Journal of Information Processing Vol.23 No.5 716–729 (Sep. 2015)

[DOI: 10.2197/ipsjjip.23.716]

Regular Paper

Design and Implementation of a Java Bytecode Manipulation Library for Clojure

Seiji Umatani[1,a) Tomoharu Ugawa[2] Masahiro Yasugi[3]

Received: November 14, 2014, Accepted: March 17, 2015

Abstract: Recently, the Java Virtual Machine (JVM) has become widely used as a common execution platform for various applications. There is often the need to manipulate bytecodes at class-load time, particularly in application domains that demand dynamic modification of program behaviors. Whereas several bytecode manipulation tools for Java exist for implementing such behaviors, JVM is also the platform for various modern programming languages, and there is no need to write bytecode manipulation programs exclusively in Java. In this paper, we propose a novel bytecode manipulation library for Clojure, a Lisp dialect running on JVM. Our library is as expressive and flexible as ASM, the de facto standard bytecode manipulation tool in Java, while enabling more concise representation of typical manipulation cases. Our library works at class-load time as a series of rewrites of (parts of) the tree representing the target class file, basically in a similar way to Lisp's macro system. However, our library differs from Lisp's macro system in the following significant respects. First, instead of matching a single name against the first item of the target form (tree), our library matches a tree pattern against the target tree itself during macro expansion so that users can define rewriting rules for raw class files that cannot contain any special tags (names) for pattern matching. Furthermore, along with matching tree patterns, our library can extract any information from the static context easily and thus allows users to avoid cumbersome manual management of such information.

Keywords: bytecode, tree transformation, macro

1. Introduction

The Java Virtual Machine (JVM) is designed such that programmers can freely manipulate Java bytecode at class loading time. Utilizing this fact, we can accomplish profiling, debugging, testing, security checking, and so on by dynamically analyzing the class files the JVM loads for execution.

In recent years, along with the popularization of several high-level programming languages running on the JVM such as Scala, Clojure, and Groovy as well as language implementations targeting the JVM such as JRuby, Rhino, and Jython, various applications that use the JVM as the execution platform have been developed. If we use a dynamic analysis tool that targets class files, we can profile, for instance, the runtime behavior of Scala programs as well as that of Java programs with a single tool.

To manipule class files, there are several Java bytecode manipulation libraries (mainly in Java) that make it easy to write various kinds of manipulations by abstracting the internal structure of class files. Major Java bytecode manipulation tools are classified into two categories. The first category contains tools that allow us to express low-level manipulations using the visitor pattern of the syntax tree representing the class file structure. Representative of this category is ASM [18], the de facto standard bytecode manipulation library. The second category contains tools based on aspect-oriented programming (AOP), which enable us to weave code fragments into specific parts of the bytecode. The readability of the code written in the former tool is relatively low, since, with the visitor pattern, a manipulation of the syntax tree that depends on multiple nodes is divided into multiple methods. Furthermore, because we must manage static context information manually, even a simple analysis or manipulation may cause a rather complicated implementation, depending on the structure of its target subtree. As for the latter category, there is a wide range of developments and studies, from a tool that directly uses AspectJ [1] to the one that uses a Domain-Specific Language (DSL) that supports many advanced features to improve its flexibility [14]. However, at the time of writing this paper, there is no tool that has a flexibility comparable to the former tools, and so it seems that their applicability is limited.

On the other hand, as mentioned previously, because the JVM is widely used as a common platform for various high-level programming languages, there is no need to write bytecode manipulations in Java or in any object-oriented paradigms. Therefore, in this paper, we propose BcMacro, a bytecode manipulation library for Clojure. In order to have both flexibility that is comparable to ASM and readability that is achieved by concise notation, BcMacro derives its core concept from the idea of Lisp macros. The rewriting of syntax trees with a macro is a more declarative means of meta-programming than a collection of node manipulations along with the traversal of a visitor. Furthermore, from a practical aspect, we consider that the gradual development of ma-

[1] Graduate School of Informatics, Kyoto University, Kyoto 606–8501, Japan

[2] Kochi University of Technology, Kami, Kochi 782–8502, Japan

[3] Department of Artificial Intelligence, Kyushu Institute of Technology, Iizuka, Fukuoka 820–8502, Japan

[a)] umatani@kuis.kyoto-u.ac.jp

nipulation code, enabled by Lisp's dynamic and interactive evaluation environment, can improve its productivity, especially for software equipped with complex and low-level bytecode manipulations. Incidentally, although BcMacro is a library used in Clojure programs, the class files that are BcMacro's manipulation targets may be compiled from source programs in any language, such as Java or Scala.

BcMacro's macro expansion (subtree rewrite) mechanism is largely different from Lisp macros, whose main purpose is to extend Lisp's syntax, in the following points. First, to specify rewritings of raw bytecode, instead of looking at the first element of each form, BcMacro's macro expansion matches a part of the syntax tree with a *tree pattern*. The syntax of BcMacro' tree pattern is flexible enough to extract an arbitrary bytecode fragment from the tree. Moreover, BcMacro's tree pattern can contain the static context of the rewrite target; with this feature, we can easily extract any static context information along with the pattern matching. This removes the burden of manually managing the static context information that is imposed if we write with the visitor pattern.

The main difference between BcMacro and other tree manipulation libraries (whose target data are often XML documents) is that BcMacro is implemented as an extension to Clojure. Syntax trees are represented using Clojure's primitive data types. Tree patterns are also written in a syntax that is compatible with Clojure's primitive literal notations. Therefore, we can analyze or manipulate syntax trees in an intuitive way using BcMacro, cooperating seamlessly with arbitrary Clojure code. Moreover, BcMacro's tree patterns are first-class Clojure objects. Therefore, capabilities such as dynamic pattern composition from multiple patterns are possible; this is also one of the sources of BcMacro's flexibility.

BcMacro's functionality that uses tree patterns to rewrite subtrees is a generic manipulation mechanism and so, in principle, can be applicable to general tree structures other than those in Java bytecode. However, in this paper, we primarily intend to design a practical tool by assuming concrete use cases that are widely known; therefore, we chose typical Java bytecode as a concrete rewrite target. Particularly, in the design of tree patterns, we concentrate on realizing concise and straightforward notations, paying less attention to more theoretical aspects.

The organization of the remainder of this paper is as follows. First, we overview the related tools and studies in Section 2. Next, we explain the design of the proposed BcMacro system in Section 3. In Section 4, we present several example BcMacro programs illustrating practical Java bytecode transformations. Next, in Section 5, we describe the implementation of BcMacro using tree automata techniques [4]. Finally, we conclude the study in Section 6.

2. Related Work

Various tools that analyze or manipulate Java bytecode have been developed. However, most of them deal with class files in the same way as the Document Object Model (DOM) of HTML or XML documents. They generate trees that reflect the structure of the class file format. ASM [18], which is the de facto standard tool, provides two data models, the Core and Tree application programming interfaces (APIs). With the Core API, the user can write a program that processes the stream of nodes of the tree that appear in the same order as depth-first traversal while the Tree API allows the user program to manipulate the tree. Although the Tree API is more relevant to the BcMacro data model than the Core API, a program that uses this API is still likely to be complicated when a complicated process depending on multiple contexts must be written. This is because the program must explicitly traverse the tree to collect context information. In contrast, with BcMacro, the user can write such a complicated process in a simple program thanks to BcMacro's declarative tree pattern that can be expanded or shrunk to an arbitrary range.

AOP based tools help analyze the dynamic behavior of Java programs [1], [14], [19], [20]. These tools provide high-level notation based on the join point model, which allows us to avoid describing low-level bytecode manipulation. Nevertheless, they are not flexible enough. For example, the set of join points are typically given a priori. DiSL [14] proposed the open join point model that allows the user to extract some information from the context at an arbitrary code point. However, such information is limited by the Java interface representing the context in advance. Moreover, though the special-purpose API allows a simpler description than low-level tools such as ASM, DiSL requires imperative description using the API for context manipulations. This is in sharp contrast to our pattern language, which provides a declarative interface for extracting context information.

With respect to tree manipulation, BcMacro shares many ideas with XML document tools and research. The main difference from these tools is that, in BcMacro, tree patterns are first-class objects of Clojure. Therefore, we can rely on various programming techniques in Clojure to describe the rewriting code of tree patterns. In the rest of this section, we discuss techniques for XML processing, focusing on additional differences.

Some tree patterns that expand the XML schema (representing the *type* of the XML document) to extract subtree by allowing us to use bound variables in the schema have been proposed [6], [7], [11]. In particular, a functional language for XML processing, XDuce [7], which has tree patterns as a built-in language facility, is similar to Clojure with BcMacro. However, there are some differences in the details of expressiveness. With XDuce in particular, we can express various *horizontal* patterns such as a sequence of child nodes sharing a common parent while *vertical* patterns such as an ancestor and descendant are limited. In contrast, BcMacro does not provide a generic regular expression, but it allows us to describe both horizontal and vertical patterns flexibly. This flexibility helps extract information from a context that is distant from the rewriting target.

XPath [3] enables us to extract arbitrary nodes by specifying a path from a starting node to the target node that may be an arbitrary distance from the starting node. In XPath, the path is represented as a sequence of explicit *movements*. XPath is operational in comparison with node extraction, i.e., binding a variable to the node, using tree patterns. Furthermore, it is difficult to extract various kinds of nodes spread over a tree by a single path because a single path can only uniformly extract nodes that match conditions represented by the path.

Kutsia [13] proposed a pattern matching technique for XML data whose expressiveness is the same as BcMacro. Particularly, Kutsia's context and sequence variables correspond to BcMacro's sequence abbreviation patterns and nest abbreviation patterns that can express sequences of siblings and ancestors of arbitrary length. However, its matching algorithm differs from BcMacro in two respects. First, BcMacro's algorithm searches for the root of the subtree that is the substitution target while Kutsia matches from the root of the tree in a top-down left-to-right manner. Second, BcMacro finds a single match based on several rules while Kutsia enumerates all matches. The reason why BcMacro's matching is target-centric is that we believe that it is practical to match the nearest candidate to the target. Because BcMacro may traverse over the tree in every direction to choose the most preferable match, BcMacro uses a more complicated algorithm. Nevertheless, both algorithms are the same in essence. Another difference is that Kutsia assumes that children of a particular node are ordered because it deals with XML documents. Although an extension to deal with unordered children is also presented in the paper, it is more efficient to deal with a BcMacro map node as a primitive data type in Clojure.

Note that the XML processing technologies Xduce [7] and XPath [3] have a strong connection to a different tree automata technique. As we mention in Section 5, BcMacro is implemented based on a combination of these two kinds of automata. Thus, BcMacro does not propose a theoretically more powerful tree automaton. Nevertheless, BcMacro provides high usability as a domain specific language because of its seamless interoperability with Clojure.

For the purpose of program transformation, many studies on higher order patterns (those that allows us to bind variables to higher order functions) have also been performed [5], [10], [22]. For example, the BcMacro pattern `[... x]`, which matches a sequence of any length and binds a variable `x` to the last element of the sequence (we explain the details later), can be described as `c [x]` in a higher order pattern [*1]. For instance, the pattern matches a sequence `[1 2 3]`, resulting in a binding of `c` to a function

```
(fn [xs] (cons 1 (cons 2 xs)))
```

that creates a context around the given subtree.

Huet and Lang [10] realized a translation of programs that contain multiple argument functions as their terms by a higher order pattern matching where a function to be bound is limited up to second order. Its matching algorithm is the same as that of Kutsia. It is a simple top-down left-to-right recursive search from the root for all matches. Moor and Sittampalam [5] enabled more complicated program transformations by allowing more higher order functions to be matched. For example, a combination of pattern matching functions (see below) of BcMacro can be expressed as a higher order pattern that takes a second order pattern as an argument. Yokoyama et al. [22] identified a condition of patterns that gives a unique solution and proposed a deterministic algorithm based on the condition rather than enumerating all solutions. This is in sharp contrast to BcMacro, which returns a unique match based on built-in rules. Note that Hu et al. [8] implemented Yicho, which is an implementation of the higher order pattern of Yokoyama et al. in Template Haskell.

Mohnen [16] expanded the pattern matching of Haskell to a higher order pattern matching, which has virtually the same functionality as the second order pattern matching of Huet and Lang [10], to deal with contexts. Unlike studies on program transformation, where the target of matching is program code, Mohnen's target is a tree constructed by an algebraic data type (ADT). (BcMacro shares this intended use.) Mohnen hence limited higher order functions to those that only apply constructors in their bodies so that variables can bind to only ADT data structures. The matching algorithm is a top-down left-to-right search and returns the first solution. Mohnen and BcMacro differ in that they have different designs, and they differ in the search order, resulting in different solutions. They are similar in that they are embedded in general programming languages. However, Mohnen expanded the pattern matching of Haskell and does not provide any functionality to deal with patterns as a first class object, such as a function object, in the host language. Furthermore, Mohnen only supports up to second order patterns. This makes it difficult to write complicated programs such as combinations of patterns.

Most of these studies on higher order patterns assume statically typed languages. This is in sharp contrast to BcMacro, which is built on the dynamically typed language Clojure. For example, the reason why the variable `c` in the previous example `c [x]` is bound to the function creating a flat list is that `c` is limited by type to functions that take a list of elements of a uniform type and return a list of the same type. If we applied the same higher order pattern to Clojure, which allows heterogeneous lists, for example, a matching to the value `[1 2 [3]]` would succeed with a binding of `c` to

```
(fn [xs] (cons 1 (cons 2 (cons xs nil))))
```

We also believe that the domain specific pattern notation of BcMacro, which does not use types not explicitly appearing in the patterns, is more intuitive and friendly to users than function application terms. Nonetheless, the functionality of higher order patterns that extract contexts as functions is interesting. It could be worth considering for bytecode manipulations.

3. Design

We first present an overall picture of the BcMacro system, the proposed bytecode manipulation library for Clojure, by describing its general usage.

Users express various manipulations to the abstract syntax tree (AST), which is the internal representation of a class file (Section 3.1), by writing (typically multiple) macro definitions. Macro definitions of BcMacro (Section 3.3) specify the rewriting of a part of the target AST in essentially the same style as Lisp's macro definitions. That is, each definition contains a part that identifies its rewrite target (called the *target subtree*) and the Clojure code that constructs the tree put in place of the target subtree (called the *expansion tree*).

In a Lisp macro definition, its rewrite target is specified by its macro call form that includes names bound to its arguments. On

[*1] Here, we regard a vector of Closure in the same light as a list of `cons` and `nil` for simplicity.

the other hand, in a BcMacro macro definition, its rewrite target is specified by a tree pattern (Section 3.2) that represents the shape of the target subtree.

In Lisp, if the source code contains a macro definition, macro expansions based on this definition are done automatically at compile-time. On the other hand, in BcMacro, from each macro definition, a special Clojure function called a *macro expansion function* is generated; it is manually accessible to users. In typical BcMacro use cases, however, macro expansion functions are automatically and repeatedly applied to the AST of a class file at class loading time until no more expansion is possible. After that, the processed AST is executed, passing through the normal JVM verifier, and so on. For other cases, BcMacro provides the means to manually apply a set of macro expansion functions explicitly specified by the user to a specific part of the AST (Section 3.4).

Moreover, in BcMacro, we can generate a function (called a *pattern matching function*) that executes pattern matching only (i.e., without rewriting the target) from a macro definition (Section 3.5). We can express complicated patterns by composing multiple pattern matching functions, and thus can handle processes that are difficult to describe in a simple macro expansion function.

3.1 Data Model

Unlike tools based on object-oriented languages such as ASM, BcMacro represents the AST of a class file using only the data types primitively supported by Clojure, the host language of BcMacro (**Fig. 1**).

First, integers and strings in a class file appear at the AST's leaves as objects of Clojure's corresponding primitive types. Next, BcMacro uses only vectors and maps to represent intermediate nodes of the AST; it specifically does not use lists or other types of sequences for this purpose. The reason for this restriction is that the class file format is defined only in terms of arrays and records in the JVM specification [15], and Clojure's primitive vectors and maps, respectively, are enough to represent such kinds of JVM data.

The square brackets in the figure represent a vector intermediate node and, between the vector and each of its elements (i.e., its child nodes), there is an edge labelled with the corresponding index. The curly brackets in the figure represent a map intermediate node and, between the map and each of its elements (i.e., its child nodes), there is an edge labelled with the corresponding key. Note that the name of each key in the AST completely coincides with the name of the corresponding field in the JVM specification, and so there is no need to remember tool-specific naming conventions.

Unlike XML, which is widely used for processing tree structures, there is no ordering among the children of a map node in BcMacro's data model [*2]. Therefore, each key of the map nodes is represented, not as a single node having its associated value as its own child, but as a label of the edge connected to its associated value. Employing such a data model is the major reason for BcMacro to use the original pattern expressions described in the next section instead of the standard regular expressions for the sequence of children, as in the XML patterns or schema described in Section 2.

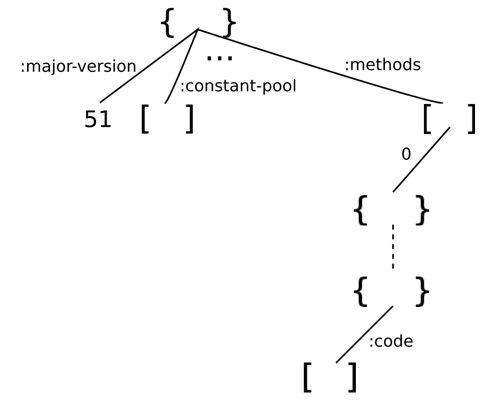

Fig. 1 AST of Java bytecoe.

Moreover, because there is no ordering among siblings, our implementation of BcMacro (described in Section 5) cannot transform ASTs, which are unordered multi-branch trees, into (ordered) binary trees with the first-child/next-sibling references commonly used in XML processing.

3.2 Tree Patterns

In this section, we explain *tree patterns*, the notation used in BcMacro for specifying an arbitrary subtree of the AST. As described briefly in Section 1, each tree pattern consists of the part corresponding to its target subtree (*target pattern*) and the part corresponding to the static context (in the remainder of the paper, simply referred to as the context) enclosing the target subtree (*context pattern*). The entire tree pattern successfully matches the target subtree if and only if the following two submatchings succeed:

(1) Its target pattern matches the target subtree in a top-down manner.

(2) Its context pattern matches the context in a bottom-up manner.

(The exact meanings of "top-down" and "bottom-up" are explained in Sections 3.2.1 and 3.2.2, respectively.)

Each target pattern and context pattern has its own set of variations and notations, and hence we explain them separately in order. The complete BNF grammar of the tree pattern syntax is shown in **Fig. 2**, where *Key*, *Var*, *Form*, and *Val* in the grammar represent Clojure keywords, variables, forms, and literal values, respectively.

3.2.1 Target Pattern

There are six kinds of target patterns: variable, constant, unquote, vector, sequence, and map patterns. Each pattern has its own concise notation. In what follows, we explain their meanings using some simple examples.

Variable pattern: A variable pattern is represented by a Clojure symbol and matches an arbitrary subtree. Variables in the pattern of a macro definition can be accessed from the macro body. For example, if a macro expansion function is defined

[*2] Of course, there is some ordering among them in the class file representation. However, since permutating them does not affect the meaning of the program, we can regard their ordering as insignificant.

Context	::=	*Target*
	|	*VecContext*
	|	*MapContext*
	|	`#nest` *Context*
VecContext	::=	`[` *SeqLContext Context SeqRContext*
		As_{opt} `]`
SeqLContext	::=	ϵ
	|	`...`
	|	*SeqLContext Elem*
	|	*SeqLContext Elem* `...`
SeqRContext	::=	ϵ
	|	`...`
	|	*Elem SeqRContext*
	|	`...` *Elem SeqRContext*
MapContext	::=	`{` $KVSeq_{opt}$ *Context Key* $KVSeq_{opt}$
		As_{opt} `}`
Target	::=	`(` *Seq* As_{opt} `)`
	|	`@(` *Seq* As_{opt} `)`
Seq	::=	*Elem*
	|	*Elem Seq*
	|	`&` *Pat*
Elem	::=	*Val*
	|	*Map*
	|	*Pat*
	|	˜*Form*
	|	`#when` *Form*
Pat	::=	*Var*
	|	*Vec*
Vec	::=	`[` *Seq* As_{opt} `]`
Map	::=	`{` *KVSeq* As_{opt} `}`
KVSeq	::=	*KV*
	|	*KV KVSeq*
KV	::=	*Elem Key*
As	::=	`:as` *Var*

Fig. 2 Grammar of tree pattern.

as in the following code[*3], `(dup [1 2 3])` is evaluated as `[[1 2 3] [1 2 3]]`:

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

Furthermore, a variable in a pattern can be accessed from within the same pattern that contains that variable. However, the extent of such accesses are limited only after the accessed variable matches some subtree. More specifically, in vector patterns and map patterns, which contain subpatterns for their child elements, the ordering of tree traversal for pattern matching is defined strictly; therefore, it is possible to access a variable only if there is a before/after relation between the matching of the variable and its access.

Incidentally, as in patterns of Clojure, ML, Haskell, and so on, multiple occurrences of a variable pattern of the same name in a single tree pattern are not allowed.

Constant pattern: Any Clojure constant literal can be used in a tree pattern. It matches a node whose value is equal to that literal. For example, the following macro:

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

matches the tree consisting solely of an integer 3.

Unquote pattern: Sometimes we want to use Clojure's features in a tree pattern, such as accessing Clojure variables from inside the pattern. In such cases, we can write ˜*f* in the pattern. Here, *f* denotes an arbitrary Clojure form and, if it evaluates to a value that is allowed to occur in the pattern, such as a numeral value and a string, the value is embedded there. For example, the previous macro `three` can also be defined as follows:

```
(def two 2)
(defbcmacro three (˜(inc two)) ...)
```

Furthermore, if *f* evaluates to a pattern matching function or macro expansion function, the involved patterns are composed. We explain the details of pattern composition in Section 3.5.

If *f* evaluates to some value other than the above, it is an error.

Vector pattern: Pattern `[`p_0 p_1 `...` p_{n-1}`]` matches a vector of length n if each subpattern p_i matches the vector element at the corresponding index i. For example, with the following macro:

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

a vector of length four is transformed to a vector containing only the first and the third elements.

Furthermore, as in Clojure's destructuring, we can match the pattern *p* of `[ ... &` *p*`]` with a vector that contains tail elements of the target vector, starting from a certain position.

Pattern matching a vector pattern is performed by traversing the target tree in a top-down manner, that is, in a depth-first and left-to-right order. This ordering is one of the precedence rules of BcMacro, defined to ensure a single match solution, that is, resolving the possibility of several solutions to occur in a pattern matching (i.e., the ambiguity of the pattern). According to this ordering, the before/after relation between any pair of subpatterns is fixed. Therefore, in a subpattern, we can refer to variables bound in other subpatterns before it. For instance, the pattern `[x ˜x]` matches a vector of length two whose elements are the same.

Sequence pattern: We sometimes want to replace a subsequence of a vector with some other sequence instead of replacing the entire vector. In such cases, we can express the replacement target simply as a sequence of patterns. The sequence of patterns matches the target if the first pattern of the sequence matches the target subtree and the remaining patterns match the right siblings (subtrees) in the correct order. For example, the macro:

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

matches element 1 of the tree `[0 1 2 3 4]`.

Map pattern: Pattern `{`p_0 k_0 `...` p_{n-1} k_{n-1}`}` matches a map that contains key/value pairs, each of which matches key/pattern pair `[`p_i k_i`]` of pattern ($i = 0, \ldots, n-1$). For example, `{51 :major-version x :minor-version}` matches any map containing the value 51 associated with key `:major-version` and also containing some value associated with key `:minor-version`. Note that the map can contain other values associated with other keys.

Unlike vector patterns, because there is no ordering among key/value pairs of a map, there is no before/after relation of tree traversal for map patterns.

BcMacro also has two extra target patterns that can be included only in vector patterns or map patterns, the as and guard patterns.

An as pattern is written as `[... :as x]` or `{... :as x}`, and the variable `x` is bound to the entire target subtree matching the pattern.

A guard pattern is written as `#when` *pred* and can be embedded in an arbitrary element position of a vector or map (for maps, only in a value position). A pattern containing a guard pattern as its component can match only if *pred* evaluates to true (in Clojure,

[*3] The exact specification of `defbcmacro`, which defines a macro expansion function, and the exact meaning of calling a macro expansion function is explained in Section 3.3 and Section refsubsec:expansion, respectively.

every value other than `nil` or `false` is regarded as true).

Note that the timing of the evaluation of *pred* differs between vector patterns and map patterns. For a vector pattern, its guard pattern is evaluated according to the precedence rule defined for normal element subpatterns; that is, its before/after relation is determined based on the position at which it is embedded in the vector. For a map pattern, although there is no traversal ordering among its key/value pairs, as mentioned previously, its guard pattern is evaluated after the entire traversal of the corresponding map is complete so that we can refer to variables bound elsewhere in the map pattern to write the predicate `pred`.

Effective usages of the as and guard patterns are illustrated in Section 4.

3.2.2 Context Pattern

A context pattern is used to express the parent node or ancestor node (that is a vector or map) of a target subtree. Its notation is essentially the same as that of the vector or map patterns of the target pattern in the previous section. For example, the macro:

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

expands `[0 1 2 3 4]` into `[0 6 4]`. Here, the target pattern inside the context pattern is enclosed by a pair of parentheses, which acts as a boundary between the static context and replacement target[*4]. In BcMacro, each macro expansion replaces only one part of the tree, and there is exactly one target pattern enclosed with parentheses in the whole tree pattern.

When replacing a target subtree with a new tree, we sometimes want to splice it, like the unquote-splicing of Lisp macros. In such cases, parentheses are prefixed with `@`. For example, the macro:

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

expands `[0 1 2 3 4]`, not into `[0 [I II III] 4]`, but into `[0 I II III 4]`.

Tree traversal of a context pattern is performed in a bottom-up and breadth-first manner, contrary to that of a target pattern. Moreover, if the context pattern is a vector, the traversal is performed starting at the target pattern, first by matching each of the left-sibling subpatterns with the corresponding vector element right-to-left, then by matching each of the right-sibling subpatterns with the corresponding vector element left-to-right. For example, in the following context pattern:

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

`a`, `b`, and `c` are matched in that order, and then `x` and `y` are matched. Note that there is no before/after relation between the matchings of `x` and `y`.

In addition to the above patterns, we can write the following two extra patterns in context patterns.

Sequence abbreviation pattern: By placing a token `...` at an arbitrary position in a vector pattern, we can skip any number of elements (including zero). The skip is done according to vector's traversal order until an element that matches the subsequent pattern of `...` (if any) is found. For example, with the pattern:

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

after matching the target pattern `(t)`, the rightmost element of the left siblings that matches `{_ :foo}` is searched for. The variable `y` is then bound to the first element. The `...` to the right of the target pattern, which has no subsequent pattern in the vector pattern, means that the matched vector can have any number (and shape) of elements as right siblings of the target subtree.

Nest abbreviation pattern: By prefixing an arbitrary subpattern (target or context pattern) of a context pattern with `#nest`, we can specify that the prefixed subpattern is not necessarily an immediate child, but can be a descendant at an arbitrary depth.

For example, the pattern `{ #nest[1 (x) y] :foo}` matches not only `{:foo [1 2 3]}` but also `{:foo [100 {:bar [1 2 3]} 200]}`.

Just as multiple sequence abbreviation patterns can appear in a single vector pattern, multiple nest abbreviation patterns can be nested.

Using these abbreviation patterns, we can express the tree pattern in Fig. 1 concisely as:

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

Because each abbreviated part of the figure $(\cdots)$ is straightforwardly mapped to a `...` or `#nest` pattern, we find that the intuitive shape of the tree structure is naturally expressed in the above tree pattern.

3.3 Macro Definition

In addition to `defbcmacro`, which was used in the previous sections, BcMacro provides some other ways to define macros.

(1) `(bcmacro` *pattern body*`)`
(2) `(defbcmacro` *name pattern body*`)`
(3) `(letbcmacro [(`*name pattern mbody*`)`$^{+}$`]` *lbody*`)`

In BcMacro, (1) is used for generating an anonymous macro expansion function, (2) is used to define a global macro, and (3) is used to define (possibly multiple) local macro. We can write any Clojure code for the macro body (*body* or *mbody*) and, when macro expansion is executed on a tree, the target subtree matched with *pattern* is replaced with the evaluation result of the body. The detailed usage of macros, defined in the above three ways, is explained in the next section.

3.4 Macro Expansion

Macro expansion functions, defined (or generated anonymously) by the methods of the previous section, can be used for macro expansion by calling them as ordinary Clojure functions. Every expansion function accepts as its sole argument the location of the replacement target in the tree (the concrete representation of this location is described in Section 5.3). If its pattern matches the subtree at the argument location, it replaces the subtree and then returns the same location as its result. Otherwise, it throws an exception.

The above method of macro expansion is inconvenient because we must specify the location of a target subtree explicitly. Therefore, BcMacro provides two functions that, as Lisp macros, automatically search for a subtree (subtrees) matching the pattern of the defined macros and replacing it (them).

[*4] The fact that all patterns in the previous section are enclosed with parentheses means that their context patterns are empty.

- (bcexpand-1 *tree*): traverses *tree* in depth-first order, macro-expands the first matching subtree, and returns the entire tree. If no subtree matches the defined macros, it returns *tree* without any change.
- (bcexpand-all *tree*): traverses *tree* in depth-first order, macro-expands all matching subtrees, and returns the entire tree. If no subtree matches the defined macros, it returns *tree* without any change.

When we use bcexpand-all, we must be careful to ensure that its expansion process does not loop infinitely. For instance, after several macro expansion processes, a pattern may match a part of its own expansion subtree, depending on its shape. Therefore, in BcMacro, we can suppress macro expansion of an arbitrary node by adding specific metadata to it. More specifically, if we construct the expansion subtree *tree* by (with-aside [*mfun*$^+$] *tree*), *tree* is removed from the set of target nodes of the macro expansion functions *mfun*$^+$.

Furthermore, aside from controlling the number of expansions, BcMacro provides the means to limit the set of macros used for a macro expansion. First, if we call bcexpand-{1,all} with a single argument, the set of used macros is limited to those bound by vars of the current namespace (*ns*), that is, in Clojure's top-level environment. Therefore, common modularization techniques using Clojure's namespaces can also be applied to BcMacro programs.

Moreover, we can limit the set of used macro expansion functions by explicitly passing it as an additional argument to bcexpand-{1,all}, as in (bcexpand-{1,all} [*mfun*$^+$] *tree*). By using this feature in combination with letbcmacro, which localizes macro definitions, we can further improve the modularity of BcMacro programs and obtain more flexible control over macro expansions.

3.5 Composition of Patterns

In previous sections, we use a tree pattern only for generating a macro expansion function, for instance, as an argument to the function bcmacro. BcMacro also provides the function (bp *pattern*), which generates a pattern matching function, i.e., a Clojure function that performs pattern matching of *pattern* only.

A pattern matching function accepts a target subtree as an argument and returns a truth value. If its pattern matching succeeds, it returns true; otherwise, it returns false.

We typically use a pattern matching function by embedding it into another pattern using the unquote pattern explained in Section 3.2.1. More specifically, as for an unquote pattern of the form ˜*f*, if *f* evaluates to a Clojure function, BcMacro regards it as a pattern matching function and attempts to match it with the subtree (and its context) located at the position corresponding to the unquote pattern.

For example, if we want to dynamically change replacement targets according to situations, the embedding of a pattern can be conveniently used in combination with Clojure's function composition facility as follows:

```
(def pat1 (bp ...))
(def pat2 (bp ...))

(defn dyn-cond [t]
  (cond
    <situation1> (pat1 t)
    <situation2> (pat2 t)
    :else (and (pat1 t) (pat2 t))))

(bcmacro [(~dyn-cond) ...] ...)
```

As another example, if we want to define several macros, all of which have a common context pattern, by defining the context pattern separately using bp, we can improve the code's reusability.

Furthermore, in BcMacro, macro expansion functions generated by, for instance, bcmacro in Section 3.3 can also be embedded in patterns. More specifically, as for an unquote pattern of the form ˜*f*, if *f* evaluates to a macro expansion function[*5], BcMacro attempts a macro expansion; that is, BcMacro attempts to match *f* with the subtree (and its context) located at the position corresponding to the unquote pattern and furthermore replaces it with the expansion subtree. Note that an exception thrown from within the macro expansion function is propagated to the pattern matching of the enclosing pattern.

The form *f* of pattern ˜*f* can be a call to a Clojure's higher-order function, which accepts a pattern matching function (or a macro expansion function) as its argument and returns another pattern matching function (or another macro expansion function). Such a use of the unquote pattern provides a functionality similar to Lisp macros at pattern level. More precisely, a Lisp macro does the following:

> After replacing each macro call in a program with the corresponding macro-expanded code, the entire program is executed.

Meanwhile, BcMacro's unquote pattern does the following:

> After replacing each ˜*f* in a pattern with the pattern matching function (or the macro expansion function) that *f* returns, the entire pattern is used for pattern matching.

The above contrast highlights the similarity between them.

For example, the condition "the target subtree does *not* match this pattern" can be embedded in a pattern as follows:

```
(defn pat-not [pat] (fn [t] (not (pat t))))
(bcmacro [(x) ~(pat-not (bp 1)) ...] ...)
```

This pattern binds x to the first element of a vector whose second element is not 1.

As another example, we can define the #when pattern presented in Section 3.2.1 as a Clojure macro:

```
(defmacro pat-when [pred]
  `(fn [t] ((bcmacro (_) ~pred) t)))
(bcmacro
  [(x) ~(pat-when (> x 0)) ...] ...)
```

where, although bcmacro in pat-when evaluates predicate pred

[*5] In order to distinguish macro expansion functions from normal functions, BcMacro attaches the metadata {:bcmacro true} to each macro expansion functions.

as its expansion subtree, the tree is actually used as the return value of the pattern matching function generated by the enclosing fn; that is, it is used as a truth value representing the success or failure of the pattern match.

4. Manipulation of JVM Class Files

In the previous section, we explained the usage of the BcMacro library with some simple examples that are irrelevant to Java bytecode. In this section, we first show more practical examples by rewriting typical examples of Java bytecode manipulation described in the ASM paper [12] into BcMacro programs. Though these examples seem to be written under the assumption that the target bytecode is mainly generated by the Java compiler, we may suppose that bytecodes generated from programs in other languages that use features equivalent to Java have similar structures. Incidentally, because the Java programs in Ref. [12] frequently call methods defined elsewhere, a simple line-by-line comparison between ASM and BcMacro is impossible.

Secondly, we extract the implementation code related to bytecode translation from Ref. [21], which studies the extension of JVM with tail call optimization, and show that these translations can be easily written in BcMacro.

4.1 Simple Transformations

Addition of an Interface

Adding an interface to a class's implementing interfaces is achieved by the following code.

```
(defn constant-utf8 [v]
  {:kind  :cp-info
   :tag   1
   :value v})

(defn constant-class [idx]
  {:kind       :cp-info
   :tag        7
   :name-index idx})

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

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

The function add-interface takes as its arguments cfile, the AST representing a class file, and iface, the name (string) of an added interface. It binds cf to the map representing the whole class file using an as pattern, and also extracts only necessary fields with a map pattern. When it matches to a class file, it adds iface at the tail of cp, a field representing the constant pool, and also modifies other fields related to the constant pool and the interfaces (cpc, ifc, and if) appropriately. While constructing a new map changed by these additions/modifications, it updates only the affected fields by calling assoc with cf.

In the above process, the addition of an interface at the tail of cp is done by the auxiliary function add-class-info, which relies only on vector manipulation of pure Clojure. Furthermore, the function constant-utf8 and constant-class are simple auxiliary functions for constructing maps of the appropriate form, each representing an element of the constant pool.

In this example, we must simultaneously change multiple fields of the map that represents a class file, and that is why we enclose the entire map in parentheses as the rewrite target. Moreover, because we know that it is sufficient to rewrite a class file only once, we use bcexpand-1.

We can also write the addition of a field or method to a class in a similar way.

Replacement of Field Access

The field access instruction getfield has an index into the constant pool as its operand, and the constant pool entry at that index is the name (and the type) of the target field. The replacement of the operand old with the field new is achieved as follows. In this code, we assume that new is the valid index of some field. If new indicates a non-existent field, we may instead add it to the constant pool in the same way as the addition of an interface.

```
(defn replace-field-access
  [cfile old new]
  (letbcmacro
   [(replfa [_ :getfield (˜old)]
      (with-aside [replfa] new))]
   (bcexpand-all [replfa] cfile)))
```

The function replace-field-access replaces every occurrence of getfield whose operand is old with a getfield whose operand is new. In this example, since each occurrence of such a getfield is the replacement target of the macro, we use bcexpand-all along with with-aside.

Note that we can write the above code by resorting only to the knowledge of the AST representation of the replacement target, that is, each bytecode instruction has the form [⟨address⟩ ⟨opcode⟩ ⟨operand$_1$⟩...]. With the help of BcMacro's powerful pattern matching facility, we can avoid writing code that does not directly correspond to the intent of replace-field-access.

We can also write the replacement of a method invocation in a similar way.

Addition of Code at Method Entry

The following code adds the sequence of instructions code at the beginning of the method body of each method in a class file.

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

Here, we specify only the part of interest (the head of the instruction sequence) in the pattern while abbreviating the other part using the `...` pattern. Furthermore, we can express the insertion of `code` in a simple manner using the `@` prefix.

However, in the above code, if there is a jump instruction in the instructions that matches the `...` pattern, its target address becomes inconsistent, separated from the right address by the length of `code`. An improved macro that handles this situation can be defined as follows. (We omit the definition of `adjust-offset` for the simplicity. It is implemented with a simple use of `map`.)

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

Addition of Code at Method Exit

In a similar way to the addition at the method entry, we can add an arbitrary sequence of code just before each return instruction.

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

With the help of the `...` pattern and the vector pattern containing `:return`, we can specify the insert position of `code` in a simple manner.

Furthremore, in this example, if we want to adjust the target address of the jump instructions, we can apply the same process as `insert-begin-2` only to the second `...` pattern.

Replacement of Method Body

In order to replace the body of the method named `mname` with `code`, we first consider the following naïve macro definition.

```
(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)))
```

In the class file, the name of each method is included in the constant pool (of type vector) as a string, and its method body, which is included in the `:methods` field, only contains the index of the corresponding constant pool entry. Therefore, in the predicate form of the `#when` guard that is used to search for a method declaration whose method name is `mname`, the code compares the name extracted from the vector `cp`, using the index bound to the pattern variable `index`, with `mname`.

In this way, BcMacro's `#nest` pattern is useful for describing a process that needs multiple data that are placed in locations distant from each other in the tree. Furthermore, because each method declaration has a complex structure, it uses another `#nest` pattern to extract the bytecode instructions of the method body.

However, as described in the previous section, variable bindings in context patterns occur in the inner-first order, and the variable `cp` in the `#when` predicate form is actually still unbound when it is evaluated. To solve this problem, we can separate the process into two macro definitions as follows.

```
(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))]
    (bcexpand-1 [get-cp] cfile)))
```

This code first extracts the constant pool using the macro in the function `replace-method`, and then calls the function `replace-method-aux`, which contains the macro for replacing the method body with the extracted constant pool as an additional argument. Because BcMacro's functionality is embedded in Clojure, seamless cooperation between BcMacro macro definitions and raw Clojure code is possible. That is why, as the above example illustrates, we can change the tree manipulation process flexibly by slightly modifying the coding style of the raw Clojure code.

4.2 Tail Call Optimization

In this section, among the proposed implementation methods of Ref. [21], we re-implement only the part that is directly related to Java bytecode translation.

First, to explain the fundamental approach, we consider the following simple macro that detects and replaces a tail call instruction.

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

In the above pattern, an occurrence of the `invokestatic` instruction followed by `ireturn` is regarded as a tail call, and in that case, the opcode of that occurrence is replaced with the special-purpose instruction `tailinvokestatic`. Note that we assume that the return type of the method called by this `invokestatic` instruction is `int`.

Next, we present an implementation that considers the existence of exception handlers.

```
(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)))
```

For each exception handler, the start and end addresses of the bytecode between which the handler is active are stored in the exception table `ext`. By comparing `addr`, the address of the `invokestatic` instruction, with these addresses, we can check that the exception handler is active for this `invokestatic` instruction. If it is active, this instruction is not a tail call and thus not replaced.

Finally, we present an implementation that considers the return type of the method called by `invokestatic` instructions.

```
(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)))
```

We can reach the signature information of the method called by an `invokestatic` instruction by dereferencing indirect references (indices) into the constant pool several times, starting from the index operand `idx` of the `invokestatic` instruction. We can obtain the return type of the method by simply looking at the last character of the method signature of type string. These operations can be easily done with the raw Clojure code of the macro body.

As the examples of this section illustrate, in the BcMacro system, users can choose either of two coding styles: one in which a rewrite operation is entirely defined in a single macro definition, or another in which we first narrow down rewrite targets using a rough pattern and then perform more involved data manipulations such as the extraction of node information and checking of more refined conditions for node properties in normal Clojure code. On the other hand, with visitor patterns, for instance, users do not have such a choice and are forced to write the operation as a collection of separated visit methods for tree nodes.

5. Implementation

As mentioned in the comparison with XML technologies in Section 2, a BcMacro tree pattern is implemented by translating it into Clojure code that behaves as a combination of two kinds of tree automata (more precisely, it also includes some BcMacro-specific extensions). Macro expansion functions perform pattern matching by executing the translated Clojure code against ASTs that are taken as their arguments.

Actually, translation of tree patterns into Clojure code is implemented as a two-level translation phase mediated by an intermediate language that models the tree automata in a straightforward manner. First, in Section 5.1, we describe the specification of the intermediate language. Next, in Sections 5.2 and 5.3, we explain translation from tree patterns to intermediate code fragments and translation from intermediate code fragments to Clojure code, respectively. Furthermore, in Section 5.4, we explain the implementation of macro expansion functions such as `bcexpand-all`.

5.1 TA-based Intermediate Language

The tree-automata (TA) based intermediate language used internally in BcMacro is based on a behavior model that combines features of both tree-walking automata [*6] used for the implementation of XPath [2], and the marking tree automaton, used for implementation of XML patterns [17].

A tree-walking automaton repeatedly operates two actions: it checks whether a certain constraint (such as "Its label name is ...." or "It is the root of the entire tree.") is satisfied by the current tree node (called the *current location*), and one-step movements from the current location to its parent/child node or adjacent siblings. It begins to operate with its initial node as the current location and with its initial state as the current state. It repeats the tests and movements specified in its set of transition rules until it enters one of its final states or gets stuck. The node at which it enters a final state is defined as the extracted node of the automaton. It is known that each path expression of XPath exactly corresponds to a set of transition rules.

Because the syntactical structure of each tree pattern of BcMacro is recursive, it seems that there is no natural correspondence between a tree pattern and the sequence of sequential operations of a tree-walking automaton. However, we can create a correspondence between them by decomposing the whole tree pattern into primitive patterns and sequentially executing the set of sequential operations corresponding to these primitive patterns. That is, while matching a tree pattern with some subtree of a tree, we can construct a tree automaton whose initial location is the root of the subtree and whose transition rules encode the actions such that the tree automaton walks through a part of the tree while testing whether each subpattern (included in the target pattern or the context pattern) can be matched with the appropriate

[*6] Although the paper calls it the caterpillar automaton, it is essentially a tree-walking automaton.

Table 1 Intermediate language instructions.

Instruction form	Explanation
`(:var` *x*`)`	marks the current location with *x* (i.e., binds a variable *x* to the subtree rooted at the current location).
`(:left)`	moves to the left sibling.
`(:right)`	moves to the right sibling.
`(:vec-down)`	tests whether the current node is a vector and, in that case, moves to the leftmost child.
`(:vec-up)`	tests whether the current node is an element of some vector and, in that case, moves to the parent.
`(:map-down` *key*`)`	tests whether the current node is a map and, in that case, moves to the child corresponding to *key*.
`(:map-up)`	tests whether the current node is an element of some map and, in that case, moves to the parent.
`(:reset)`	moves the current location to the initial node (i.e., the root of the target subtree).
`(:val` *v*`)`	tests whether the subtree rooted at the current location is equivalent to the constant *v* using Clojure's `equal` function.
`(:unquote` *exp*`)`	tests whether the subtree rooted at the current locaion is equivalent to the result value of *exp* using Clojure's `equal` function.
`(:guard` *exp*`)`	tests whether the result value of *exp* is `true` in the environment including `:var` bindings so far.
`(:no-left?)`	in the assumption that the current node is an element of some vector, tests whether its left sibling does not exist.
`(:no-right?)`	in the assumption that the current node is an element of some vector, tests whether its right sibling does not exist.
`(:some-left` *s*`)`	in the assumption that the current node is an element of some vector, tests whether there is another element matching the instruction sequence *s* in the set of its left siblings.
`(:some-right` *s*`)`	in the assumption that the current node is an element of some vector, tests whether there is another element matching the instruction sequence *s* in the set of its right siblings.
`(:nest` *s*`)`	tests whether there is a node matching the instruction sequence *s* while walking toward the root of the entire tree.
`(:remove-and-left)`	executes the zipper function `remove-and-left`.
`(:merge-rights)`	executes the zipper function `merge-rights`.

node.

A marking tree automaton walks through the tree in the same way as an ordinary top-down/bottom-up tree automaton, and additionally may mark arbitrary nodes, each with an element of the set of variables as needed. When the automaton terminates, each variable marked at some node in the tree is bound to the subtree rooted at this node. In BcMacro, this marking functionality is used for implementing bindings of variable patterns and as patterns.

The set of BcMacro's intermediate language instructions, whose design is based on the above tree automata, is listed in **Table 1**.

The set of tests that can be applied to the node of the current location is limited to those that are sufficient to realize the tree patterns. Each of these tests is implemented as a Clojure function that tests a particular property of vectors and/or maps.

Instructions `:some-left`, `:some-right`, and `:nest` are used for implementing abbreviation patterns of context patterns. They could also be implemented using regular expression paths of the path expressions proposed in Ref. [2] (not included in the standard XPath) in principle. However, for simplicity of implementation, we implemented these instructions directly as Clojure recursive functions.

The zipper functions executed in the last two instructions are explained in Section 5.3 and their usage is described in Section 5.2.

5.2 Translation from Tree Pattern to Intermediate Code

We describe the translation from various kinds of tree patterns to sequences of intermediate language instructions by showing some typical examples.

For example, the pattern `([1 2 x y & r])` is translated to the following sequence of instructions:

```
[(: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)]
```

This code first moves to the leftmost element of a vector by `(:vec-down)` and then tests whether the element is equal to `1` by `(:val 1)`. Next, it moves to the right neighbor and performs a similar test. A variable pattern, such as `x`, is bound by `(:var x)`. Because a sequence of elements after `&` must be bound to the variable `r` as a single vector, it executes `(:merge-rights)` just before the instruction `(:var r)`. After traversing the elements of the vector from left to right in this way, it finally returns to its start position by executing `(:remove-and-reft)` and `(:vec-up)`.

The pattern `{t :tag, (mj) :major, [m n] :minor}`, which contains a context pattern, is translated to a combination of the following sequence of instructions:

```
[(:var mj)]
```

which performs pattern matching with the target subtree, and the following sequence of instructions:

```
[(:map-key :major) (:map-up) (:map-down :tag)
 (:var t) (:map-up) (:map-down :minor)
 (:vec-down) (:var m) (:right) (:var n)
 (:remove-and-left) (:vec-up) (:map-up)]
```

which performs pattern matching with the context. As described in Section 3, pattern matching of a context pattern is done in breadth-first and bottom-up, and its instruction sequence is arranged as above.

5.3 Translation from Intermediate Code to Clojure Code

A sequence of intermediate language instructions is further translated to the Clojure code fragments whose behavior directly corresponds to the intuitive explanations of Table 1. In order to realize one-step walks towards arbitrary directions (parent, child, and left/right siblings), the translated Clojure code uses the zipper functions [9] at runtime. A zipper is a data structure that we use to focus on a particular tree node at every moment. Using it, we can efficiently perform various tree manipulations (such as insertion, deletion, and replacement of subtrees) around the node of focus in functional programming style. Furthermore, we can efficiently

Table 2 Zipper functions.

Function form	Explanation
(lefts *loc*)	in the assumption that *loc* is an element of some vector, returns the sequence of its left siblings.
(rights *loc*)	in the assumption that *loc* is an element of some vector, returns the sequence of its right siblings.
(key *loc*)	in the assumption that *loc* is an element of some map, returns the key that is associated with the element.
(left *loc*)	in the assumption that *loc* is an element of some vector, returns the location of its left sibling. If the left sibling does not exist, it returns `nil`.
(right *loc*)	in the assumption that *loc* is an element of some vector, returns the location of its right sibling. If the right sibling does not exist, it returns `nil`.
(down *loc*)	in the assumption that *loc* is a vector, returns the leftmost element. If the vector is empty, it returns `nil`.
(down *loc* *key*)	in the assumption that *loc* is a map, returns the location of the element that is associated with *key*.
(up *loc*)	returns the location of the parent node of *loc*. If *loc* is the root of the entire tree, it returns `nil`.
(replace *loc* *node*)	replaces the subtree rooted at *loc* with *node* and returns the location of *node*.
(replace-splicing *loc* *node*)	in the assumption that *loc* is an element of some vector, replaces the subtree rooted at *loc* with a vector *node* by splicing it. It returns the location of the leftmost element of *node*.
(merge-rights *loc*)	generates a new vector consisting of the subtree rooted at *loc* and its right siblings, and replaces the subtree rooted at *loc* with the vector. It returns the same location as the argument.
(remove-and-left *loc*)	removes the subtree rooted at *loc* and returns the location of its left sibling (or `nil` if it does not exist).
(->zip *node*)	takes *node*, the root node of some tree, turns the tree into a zipper, and returns the location of the root node.
(node *loc*)	returns the subtree rooted at *loc*.
(root *loc*)	repeatedly calls `up`, starting from *loc* to the root, and returns (the root node of) the entire tree.
(next *loc*)	returns the next location of *loc* in the order of the depth-first traversal of the entire tree.
(end? *loc*)	returns `true` if *loc* is the last location in the order of the depth-first traversal of the entire tree. Otherwise, it returns `false`.

move the node of focus towards arbitrary directions.

In fact, Clojure supports zipper functions as a part of the standard library. For the purpose of implementing BcMacro, the functionality of the standard library for vector nodes is sufficient. On the other hand, for map nodes, the standard library creates the corresponding zipper structures by first transforming each map into a vector of two-element key/value vectors; such a transformation makes it difficult for BcMacro to treat map's elements as unordered sets, as described in Section 3.1. Therefore, in our implementation, we use our original zipper library, which has modified zipper structures for maps as well as several optimized functions for our purposes.

The list of BcMacro zipper functions is shown in **Table 2**.

Using zipper functions, for instance, we can translate the intermediate language instruction `(:var x)`, which binds `x` in the original pattern with the subtree rooted at the current location, to the following Clojure code:

```
(let [loc (->zip tree-root)
      ...
      x (node loc)
      ...]
  ...)
```

where Clojure's `let` expression is equivalent to other Lisp's `let*` expression.

As another example, the intermediate instruction `(:val v)`, which matches the current node with constant `v`, is translated to the following Clojure code:

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

where, if the matching fails, a dedicated exception is thrown. (For catching this exception, refer to `bcapply` in the next section.)

The other intermediate language instructions can be translated to Clojure code using zipper functions in the same straightforward way. Instructions `replace` and `replace-splicing` are used for replacing matching subtrees, `merge-rights` is used for processing a vector's `&` patterns, as described in the previous section, `lefts` and `rights` are used for realizing intermediate language instructions `no-left?` and `no-right?`, and `next` and `end?` are functions for traversing the entire tree and are used for auto macro expansions in the next section.

Finally, we explain the Clojure macro `bcmacro`, which generates a macro expansion function using the two-level translations of the last two sections. The definition of `bcmacro` is as follows.

```
(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#)))))
```

When the Clojure system compiles a source program that calls `bcmacro`, it generates a pair of the intermediate instructions `tgt` and `cxt` using the function `compile-pat`, which performs the translation of Section 5.2, and then generates Clojure code fragments using the function `emit-code`, which performs the translation of Section 5.3.

The Clojure macro `bp` is implemented in the almost same way as `bcmacro` except that it does not include the code for replacing the matching subtree and the whole code is enclosed with an exception handler so that, if matching fails, it returns `nil` instead.

5.4 Macro Expansion

The functions bcexpand-{1,all}, used directly by users for auto macro expansions, are implemented with the following lower-level functions bcfind and bcapply, which process a particular location of each zipper structure:

```
(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))))))
```

The function bcapply applies macro expansion functions in the sequence macros to location loc in order, and returns the result of the first-matching one. If nothing in macros matches, it returns nil. Function bcfind walks the entire tree in depth-first order, starting from location loc, while applying bcapply to each node in that order. Function bcfind returns the first non-nil result of bcapply (the location of the matching macro function's expansion result). If no macro expansion succeeds even after the traversal of the entire tree, it returns nil.

Using the above functions, bcexpand-{1,all} can be implemented as follows.

```
(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))))
```

6. Conclusion

In this paper, we propose BcMacro, a tool for writing the analysis and manipulation of JVM class files conveniently in Clojure. BcMacro is based on the data model that treats the entire class file as a syntax tree. Its manipulation unit is a macro definition that consists of a tree pattern specifying its rewrite target and a pure Clojure expression used for constructing a new subtree that is placed instead of the rewrite target.

The advantages of BcMacro are as follows:

- One tree pattern can extract multiple context data placed at distant locations.
- With the help of Clojure's macros, the syntax of BcMacro's tree patterns is intuitive and straightforward for Clojure programmers.
- Aside from auto macro expansions, we can trigger a macro expansion from Clojure code manually. We can also seamlessly incorporate the part of the code that is written declaratively with tree patterns with raw Clojure code.

For future work, we first want to identify manipulations that are impossible with the current BcMacro by investigating many real applications that manipulate Java bytecode.

Because BcMacro's macro definitions and tree patterns are independent of the Java bytecode format, they can be used for other binary data or general tree structures. Finding such applications and extending BcMacro's functionality according to their needs are another direction for our future work.

Furthermore, BcMacro's abstract syntax tree is constructed only from vectors and maps, which are pervasive data types supported by most modern programming languages. Therefore, we think that porting BcMacro to other languages or developing a variant of BcMacro in other languages will be relatively easy.

We also would like to extend BcMacro with features that automatically control the consistency of the Java bytecode, such as the preservation of mutual references in the constant pool and the auto-update of the offset operand of each jump instruction.

Acknowledgments This work was supported by JSPS Grant-in-Aid for Young Scientists (B) Grant Number 24700028. We are grateful to Dr. Keisuke Nakano from Kyoto University and reviewers of the paper, for their valuable comments on related work.

References

[1] Bodden, E. and Havelund, K.: Aspect-oriented Race Detection in Java, *IEEE Trans. Software Engineering*, Vol.36, No.4, pp.509–527 (2010).

[2] Brüggemann-Klein, A. and Wood, D.: Caterpillars: A context specification technique, *Markup Languages: Theory and Practice*, Vol.2, No.1, pp.81–106 (2000).

[3] Clark, J. and DeRose, S.: XML path language (XPath), available from ⟨http://www.w3.org/TR/xpath/⟩ (1999).

[4] Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S. and Tommasi, M.: Tree Automata Techniques and Applications, available from ⟨http://www.grappa.univ-lille3.fr/tata⟩ (2007).

[5] de Moor, O. and Sittampalam, G.: Higher-order matching for program transformation, *Theoretical Computer Science*, Vol.269, No.1–2, pp.135–162 (2001).

[6] Frisch, A., Castagna, G. and Benzaken, V.: Semantic Subtyping: Dealing Set-theoretically with Function, Union, Intersection, and Negation Types, *J. ACM*, Vol.55, No.4, pp.19:1–19:64 (2008).

[7] Hosoya, H. and Pierce, B.C.: XDuce: A Statically Typed XML Processing Language, *ACM Trans. Internet Technology* (*TOIT*), Vol.3, No.2, pp.117–148 (2003).

[8] Hu, Z., Yokoyama, T. and Takeichi, M.: Program Optimizations and Transformations in Calculation Form, *Generative and Transformational Techniques in Software Engineering*, Springer Berlin Heidelberg, pp.144–168 (2006).

[9] Huet, G.: Functional Pearl: The Zipper, *Journal of Functional Programming*, Vol.7, No.5, pp.549–554 (1997).

[10] Huet, G. and Lang, B.: Proving and Applying Program Transformations Expressed with Second-Order Patterns, *Acta Informatica*, Vol.11, No.1, pp.31–55 (1978).

[11] Kawanaka, S. and Hosoya, H.: biXid: A Bidirectional Transformation Language for XML, *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming* (*ICFP '06*), pp.201–214 (2006).

[12] Kuleshov, E.: Using the ASM framework to implement common Java bytecode transformation patterns, *Aspect-Oriented Software Development* (*AOSD '07*) (2007).

[13] Kutsia, T.: Context Sequence Matching for XML, *Electronic Notes in*

Theoretical Computer Science, Vol.157, No.2, pp.47–65 (2006).

[14] L. Marek, L., Villazón, A., Zheng, Y., Ansaloni, D., Binder, W. and Qi, Z.: DiSL: A Domain-specific Language for Bytecode Instrumentation, *Proc. 11th Annual International Conference on Aspect-oriented Software Development* (*AOSD 2012*), pp.239–250 (2012).

[15] Lindholm, T., Yellin, F., Bracha, G. and Buckley, A.: The Java Virtual Machine Specification, available from ⟨http://docs.oracle.com/javase/specs/jvms/se8/html/index.html⟩.

[16] Mohnen, M.: Context Patterns in Haskell, *Implementation of Functional Languages*, Springer Berlin Heidelberg, pp.41–57 (1997).

[17] Niehren, J., Planque, L., Talbot, J.-M. and Tison, S.: N-ary Queries by Tree Automata, *Proc. Int. Symp. on Database Programming Languages* (*DBPL*), pp.217–231 (2005).

[18] OW2 Consortium: ASM - Homepage, available from ⟨http://asm.ow2.org⟩.

[19] Pearce, D.J., Webster, M., Berry, R. and Kelly, P.H.J.: Profiling with AspectJ, *Software: Practice and Experience*, Vol.37, No.7, pp.747–777 (2007).

[20] Rothlisberger, D., Harry, M., Binder, W., Moret, P., Ansaloni, D., Villazon, A. and Nierstrasz, O.: Exploiting Dynamic Information in IDEs Improves Speed and Correctness of Software Maintenance Tasks, *IEEE Trans. Software Engineering*, Vol.38, No.3, pp.579–591 (2012).

[21] Yamamoto, A. and Yuasa, T.: JVM Extentions to Realize Tail Recursion Optimization and First class Continuations, *IPSJ Trans. Programming* (*PRO*), Vol.42, No.SIG11 (PRO12), pp.37–51 (2001).

[22] Yokoyama, T., Hu, Z. and Takeichi, M.: Deterministic Second-order Patterns, *Inf. Process. Lett.*, Vol.89, No.6, pp.309–314 (2004).

Seiji Umatani was born in 1974, and received his B.E. degree in information science, and M.E. and Ph.D. degrees in informatics from Kyoto University, Kyoto, Japan, in 1999, 2001, and 2004, respectively. In 2004–2005, he was a research staff member in the Graduate School of Informatics at Kyoto University, and he was appointed to an assistant professor in 2005. His current research interests include programming languages, compilers, and parallel/distributed systems. He is a member of ACM and the Japan Society for Software Science and Technology.

Tomoharu Ugawa received his B.Eng. degree in 2000, M.Inf. degree in 2002, and Dr.Inf. degree in 2005, all from Kyoto University. He worked for a research project on real-time Java at Kyoto University from 2005 to 2008. In 2008–2014, he was an assistant professor at the University of Electro-Communications. He is currently an associate professor at Kochi University of Technology. His work is in the area of implementation of programming languages with specific interest of memory management. He received IPSJ Yamashita SIG Research Award in 2012.

Masahiro Yasugi was born in 1967. He received his B.E. in electronic engineering, M.E. in electrical engineering, and Ph.D. in information science from the University of Tokyo in 1989, 1991, and 1994, respectively. In 1993–1995, he was a fellow of the JSPS (at the University of Tokyo and the University of Manchester). In 1995–1998, he was a research associate at Kobe University. In 1998–2012, he was an associate professor at Kyoto University. Since 2012, he is a professor at Kyushu Institute of Technology. In 1998–2001, he was a researcher at PRESTO, JST. His research interests include programming languages and parallel processing. He is a member of ACM and the Japan Society for Software Science and Technology. He was awarded the 2009 IPSJ Transactions on Programming Outstanding Paper Award.