Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE '03)
LNCS 2830, pp.364-376, Springer-Verlag, 2003.

An Easy-to-Use Toolkit for Efficient Java Bytecode Translators

Shigeru Chiba Muga Nishizawa

Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology
Email: {chiba,muga}@csg.is.titech.ac.jp

Abstract. This paper presents our toolkit for developing a Java-bytecode translator. Bytecode translation is getting important in various domains such as generative programming and aspect-oriented programming. To help the users easily develop a translator, the design of our toolkit is based on the reflective architecture. However, the previous implementations of this architecture involved serious runtime penalties. To address this problem, our toolkit uses a custom compiler so that the runtime penalties are minimized. Since the previous version of our toolkit named *Javassist* has been presented in another paper, this paper focuses on this new compiler support for performance improvement. This feature was not included in the previous version.

1 Introduction

Since program translators are key components of generative programming [5], a number of translator toolkits have been developed. For the Java language, some toolkits like EPP [9] and OpenJava [18] allow developers to manipulate a parse tree or an abstract syntax tree for source-level translation. Other toolkits, such as BCEL [6], JMangler [13], and DataScript [1], allow manipulating a class file, which is a compiled binary, for bytecode-level translation. The ease of use and the power of expressiveness are design goals of these toolkits. The latter goal means what kinds of translation are enabled. The former goal is often sacrificed for the latter one.

The bytecode-level translation has two advantages against the source-level translation. First, it can process an off-the-shelf program or library that is supplied without source code. Second, it can be performed on demand at load time, when the Java virtual machine (JVM) loads a class file. A disadvantage of the bytecode translation is, however, that the toolkits are difficult to use for developers who do not know detailed specifications of the Java bytecode.

To overcome this problem, a few researchers have proposed bytecode translator toolkits that provide higher-level abstraction than raw bytecode. For example, our toolkit named *Javassist* [4] provides source-level abstraction based

This work was supported in part by the CREST program of Japan Science and Technology Corp.

on the reflective architecture. The application programming interface (API) of Javassist is designed with only source-level vocabulary like *class* and *method* instead of bytecode-level one like *constant pool* and *invokevirtual*. Javassist interprets the program transformation written with the source-level vocabulary and executes an equivalent transformation at the bytecode level.

So far Javassist has allowed only limited modification of a method body for performance reasons. Other reflection-based translator toolkits enable inserting a hook at an interesting execution point such as method invocation in a method body so that the execution can be intercepted to change the computation at that point or to append extra computation there. The execution contexts at that point are converted into regular objects and passed to the intercepting code. If the intercepting code modifies these objects, then the modifications are reflected on the original execution contexts. The former conversion is called *reify* and the latter one is called *reflect*. Although this modification mechanism still restricts a range of possible program transformation, a number of research activities have revealed that it covers a wide range of application domains. Also, the idea of this mechanism has been also adopted by aspect oriented programming systems such as AspectJ [11, 12]. However, the reify and reflect operations are major sources of runtime overheads due to the reflective architecture. Existing reflective systems such as Kava [19] and Jinline [17] always perform these operations and thus imply not-negligible runtime overheads.

This paper presents our solution of this performance problem. We developed it for a new version of Javassist, which now allows the users to modify a method body as other reflection-based translator toolkits. To reduce runtime penalties due to the reify and reflect operations, Javassist suppresses unnecessary part of these operations as much as possible. A compiler specially developed for Javassist enables this suppression. It can compile source code written in Java with extensions for reification and reflection. Javassist uses this compiler for compiling intercepting code into efficient bytecode.

In the rest of this paper, we first show a performance problem of the previous implementations of the reflective architecture in Section 2. We next present our solution developed for Javassist in Section 3. We mention the results of our micro benchmark in Section 4. Related work is discussed in Section 5. We conclude this paper in Section 6.

2 Modifying a Method Body

Jinline [17] is a typical translator toolkit that is based on the reflective architecture and enables modifying a method body. From the implementation viewpoint, it is a Java class library developed on top of Javassist by a research group different from ours. With Jinline, the users can develop a bytecode translator that substitutes a hook for a specific expression in a method body. For example, it can substitute hooks for all the accesses to the fields in a Point class. The hooks can call the methods specified by the user so that the methods perform the altered behavior of the field-access expressions. In the case of Jinline, the call to

this method is inlined in the method body containing the replaced expression; this is why the toolkit is called Jinline.

The Jinline users can define a method that implements new behavior of the hooked expression and then they can specify that method to be called by the hook. However, the definition of that method must be subject to the protocol provided by Jinline. For a simple example, we below show a translator for enforcing the Singleton pattern [8]. Suppose that the translator guarantees that only a single instance of a class is created if the class implements an interface named Singleton.

First, the Jinline users prepare the following class for runtime support:

```
class Factory {
  static Hashtable singletons = new Hashtable();

  Object make(Object[] jinArgs) {
    String classname = (String)jinArgs[2];
    Class c = Class.forName(classname);
    Object obj = singletons.get(classname);
    if (obj == null) {
      Constructor cons = c.getDeclaredConstructor(...);
      obj = cons.newInstance((Object[])jinArgs[3]);
      singletons.put(classname, obj);
    }
    return obj;
  }
}
```

Then, they write a translator program using the Jinline toolkit. Since Jinline can notify the program whenever an interesting expression is found in the given method body, the program can have only to receive an object representing that expression and examine whether a hook must be substituted for that expression. To examine it, the program can use the lexical contexts supplied by Jinline. If a hook is substituted, the program must specify the method called by the hook. For the example above, we must specify the make method in Factory if the expression is the creation of an instance of a singleton class.

The parameter jinArgs to make is constructed by the hook at runtime. jinArgs[2] is the class name of the created object and jinArgs[3] is an array of Object representing the actual parameters to the constructor of the created object. The make method uses these runtime contexts to implement the Singleton pattern. For example, it uses jinArgs[3] for creating an object through the standard reflection API of Java [10]. The value returned by make is used as the result of the original expression. The type of the return value is converted by the hook into an appropriate type.

Constructing jinArgs and converting the type of the return value correspond to the *reify* and *reflect* operations of the reflective architecture. Exposing runtime contexts of the hooked expression through jinArgs is significant since they are needed to implement the altered behavior. It also makes the make method generic enough to deal with the instantiation of any class. Suppose that the hooked expression is "`new Point(3, 4)`". If the hook did not perform the reify

operation, it would directly pass the constructor parameters 3 and 4 to the make method. The make method would have to receive two int parameters and thus it could not be generic since it could not deal with instantiation with different types of parameters. A different make method would be necessary for every constructor with a different signature.

However, the reify and reflect operations are major sources of runtime penalties. If a parameter type is a primitive type, then these operations involves conversion between the primitive type such as int and the wrapper type such as java.lang.Integer. Since these operations are performed whenever the hooked expression is executed, this overhead is not negligible.

3 Javassist

Javassist [4] is a reflection-based toolkit for developing Java-bytecode translators. It is a class library in Java for transforming Java class files (bytecode) at compile time or load time. Unlike other libraries that are not based on reflection, it allows the users to describe transformation with source-level vocabulary; the users do not have to have detailed knowledge of bytecode or the internal structure of Java class file.

The Javassist users can first translate a Java class file into several objects representing a class, field, or method. The users' programs can access these "meta" objects for transformation. Introducing a super interface, a new field, and so on, to the class is performed through modifying these objects. The modifications applied to these metaobjects are finally translated back into the modifications of the class file so that the transformation is reflected on the class definition. Since Javassist does not expose internal data structures contained in a class file, such as a constant pool item and a method_info structure, the developers can use Javassist without knowledge of Java class files or bytecode.* On the other hand, other libraries such as BCEL [6] provide objects that directly represent a constant pool item and a method_info structure.

Javassist allows the users to modify a method body as Jinline does. To avoid the performance problem, Javassist lets the users explicitly specify when the reify and reflect operations should be performed. Several meta variables and types are available for specifying this in the code executed by a hook. Javassist analyzes the occurrences of these meta variables and types and it thereby eliminates unnecessary reify and reflect operations.

3.1 Structural Reflection

The CtClass object is an object provided by Javassist for representing a class obtained from the given class file. It provides the almost same functionality of

* For practical reasons, Javassist also provides another programming interface to directly access the internal data structures in a class file. However, normal users do not have to use that interface.

Table 1. Part of methods for modifying a class

Methods in CtClass	Description
`void setName(String name)`	change the class name
`void setModifiers(int m)`	change the class modifiers such as `public`
`void setSuperclass(CtClass c)`	change the super class
`void setInterfaces(CtClass[] i)`	change the interfaces
`void addField(CtField f, String i)`	add a new field
`void addMethod(CtMethod m)`	add a new method
`void addConstructor(CtConstrutor c)`	add a new constructor

Table 2. Part of methods for modifying a member

Methods in CtField	Description
`void setName(String n)`	changes the field name
`void setModifiers(int m)`	changes the field modifiers
`void setType(CtClass c)`	changes the field type
Methods in CtMethod, CtConstructor	**Description**
`void setName(String n)`	changes the method name
`void setModifiers(int m)`	changes the method modifiers
`void setExceptionTypes(CtClass[] t)`	sets the types of the exceptions that the method may throw
`void setBody(String b)`	changes the method body

introspection as the java.lang.Class class of the standard reflection API. Introspection means to inspect data structures, such as a class, used in a program. For example, the getName method declared in CtClass returns the name of the class, the getSuperclass method returns the CtClass object representing the super class. getFields, getMethods, and getConstructors return CtField, CtMethod, and CtConstructor objects representing fields, methods, and constructors, respectively. These objects parallel java.lang.reflect.Field, Method, and Constructor. They provide various methods, such as getName and getType, for inspecting the definition of the member. Since a CtClass object does not exist at run time, the newInstance method is not available in CtClass unlike in java.lang.Class. For the same reason, the invoke method is not available in CtMethod and so forth.

Unlike the standard reflection API, Javassist allows developers to alter the definition of classes through CtClass objects and the associated objects (Table 1 and 2). For example, the setSuperclass method in CtClass changes the super class of the class. The addMethod method adds a new method to the class. The definition of the new method is given in the form of String object representing the source text. Javassist compiles the source text into bytecode on the fly and adds it into the class file. The addField method adds a new field. It can take source text

representing the initial value of the field. Javassist compiles the source text and inserts it in the constructor body so that the field is appropriately initialized.

The setName method in CtClass changes the name of the class. To keep consistency, several methods like setName perform more than changing one attribute field in a class file. For example, setName also substitutes the new class name for all occurrences of the old class name in the class definition. The occurrences of the old class name in method signatures are also changed.

3.2 Behavioral Reflection

The new version of Javassist allows the users to modify a method body as other reflection-based toolkits. The users can develop a bytecode translator that inserts a hook at the beginning or end of a method body. The bytecode translator can also substitute a hook for a specific expression in a method body. The hook executes intercepting code specified by the users in the form of source text. The intercepting code can be a single Java statement or several statements surrounded by {} and it can directly execute the altered behavior of the hooked expression or call another method for indirectly executing the altered behavior. The code is inlined in the hook and thus executed in the same scope as the original expression. It can access private fields of the object although it cannot access local variables.

The following example substitutes the hooks for the caller-side expressions that invoke the move method in the Point class if the expressions belong to the Graph class:

```
CtClass cc = ClassPool.getDefault().get("Graph");
cc.insrtument(new ExprEditor() {
  public void edit(MethodCall m) {
    if (m.getClassName().equals("Point")
        && m.getMethodName().equals("move"))
      m.replace("{ System.out.println(\"calling move()\");"
              + "  $_ = $proceed($$); }");
  }
});
```

The hook executes the code printing a log message. The variable cc is the CtClass object representing the Graph class.

The instrument method in CtClass receives an ExprEditor object and scans the bodies of all the methods declared in the class, in the case above, the Graph class. If an interesting expression is found, the edit method is invoked on the given ExprEditor object with a parameter representing the expression. The edit method can be invoked if a method call, a field access, object creation by the new operator, an instanceof expression, a cast expression, or a catch clause is found. The parameter is a MethodCall, FieldAccess, NewExpr, Instanceof, Cast, or Handler object, respectively. These objects provide various methods for inspecting the lexical contexts of the expression.

The edit method can inspect the given parameter to determine whether a hook must be substituted for the expression. If the hook must be substituted,

the replace method is called on the given parameter. It replaces the expression with the hook that executes the given Java statement or block. For the example above, the hook executes the following block:

```
{ System.out.println("calling move()");
  $_ = $proceed($$); }
```

The second statement in the block is written with special variables starting with $, which are extensions to Java by Javassist. It executes the original method-call expression.

Besides the instrument method, insertBefore, insertAfter, and addCatch methods are available in the CtMethod and CtConstructor classes. They receive source text as a parameter, compile it, and insert the hook executing the compiled code at the beginning or end of the method. The addCatch method inserts the hook so that the hook will be executed when an exception of the specified type is thrown in the method body. The hook executes the code given to addCatch as a parameter.

3.3 Meta Variables and Types

The reify and reflect operations have been major sources of runtime overheads in the reflective architecture. They are operations for converting runtime contexts to/from regular Java objects so that the program can access and modify them. Since other toolkits like Jinline use a regular Java compiler for compiling the code executed by a hook, the code must be written as a regular Java method, which must take the reified object as a parameter and return an object to be reflected on the runtime contexts. Thus, the hook must always perform the reify and reflect operations before/after invoking the method even though they might be often unnecessary.

To avoid this problem, Javassist uses a Java compiler specially developed for compiling the intercepting code. The compiler interprets several symbols in the source text of that code as *meta* variables or types (Table 3). These symbols are used to access the runtime or lexical contexts of the expression replaced with the hook. If necessary, this access involves the reify and reflect operations. From the implementation viewpoint, these symbols are macro variables expanded to context-dependent text at compile time.

The meta variables enable Javassist to perform the reify and reflect operations on demand. Javassist does not perform these operations if the code executed by the hook does not need them. If only part of the runtime contexts must be reified or reflected, the compiler produces optimized bytecode to minimize runtime penalties due to the reify and reflect operations.

We below show details of some significant meta variables and types.

- $0, $1, $2, ...

They represent method parameters if the hooked expression is method call. $0 represents the target object. The types of $0, $1, ... are identical to the types

Table 3. Meta variables and types

`$0, $1, $2, ...`	parameter values
`$_`	result value
`$$`	a comma-separated sequence of the parameters
`$args`	an array of the parameter values
`$r`	formal type of the result value
`$w`	the wrapper type
`$proceed(..)`	execute the original computation
`$class`	a `java.lang.Class` object representing the target class
`$sig`	an array of `java.lang.Class` representing the formal parameter types
`$type`	a `java.lang.Class` object representing the formal result type
`$cflow(..)`	a mechanism similar to `cflow` of AspectJ

of the corresponding parameters. If the value of $1, $2, ... is updated, the value of the corresponding parameter is also updated.

If the hooked expression is object creation, then $1, $2, ... represent the actual parameters to the constructor. If it is field assignment, then $1 represents the assigned value. The other variables like $2 are not available.

- `$_`

The meta variable `$_` represents the result value of the hooked execution. If a new value is assigned to this meta variable, then the assigned value becomes the result of the method call, field read, object creation, and so on. The type of `$_` is identical to the type of the result value of the original expression. If the result type is void, then the type of `$_` is Object.

- $$

The meta variable $$ is interpreted as a comma-separated sequence of all the actual parameters. For example, if the hooked expression is a call to a method move(int, int, int), then $$ is syntactically equivalent to $1, $2, $3. move($$) is equivalent to move($1,$2,$3). This meta variable abstracts the number of parameters from the source text so that the source text can be generic.

- $proceed

This is a meta method. If it is invoked, then the original computation of the hooked expression is executed. For example, if the expression is a method call, then the originally called method is invoked. If the expression is field read, then $proceed() returns the value of the field. Typical usage of this meta method is as following:

```
$_ = $proceed($$);
```

This executes the original computation with the current runtime contexts, which may be updated through the meta variables.

Note that the types of the parameters of $proceed is the same as those of the original ones. The result type is also the same. If the expression is field read, then $proceed does not take a parameter. If the expression is field assignment, then $proceed takes a new value of the field as a parameter. If the expression is object creation, then $proceed takes the same parameters as the constructor.

• $args

The meta variable $args represents an array of all the parameters. The type of this meta variable is an array of Object. Whenever this meta variable is read, a new copy of the array is created and the parameters are stored in the array. Note that $args is different from $$; $args can be used as a regular Java variable whereas $$ is syntax sugar used only with a method call.

If the type of a parameter is a primitive type such as int, then the parameter value is converted into a wrapper object of that primitive value. For example, an int value is converted into a java.lang.Integer object to be stored in $args.

If an array of Object is assigned to $args, then each element of that array is assigned to each actual parameter. If a parameter type is a primitive type, the type of the corresponding array element must be a wrapper type such as java.lang.Integer. The value of the element is converted from the wrapper type to the primitive type before it is assigned to the parameter.

• $r and $w

These are meta types available only in a cast expression. $r represents the result type of the hooked expression. If the expression is a method call, then $r represents the return type of the method call.

If the result type is a primitive type, then ($r) converts the value from the wrapper type to the primitive type. For example, if the result type is int, then

```
Object res = new Integer(3);
$_ = ($r)res;
```

converts res into the value 3 and assigns it to the meta variable $_ of type int.

If the result type is void, then the type cast operator with ($r) is ignored. If the type cast operator with $r is used in the return statement, then that statement is regarded as the return statement without any return value. For example, if res is a local variable and the result type is void, then

```
return ($r)res;
```

is regarded as:

```
return;
```

This specification is useful for the generic description of the intercepting code.

($w) converts the value from a primitive type to the wrapper type. For example, in this program:

```
Integer i = ($w)5;
```

the cast expression converts the int value 5 to the java.lang.Integer object. If ($w) is applied to a value of Object type, then ($w) is ignored.

3.4 Compilation

The hook produced by Javassist does not prepare an object corresponding to jinArgs of Jinline. Rather, the prologue of the hook stores the runtime contexts in local variables without any data conversion. If the hooked expression is a method call, all the parameters pushed on the operand stack are popped and stored in the variables. The epilogue of the hook reads the value of $_, which is implemented as a local variable, and pushes it on the operand stack as the resulting value of the hooked expression. The intercepting code executed by the hook is inlined between the prologue and the epilogue.

The local variables containing the runtime contexts are accessed through the meta variables. If $args is read, the runtime contexts are obtained from the local variables and the parameters are converted into an array of Object. This conversion is the reify operation. Javassist does not perform the reify operation until it is explicitly required by the meta variables such as $args.

Since the programmers can use the meta variables for explicitly specifying when and what runtime contexts must be reified or reflected, Javassist can minimize runtime penalties due to the reify and reflect operations. If these operations are not required, Javassist never performs it. For example, if the source text of the intercepting code is only `"$_ = $proceed($$);"`, then no reify or reflect operations are performed. $proceed is compiled into the bytecode sequence for executing the original computation. For example, if the hooked expression is a method call, the parameters are loaded from the local variables onto the operand stack and the method originally called is invoked by the invokevirtual instruction. The resulting value on the operand stack is stored into $_. The overheads are only extra costs of load and store instructions.

3.5 Example

In Section 2, we showed an example of the use of Jinline. For this example, we substituted a hook for an expression for creating an object so that the Singleton pattern would be enforced.

This example can be implemented with Javassist as well. We must specify that the hook will execute the following intercepting code:

```
Object obj = Factory.singletons.get($class);
if (obj != null)
  $_ = ($r)obj;
else {
  $_ = $proceed($$);
  Factory.singletons.put($class, ($w)$_);
}
```

This intercepting code is generic; it covers all the singleton classes.

Factory.singletons is a hashtable containing the singleton objects that have been created. The intercepting code first searches this hashtable and, if it finds an object of the target class, then it returns that object. Otherwise, the code creates the object by $proceed and includes it in the hashtable. Note that the meta types $r and $w are used in cast expressions since the type of $_ (the resulting value) is the class type of the created object.

4 Experiment

To measure the runtime overhead of the hook substituted by Javassist, we executed micro benchmark tests. We used Javassist to replace a method-call expression with a hook that executes the following intercepting code:

```
{ $_ = $proceed($$); }
```

This only executes the original computation, which is a method invocation with the original parameters. We measured the elapsed time of executing this hooked expression. The body of the called method was empty. The measured time represents the overhead of Javassist in the best case where no reification or reflection is required.

For comparison, we also measured the elapsed time with the hook that invokes the originally called method through the standard reflection API of Java. The given code is:

```
{ $_ = ($r)method.invoke($0, $args); }
```

Here, method is a static field containing a java.lang.reflect.Method object. It represents the originally called method. Note that $args is used for obtaining the parameter list and ($r) is for converting the type of the result value. They are the reify and reflect operations. The measured time represents the minimum overhead of the previous implementation technique of the reflective architecture, which always performs the reify and reflect operations. Furthermore, we measured the time with the behavioral reflection system included in the Javassist toolkit. The system enables typical runtime reflection as Kava [19] does. In this measurement, a metaobject trapped the method call and only executed the body of the originally called method. The measured time includes the cost for handling the metaobject as well as the reify and reflect operations.

Table 4 lists the results of our measurement using Sun JDK 1.4.0_01 for Solaris 8. The listed numbers are the average of four million iterations after one million iterations. We measured for several combinations of the return type and the parameter types of the null method called by the hook. Reflection API means the standard reflection API and Behavioral means the behavioral reflection system. If Javassist was used, the elapsed time of all the combinations except one was less than 10 nanoseconds. Since the pair of call and retl machine instructions takes about 10 nanoseconds, these results mean that the overhead was negligible and thus the method invocation was inlined by the JVM. On the other hand, the other experiments showed the overheads were about 500 to 1,300 nanoseconds.

Table 4. The elapsed time of a null method call (nsec.)

Return type	void	void	String	String	void	int	int	double
Parameter types	no	String	String	String×2	int	int	int×2	double×2
Javassist	∗	∗	∗	∗	∗	∗	∗	20
Reflection API	500	550	560	630	760	880	1,110	1,290
Behavioral	560	620	620	700	820	930	1,200	1,370

Sun Blade 1000 (Dual UltraSPARC III 750MHz, 1GB memory), Solaris 8, Sun JDK 1.4.0_01
∗ indicates the time was less than 10 nsec.

5 Related Work

This work is not the first work on improving the runtime performance of reflective systems. The technique of partial evaluation [7] has been actively studied in this research field [14, 2]. However, since partial evaluation involves relatively long compilation time, it is not appropriate for bytecode translators, which may be used at load time. Compile-time reflection [3] can improve the runtime performance but it makes runtime contexts difficult to access from the program.

The pointcut-advice framework of the AspectJ language [12] is similar to the programming framework of Javassist. It allows programmers to insert a hook in a method body for executing the code given as *advice*. Like Javassist, AspectJ does not perform the reify operation unless they are explicitly requested through the special variables such as thisJoinPoint. However, AspectJ is not a bytecode translator toolkit but an aspect-oriented programming language and it does not well support the reflect operation. On the other hand, Javassist supports both of the reify and reflect operations. For example, assignment to $args updates the runtime contexts.

The problem of Jinline mentioned in Section 2 is included in other aspect-oriented systems that provide the pointcut-advice framework but are implemented as a class library. For example, JAC [15] always reifies the runtime contexts before passing them to the wrappers. PROSE [16] allows choosing whether the runtime contexts are reified or not but, if they are not reified, the description of advice cannot be generic. Our compiler-based solution will be applicable to those systems.

6 Conclusion

This paper presented a new version of our Java-bytecode translator toolkit named *Javassist*. It allows the programmers to modify a method body according to the reflection-based framework. A unique feature against other reflection-based toolkits like Jinline is that Javassist uses a customized compiler for reducing runtime penalties due to the reify and reflect operations, which are fundamentals of the reflective architecture. These runtime penalties have been disadvantages of reflection-based toolkits whlie ease of use by the high-level abstraction

has been an advantage. Javassist reduces the runtime penalties while keeping the ease of use. The version of Javassist presented in this paper has been already released to the public and getting widely used. Applications of Javassist include product-quality software like the JBoss EJB server.

References

1. Back, G., "DataScript — A Specification and Scripting Languages for Binary Data," in *Generative Programming and Component Engineering (GPCE 2002)* (D. Batory, C. Consel, and W. Taha, eds.), LNCS 2487, pp. 66–77, Springer, 2002.
2. Braux, M. and J. Noyé, "Towards Partially Evaluating Reflection in Java," in *Proc. of Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)*, SIGPLAN Notices vol. 34, no. 11, pp. 2–11, ACM, 1999.
3. Chiba, S., "A Metaobject Protocol for C++," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, SIGPLAN Notices vol. 30, no. 10, pp. 285–299, ACM, 1995.
4. Chiba, S., "Load-time structural reflection in Java," in *ECOOP 2000*, LNCS 1850, pp. 313–336, Springer-Verlag, 2000.
5. Czarnecki, K. and U. W. Eisenecker, *Generative Programming.* Addison Wesley, 2000.
6. Dahm, M., "Byte Code Engineering with the JavaClass API," Techincal Report B-17-98, Institut für Informatik, Freie Universität Berlin, January 1999.
7. Futamura, Y., "Partial Computation of Programs," in *Proc. of RIMS Symposia on Software Science and Engineering*, LNCS, no. 147, pp. 1–35, 1982.
8. Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns.* Addison-Wesley, 1994.
9. Ichisugi, Y. and Y. Roudier, "Extensible Java Preprocessor Kit and Tiny Data-Parallel Java," in *Proc. of ISCOPE '97*, LNCS, no. 1343, 1997.
10. Java Soft, "Java^{TM} Core Reflection API and Specification." Sun Microsystems, Inc., 1997.
11. Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *ECOOP'97 – Object-Oriented Programming*, LNCS 1241, pp. 220–242, Springer, 1997.
12. Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," in *ECOOP 2001 – Object-Oriented Programming*, LNCS 2072, pp. 327–353, Springer, 2001.
13. Kniesel, G., P. Costanza, and M. Austermann, "JMangler — A Framework for Load-Time Transformation of Java Class Files," in *Proc. of IEEE Workshop on Source Code Analysis and Manipulation*, 2001.
14. Masuhara, H. and A. Yonezawa, "Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Languages," in *ECOOP'98 - Object Oriented Programming*, LNCS 1445, pp. 418–439, Springer, 1998.
15. Pawlak, R., L. Seinturier, L. Duchien, and G. Florin, "JAC: A Flexible Solution for Aspect-Oriented Programming in Java," in *Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, LNCS 2192, pp. 1–24, Springer, 2001.
16. Popovici, A., T. Gross, and G. Alonso, "Dynamic Weaving for Aspect-Oriented Programming," in *Proc. of Int'l Conf. on Aspect-Oriented Software Development (AOSD'02)*, pp. 141–147, ACM Press, 2002.

17. Tanter, E., M. Ségura-Devillechaise, J. Noyé, and J. Piquer, "Altering Java Semantics via Bytecode Manipulation," in *Generative Programming and Component Engineering (GPCE 2002)* (D. Batory, C. Consel, and W. Taha, eds.), LNCS 2487, pp. 283–298, Springer, 2002.
18. Tatsubori, M., S. Chiba, M.-O. Killijian, and K. Itano, "OpenJava: A Class-based Macro System for Java," in *Reflection and Software Engineering* (W. Cazzola, R. J. Stroud, and F. Tisato, eds.), LNCS 1826, pp. 119–135, Springer Verlag, 2000.
19. Welch, I. and R. Stroud, "From Dalang to Kava — The Evolution of a Reflective Java Extension," in *Proc. of Reflection '99*, LNCS 1616, pp. 2–21, Springer, 1999.