方法は、`zipper` 関数を用いて対象部分木のロケーションを明示的に指定する必要があり不便である。そこで、Lisp のマクロ同様、木全体から適用可能な部分木を見つけ出し、展開を自動で行う関数が用意されている。

- `(bcexpand-1 tree)`: 引数に与えられた木 *tree* を深さ優先順に探索し、マッチする最初の部分木をマクロ展開し、木全体を返す。マッチする部分木が存在しない場合、引数に受け取ったのと同じ木をそのまま返す。

- `(bcexpand-all tree)`: 引数に与えられた木 *tree* を深さ優先順に探索しながら、マッチする部分木がなくなるまで、マクロ展開を行い、木全体を返す。マッチする部分木が存在しない場合、引数に受け取ったのと同じ木をそのまま返す。

`bcexpand-all` を使う場合、展開が無限ループしないよう注意が必要である。通常の Lisp のマクロの場合、展開後のフォームが無条件にマクロフォームへと展開しつづけないよう注意することで回避できるが、`bc-macro` の場合、パターンの形状によっては展開先の木に対してもマッチしてしまうことが多い。そこで、`bc-macro` では、木の任意のノードに対しメタデータを付加することで、以降のマクロ展開を抑制できるようにしている。具体的には、展開先の木を構築する際、`(with-aside [mfuns...] tree)` とすることで、*tree* は、マクロ展開関数群 *mfuns...* の対象から除外される。

また、`bc-macro` では、展開回数の制御だけでなく、マクロ展開実行時に適用されるマクロの集合を限定する仕組みも用意している。まず、上で述べたように、1引数で `bcexpand-{1,all}` を呼出した場合に適用されるマクロ定義は、現在の名前空間(`*ns*`)中の Var 集合(Clojure におけるトップレベル環境)に束縛されているものに限られる。したがって、Clojure の名前空間による一般的なモジュール化を `bc-macro` を利用するアプリケーションに対しても行える。

さらに、`(bcexpand-{1,all} [mfuns...] tree)` のように呼び出すことで、追加引数 *mfuns...* として適用されるマクロ展開関数の集合を渡すことができる。適用するマクロ集合を限定するこの機能は、`letbcmacro` によるマクロ定義の局所化と同時に用いることで、プログラムのモジュール化をより一層向上し、アプリケーションの各フェーズにおけるマクロ展開の柔軟な制御を可能にする。

3. JVM クラスファイル操作

前節では、JVM バイトコードとは関係のない単純な例を使って、`bc-macro` ライブラリの使い方を説明した。本節では、まず、ASM の文献[4]に挙げられている典型的な

JVM バイトコード操作例を `bc-macro` により書き直すことで、`bc-macro` の使用例を示す。なお、[4] 中の Java コードは、定義の載っていないメソッドを呼出したりしているため、コードの行数による単純な比較などを行うことはできない。

次に、JVM に対し末尾再帰最適化の機能を追加した研究 [6] の中からバイトコード変換に関する部分を抜き出し、その操作を `bc-macro` で簡潔に記述できることを示す。

3.1 簡単な変換の例

インターフェイスの追加

あるクラスへの実装するインターフェイスの追加は次のコードによって実現できる。

```
(defn add-class-info [cp c]
  (-> (conj cp (constant-utf8 c))
      (conj (constant-class (count cp))))))

(defn add-interface [cfile iface]
  (letbmacro
    [(addi ({cpc :constant-pool-count
              cp :constant-pool
              ifc :interfaces-count
              if :interfaces
              :as cf})
      (let [cp' (add-class-info cp iface)]
        (conj cf
              :constant-pool-count (+ cpc 2)
              :constant-pool cp'
              :interfaces-count (inc ifc)
              :interfaces (conj (+ cpc 2))))))
    (bcexpand-1 [addi] cfile)))
```

上記のコードでは、コンスタントプールをあらわすベクタ `cp` の末尾にあらたに実装するインターフェイスを追加している。

コンスタントプールを含め、クラスファイルマップ中の複数のフィールドを同時に変更するため、マップ全体を丸括弧で囲んで置換対象としている。また、置換は1度だけと分かっているので `bcexpand-1` の呼び出しで十分である。

クラスに対するフィールドの追加、メソッドの追加も同様に記述可能である。

フィールドアクセスの置き換え

フィールドアクセス命令 `getfield` は、フィールド名(とその型)を示すコンスタントプール中の値へのインデックスをオペランドに持つ。そのオペランド `old` を別のフィールド `new` (コンスタントプール中に存在すると仮定する。ない場合は、インターフェイスの追加と同様にコンスタントプールに追加すれば良い。)へと置き換えるには次のように実現する。

```
(defn replace-field-access
  [methods old new]
  (letbmacro
    [(replfa [_ :getfield (~old)]
      (with-aside [replfa] new)))
    (bcexpand-all methods)))
```

ここでは、`old` へのすべてのフィールドアクセスを対象とするため、`with-aside` と `bcexpand-all` を一緒に用いている。

メソッド呼出しの置き換えもまったく同様に可能である。メソッド先頭へのコードの追加

クラスファイル中の全メソッドを対象に、それらの本体コードの先頭に命令列 `code` を追加するには次のように書く。

```
(defn insert-begin
  [cfile code]
  (letbmacro
    [(ib {[@c] ...} :code]
      (with-aside [ib] (conj code c))))
    (bcexpand-all cfile)))
```

... パターンにより興味のある箇所だけをパターン中であらわせば良く、また、`@`指定によってコード列の挿入も自然に記述できている。

ただし、上記コードでは、... の位置にある命令中にジャンプ命令などのバイトコードオフセットを含む命令がある場合、それらのオフセットが `code` の長さ分だけずれてしまうことになる。そのような状況に対処するマクロ定義は次のようになる(`adjust-offset` の定義は簡単のため省略する)。

```
(defn insert-begin-2
  [cfile code]
  (letbmacro
    [(ib {[cs] :code]
      (with-aside [ib]
        (concat code (adjust-offset
                      cs (count code))))))
    (bcexpand-all cfile)))
```

メソッド終了直前へのコードの追加

メソッド先頭への追加と同様、リターン命令直前へも任意のコードを追加可能である。

```
(defn insert-exit
  [cfile code]
  (letbmacro
    [(ib {[... @([_ :return :as r]) ...} :code]
      (with-aside [ib] (conj code r))))
    (bcexpand-all cfile)))
```

... パターンと `:return` を含むベクタパターンによって、コードの挿入位置を自然な形で指定できている。

メソッド本体の置き換え

名前 `mname` を持つメソッドのコード本体を `code` へ置き換えるためのマクロとしては、次のような素朴なコードが考えられる。

```
(defn replace-method-wrong
  [cfile mname code]
  (letbcmacro
    [(replm {cp :constant-pool
              #nest{
                :method-info :kind
                index :name-index
                #when (= (:+value (cp index))
                          mname)
                #nest{(c) :code}
                :attributes}
              :methods}
      code])
    (bcexpand-1 [replm] cfile)))
```

しかしながら、文脈パターン中の変数のバインディングは内から外へと順に行われるため、このコードの `#when` ガード中の変数 `cp` は実際には未束縛変数である。

この問題を解決するには、次のようにしてマクロ定義を二つに分ければ良い。

```
(defn replace-method-aux
  [cp methods mname code]
  (letbcmacro
    [(replm {:method-info :kind
              index :name-index
              #when (= (:+value (cp index))
                        mname)
              #nest{(c) :code}
              :attributes}
      code])
    (bcexpand-1 [replm] methods)))

(defn replace-method [cfile mname code]
  (letbcmacro
    [(get-cp {cp :constant-pool
              (ms) :methods}
      (replace-method-aux
        cp ms mname code))])
    (bc-expand-1 [get-cp] cfile)))
```

まず、関数 `replace-method` 中のマクロによってコンスタントプールを取り出し、それを追加引数にして、メソッドを置き換えるためのマクロを含む関数 `replace-method-aux` を呼び出している。

3.2 末尾再帰の最適化

本節では、文献 [6] で提案されている手法のうち、JVM バイトコード変換に関する部分だけを取り出して `bc-macro` による実装を示す。ただし、[6] では呼出し命令が末尾再帰的であることの条件として3つを挙げているが、ここでは簡単のため、そのうちの2つだけを取り扱う。

まず、基本的な方法を説明するために、末尾再帰である呼出命令を検出・置換する素朴なマクロとして次を考える。

```
(defbcmacro tc1
  [...
    [addr (:invokestatic) idx]
    [_ :ireturn ]
    ...]
  :tailinvokestatic)
```

上記パターンでは、`invokestatic` 命令と `ireturn` 命令が連続している場合、その `invokestatic` 命令は末尾呼出しであるとしており、その場合には、オペコードを末尾再帰呼出し専用の命令 `tailinvokestatic` へ置き換える。ここで、この `invokestatic` 命令が呼び出すメソッドの返り値型が `int` であると仮定していることに注意して欲しい。

次に、さらに例外ハンドラまで考慮した実装を示す。

```
(defbcmacro tc2
  { [...
    [addr (:invokestatic) idx]
    [_ :ireturn ]
    ... ] :code
    ext :exception-table
  }
  (loop [es ext]
    (if (seq es)
      (let [[start end _ _] & es'] es)
        (if (and (<= start addr)
                  (< addr end))
          :invokestatic
          (recur es'))))
    :tailinvokestatic)))
```

例外テーブル `ext` には、各ハンドラが設定されている命令オフセットが格納されており、それと `addr` の比較を行うことによりこの `invokestatic` 命令に対し例外ハンドラが設定されているかを判定できる。もし設定されているなら、この呼出しは末尾再帰とは言えない。

最後に、`invokestatic` 命令で呼び出すメソッドの返り値型まで考慮した実装を示す。メソッドの返り値型情報は、`invokestatic` 命令のオペランドに指定されている `idx` を起点に、コンスタントプール中の値を複数回繰り返して参照することで到達できる。

```
(defbcmacro tc3
  { cp :constant-pool
```

```
#nest { [...
  [addr (:invokestatic) idx]
  [_      :ireturn          ]
  ...] :code } :methods }
(let [[cname-idx signature-idx] (cp idx)
      [name type] (cp signature-idx)]
  (if (.endsWith type "I")
      :tailinvokestatic
      :invokestatic)))
```

本節に載せた一連の例が示すように、bc-macro システムでは、ある変換操作を一つのマクロ定義だけで完全に記述する方法と、変換対象となるターゲットを大雑把なパターンによってある程度だけ絞り込み、その中に含まれるノードの切り分けや抽出などの操作を Clojure の通常のデータ操作により行う方法と、ユーザが書きやすいと思える方法を柔軟に選択できる。また、いずれの記述方法を取っても、bc-macro なら、ある特定の処理に関連したコードは（ユーザの好みに従い、多少の補助関数によって定義が分割されることはあるにしろ）ひとまとめに記述できる。ビジターパターンでは、ユーザ側にその選択肢はなく、ノード毎の visit メソッド単位で分離して記述することを強制させられる。

4. 実装

本節では、bc-macro システムの実装、とくに、木パターン表現を、パターンマッチを行う Clojure コードへ変換する方法について説明する。

4.1 Zipper

木パターンと木の特定のノードとのパターンマッチは、zipper 関数 [3] を使って行われる。直観的には、親や兄弟ノードへも効率良くアクセスできる zipper 関数を使うことにより、部分木方向へのマッチングだけでなく、文脈方向へのマッチングも効率良く実装できる。

zipper 関数は、ベクタについては Clojure 標準ライブラリ中に存在するものと帰納的にほとんど同じであるが、マップについては本システムを実現するのに相応しい性質備えている既存の実装がなかったため、独自の実装を行った。bc-macro が用いる zipper 関数の一覧を表 1 に示す。

4.2 木パターンからの照合 Clojure コードの生成

3 節で説明した木パターンは、前節の zipper 関数によって対象となる木を巡回しながらマッチングを行う Clojure コードへと変換される。木パターンが再帰的な構造を持った表現であるのに対し、典型的に zipper を使うコードは現在注目している箇所（前節の説明の loc）を引数に受け取り、あらたな loc を返す zipper 関数の一連の呼出しとなる。そこで、この両者の間の変換の繋ぎの役割を果たす中

間言語を導入する。中間言語の命令一覧を表 2 に示す。これら命令のリストは表中の直観的な説明にほぼそのまま対応した Clojure コードへとさらに変換されることになる。たとえば、パターン中の変数 *x* と現在指し示している部分木のマッチングを行う中間言語命令 (*:var x*) は、

```
(let [...
  x (node loc)
  ...]
  ...)
```

のような Clojure コードへと変換される。また、定数 *v* とのマッチングを行う中間言語命令 (*:val v*) は、

```
(let [...
  _ (when-not (= (node loc) v)
    (throw (MatchException. "...")))
  ...]
  ...)
```

のような Clojure コードへと変換される。ここで、マッチングに失敗した場合には例外を投げる（この例外の補足については、後述の *bcapply* 参照）。他の中間言語命令についても同様に、zipper 関数を用いた素直な Clojure コードへ変換することが可能である。

各木パターンから中間言語命令列への変換については、いくつかの典型的な例を示すことにする。たとえば、パターン *([1 2 x y & r])* は、部分木とのマッチングを行う命令列：

```
(:vec-down)
(:val 1)
(:right)
(:val 2)
(:right)
(:var x)
(:right)
(:var y)
(:right)
(:merge-rights)
(:var r)
(:remove-and-left)
(:remove-and-left)
(:remove-and-left)
(:remove-and-left)
(:vec-up)
```

へと変換される。また、文脈を伴うパターン *{t :tag, (mj) :major, [m n] :minor}* は、部分木とのマッチングを行う命令列：

```
(:var mj)
```

および、文脈のマッチングを行う命令列：

```
(:map-key :major)
(:map-up)
```

表 1 zipper 関数の一覧
Table 1 Zipper functions.

呼出しフォーム	機能の概要
(->zip node)	木のルート node を受け取り, zipper (場所) を返す.
(node loc)	loc の位置にある部分木を返す.
(lefts loc)	loc の親がベクタの場合, その左兄弟のシーケンスを返す.
(rights loc)	loc の親がベクタの場合, その右兄弟のシーケンスを返す.
(key loc)	loc の親がマップの場合, それに関連づけられている key を返す.
(root loc)	loc から木のルートまで全ての変更を反映しながら up し, ルートノードを返す.
(left loc)	loc の親がベクタの場合, そのすぐ左の兄弟の場所を返す. 左兄弟がない場合は nil を返す.
(right loc)	loc の親がベクタの場合, そのすぐ右の兄弟の場所を返す. 右兄弟がない場合は nil を返す.
(down loc)	loc がベクタの場合, それが指し示すノードの一番左の子ノードの場所を返す. 子ノードが存在しないなら nil を返す.
(down loc key)	loc がマップの場合, それが指し示すノードの子ノードのうち, key に関連づけられたものの場所を返す.
(up loc)	loc が指し示すノードの親ノードの場所を返す. oc が木のルートを指す場合, nil を返す.
(replace loc node)	loc の指し示す場所にある部分木を node と置き換える. 引数と同じ場所 (つまり node の場所) を返す.
(replace-splicing loc node)	loc の親がベクタの場合, loc の指し示す場所にある部分木を継ぎ木によってベクタ node と置き換える. node の一番左要素のノードを指す場所を返す.
(merge-rights loc)	loc の指し示す部分木およびその全ての右兄弟からなるベクタを, loc の指し示す部分木と置き換える. 引数と同じ場所を返す.
(remove-and-left loc)	loc の指し示す場所にある部分木を削除し, loc のすぐ左のノードの場所 (なければ nil) を返す.
(next loc)	場所 loc を含む木全体を深さ優先順に巡回する際, loc の次の場所を返す.
(end? loc)	場所 loc を含む木全体を深さ優先順に巡回する場合, loc が最後のノードなら true を, そうでないなら false を返す.

表 2 中間言語の命令一覧
Table 2 List of intermediate instructions.

命令フォーム	説明
(:var x)	現在の場所の部分木を変数 x に束縛する.
(:val v)	現在の場所の部分木が定数 v と一致するか確認する.
(:guard exp)	exp が真に評価されるか確認する.
(:unquote exp)	現在の場所の部分木が exp を評価した結果と一致するか確認する.
(:vec-down)	現在の場所の部分木がベクタであることを確認し, down を実行する.
(:vec-up)	現在の場所の部分木がベクタであることを確認し, up を実行する.
(:map-down key)	現在の場所の部分木がマップであることを確認し, (down key) を実行する.
(:map-up)	現在の場所の部分木がマップであることを確認し, up を実行する.
(:left)	left を実行する.
(:right)	right を実行する.
(:no-left?)	現在の場所の部分木がベクタのとき, 左兄弟が存在しないことを確認する.
(:no-right?)	現在の場所の部分木がベクタのとき, 右兄弟が存在しないことを確認する.
(:remove-and-left)	remove-and-left を実行する.
(:merge-rights)	merge-rights を実行する.
(:reset)	ベクタやマップの巡回中, 現在の場所を最初の場所へ戻す.
(:some-left pat)	ベクタを巡回中, 左兄弟の中に入れ子の中間言語命令列 pat にマッチする部分木が存在するか確認する.
(:some-right pat)	ベクタを巡回中, 右兄弟の中に入れ子の中間言語命令列 pat にマッチする部分木が存在するか確認する.
(:nest pat)	木のルート方向に向かって巡回しながら, 入れ子の中間言語命令 pat にマッチする文脈が存在するか確認する.

(:map-down :tag)	(:right)
(:var t)	(:var n)
(:map-up)	(:remove-and-left)
(:map-down :minor)	(:vec-up)
(:vec-down)	(:map-up)
(:var m)	へと変換される.

最後に、上記の変換を行うマクロ展開関数を生成するための Clojure マクロ `bcmacro` の定義は次のとおりである。

```
(defmacro bcmacro [pat & body]
  (let [[tgt cxt] (compile-pat pat [])
        [is-at tgt] (if (at-target? tgt)
                        [true (second tgt)]
                        [false tgt])
        tgt (compile-target tgt [])]
    '(fn [loc#]
      (let [~@(emit-code tgt)
            ~@(emit-code cxt)
            t# (do ~@body)]
        (~(if is-at
              'replace-splicing
              'replace)
         loc# t#))))))
```

`bcmacro` を呼び出す Clojure コードのコンパイル時に、`compile-pat` 関数によって中間コード `tgt`, `cxt` を求め、さらに `emit-code` 関数によって Clojure コードを生成している。

4.3 マクロ展開

ユーザがマクロ展開に直接使用する関数 `bcexpand-{1,all}` は、`zipper` 関数における木の特定の場所を対象に処理を行う、より低レベルな関数 `bcfind`, `bcapply` によって実現されている。

```
(defn bcapply [macros loc]
  (loop [[m & mnext :as ms] macros
        result nil]
    (or
     result
     (when ms
       (recur
        mnext
        (try (m loc)
              (catch Exception e nil)))))))

(defn bcfind [macros loc]
  (loop [loc loc]
    (if (end? loc)
      nil
      (or (bcapply macros loc)
          (recur (next loc))))))
```

`bcapply` は、`loc` に対し `macros` 中のマクロ展開関数を順に適用し、最初にマッチしたマクロの適用結果を返す。どれもマッチしなければ `nil` を返す。`bcfind` は、`loc` から始めて、木を深さ優先順で巡回しながらそれぞれのノードに対し、順に `bcapply` を適用する。最初に `bcapply` が `nil`

以外の値を（マクロ展開に成功し、その展開結果の場所を示す）を返したら、`bcfind` 自身もその値を返す。すべてのノードを巡回し終えてもマクロ展開に成功しなければ、`nil` を返す。

上記の関数を用いれば、`bcexpand-{1,all}` の実装は以下のとおり簡単である。

```
(defn bcexpand-1 [macros form]
  (if-let [rslt (bcfind macros (->zip form))]
    (root rslt)
    form))

(defn bcexpand-all [macros form]
  (loop [loc (->zip form)]
    (if-let [rslt (bcfind macros loc)]
      (recur rslt)
      (root loc))))
```

5. おわりに

本論文では、バイトコードをビジターパターンに依らず、関数型プログラミング言語（とくに Lisp）で便利に操作するためのツール `bc-macro` を提案した。`bc-macro` では、バイトコードへの操作を、一連のマクロ定義に従い構文木を変換するものとして取り扱う。しかしながら、提案するマクロ定義や木パターンの指定の仕方は、JVM バイトコード仕様とは独立しており、他のバイナリデータの操作に対しても汎用的に用いることが可能であると考えている。

また、抽象構文木も最近のほとんどのプログラミング言語が標準でそなえているベクタとマップだけを使って構築されているため、`bc-macro` の中心となるアイデアを用いた Lisp 以外の言語向けの木変換ライブラリをつくることも比較的容易と考えられる。

`bc-macro` では、典型的なバイトコード操作の多くを、シンプルで分かりやすく表現可能であるが、より本格的な高水準言語のフロントエンドの各種処理を、同様のマクロシステムで簡潔に記述可能かということは検討できておらず、今後の課題の一つと言える。ほかに、たとえばコンスタントプールの参照関係を保存した上での更新や、ジャンプ命令の飛び先オフセットの変更・追加等、現在はユーザが手動で管理せざるを得ないバイトコードの一貫性に関する様々な煩わしい処理についても、自動管理するための機能を追加したいと考えている。

参考文献

- [1] Bodden, E. and Havelund, K.: *Aspect-oriented Race Detection in Java*, IEEE Transactions on Software Engineering, 36(4), pp.509–527 (2010).
- [2] Bruneton, E., Romain, L. and Thierry, C.: *ASM: a code manipulation tool to implement adaptable systems*, Adaptable and extensible component systems 30 (2002).

- [3] Huet, G.: *Functional Pearl: The Zipper*, Journal of Functional Programming, 7(5), pp.549–554 (1997).
- [4] Kuleshov, E.: *Using the ASM framework to implement common Java bytecode transformation patterns*, Aspect-Oriented Software Development (2007).
- [5] Marek, L., Villazn, A., Zheng, Y., Ansaloni, D., Binder, W. and Qi, Z.: *DiSL: a domain-specific language for bytecode instrumentation*, In Proceedings of the 11th annual international conference on Aspect-oriented Software Development, pp.239–250 (2012).
- [6] 山本晃成, 湯浅太一: 末尾再帰の最適化と一級継続を実現するための JVM の機能拡張, 情報処理学会論文誌プログラミング (PRO) , 42(SIG11(PRO12)), pp.37–51 (2001).

付 録

A.1 木パターン文法

$$\begin{aligned}
 Context & ::= Target \\
 & \quad | VecContext \\
 & \quad | MapContext \\
 & \quad | \#nest \ Context \\
 VecContext & ::= [SeqContext \ Context \ SeqRContext \\
 & \quad \quad \quad As_{opt} \] \\
 SeqContext & ::= \epsilon \\
 & \quad | \dots \\
 & \quad | SeqContext \ ElemOrGuard \\
 & \quad | SeqContext \ ElemOrGuard \ \dots \\
 SeqRContext & ::= \epsilon \\
 & \quad | \dots \\
 & \quad | ElemOrGuard \ SeqRContext \\
 & \quad | \dots \ ElemOrGuard \ SeqRContext \\
 MapContext & ::= \{ KVSeq_{opt} \ Context \ Key \ KVSeq_{opt} \\
 & \quad \quad \quad As_{opt} \ } \\
 Target & ::= (Seq \ As_{opt}) \\
 & \quad | @ (Seq \ As_{opt}) \\
 Seq & ::= ElemOrGuard \\
 & \quad | ElemOrGuard \ Seq \\
 & \quad | \& \ Pat \\
 ElemOrGuard & ::= Elem \mid Guard \\
 Elem & ::= Val \\
 & \quad | Map \\
 & \quad | Pat \\
 & \quad | \sim Form \\
 Pat & ::= Var \\
 & \quad | Vec \\
 Vec & ::= [Seq \ As_{opt} \] \\
 Map & ::= \{ KVSeq \ As_{opt} \} \\
 KVSeq & ::= KV \\
 & \quad | KV \ KVSeq \\
 KV & ::= ElemOrGuard \ Key \\
 As & ::= :as \ Var \\
 Guard & ::= \#when \ Form
 \end{aligned}$$