A Yet Another `java.lang.Class`

Shigeru Chiba Michiaki Tatsubori
Institute of Information Science and Electronics
University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki 305-8573, Japan.
Phone: +81-298-53-5349 Fax: +81-298-53-5206
E-mail: `chiba@is.tsukuba.ac.jp`

1 Introduction

The Java language already has the ability for reflection [2, 4]. `java.lang.Class` is a class for class metaobjects and provides the ability for introspection at runtime. At runtime, the Java programmers can inspect various information, for example, the class of a given object, which methods that class has, and so forth. They can also get the value of a field and invoke a method specified by a character string. For example, they may write the following code:

```
Person p = new Person("Joe");
Class c = p.getClass();
Field f = c.getField("name");    // the field name is a variable
String name = (String)f.get(p); // same as String name = p.name;
```

However, the Java language has not provided the ability for language customization, or *intercession*, which is another kind of reflection. The Java programmers cannot change the super class of a class, alter the behavior of method invocation, or extend the syntax. A few extensions to Java, such as MetaJava [3], have been proposed to support the ability for language customization, but their abilities are limited to customizations that do not imply severe performance impacts.

This paper proposes an extended version of `java.lang.Class`, which enables more comprehensive language customization than other similar systems. To avoid performance degradation, we employ a technique called compile-time reflection. Most of customizations are statically applied to a Java program at compile time and the compiled code is executed by the regular Java virtual machine (JVM) with almost no performance overheads.

In the rest of this paper, we first mention requirements for the ability for language customization. Then we overview the proposed extension to Java's reflection mechanism.

2 Requirements for customizability

First of all, we briefly summarize the kinds of language customization that we should be able to perform in Java by the extended reflection mechanism. These customizations are considerably useful in practice according to our experience with OpenC++ [1].

To customize the structure of a class

The reflection mechanism should allow the meta-level programmers to manipulate the structure of a class. The programmers should be able to add a new method and field to a class and remove them from a class. Also, various properties of a class, such as the super class, the name of a method, the type of a filed, and the access rights to members, should be modifiable. For example, the following code should be accepted:

```
Class type = Class.forName("double");
Field f = new Field(type, "z");
Class c = Class.forName("Point");
c.addField(f);     // The double field z is added to Point
```

Moreover, the programmers should be able to create a new class, which is derived from other classes. This ability is useful to implement a distributed Java object. It makes it unnecessary for the programmer to explicitly define a class for the proxy object, which is automatically derived from the class for that distributed object if that ability is provided.

To alter the behavior of operations on objects

The significance of the ability to reimplement the behavior of the operations on particular objects is well known to people who are studying the use of reflection in practice. Those operations include method calls, field accesses, operator (for example, `+` and `-`) applications, type coercion (i.e. type casting). A well-known example of the use of this ability is to reimplement method calls on a proxy object so that the method calls are dispatched to the remote object represented by that proxy object. Another, but less practical, example is to reimplement field accesses so that a debug message is printed whenever the fields are read on particular objects. Here is a pseudo code for implementing that behavior:

```
class VerboseClass extends Class {
  Object fieldRead(Object o, Field f){
      System.out.println(f.getName() + " is read.");
      return super.fieldRead(o, f);
  }
}
```

If a field is read on the objects the metaclass of that is `VerboseClass`, then the method `fieldRead()` is called so that a debug message is printed.

To alter variable types

Another kind of language customization is to change the interpretation of variable types. For example, the meta-level programmers may want to store a proxy object in a variable the type of that is a distributed object. Or, they may want to do so if the variable is declared with a specific modifier such as `remote`. The meta-level program to implement that customization would be something like this:

```
class DistributedClass extends Class {
  Class realVariableType(String modifier){
      if (modifier.equals("remote"))
          return Class.forName("Proxy" + this.getName());
      else
          return this;
  }
}
```

If this metaclass `DistributedClass` is associated with a class `Person`, then a variable of the `Person` class:

```
    remote Person    joe;
```

is interpreted as a variable of the `ProxyPerson` class.

This kind of language customization is also useful to implement a mechanism similar to C++'s template. It would be convenient if the base-level programmers can write:

```
    Vector of String    v;
```

and then the variable `v` is interpreted as the variable of the class `VectorOfString`, which is automatically derived by the metaclass from the class `Vector`.

To define a new kind of syntax

The last is the ability for syntax extensions. For example, to utilize parallel computers, the programmers may want to define a new control structure `forall`, which is used as follows:

```
Matrix m1, m2;
    :
m1.forall(int i, int j, double e){
    e = m2.elementAt(i, j) * 2.0;
}
```

The block {} is executed on each element of the matrix `m1`. In the block, `e` is bound to that element, and `i` and `j` are bound to the indexes. This control structure should be compiled into byte code using threads to execute the block in parallel.

A much simpler example is an `unless` statement. It is the opposite of an `if` statement:

```
unless (b > 0) {
    b = -b;
}
```

If the expression `b > 0` is false, then the block { `b = -b;` } is executed.

At the meta level, the `unless` statement should be interpreted by a method like the following:

```
void unlessStatement(Environment env,
                     Expression exp, Block blk){
    if (!exp.evaluate(env))
        blk.evaluate(env);
}
```

This method is invoked if the program execution reaches an `unless` statement. It interprets the program according to the semantics of the `unless` statement.

3 Use of compile-time reflection

If we modify the regular Java virtual machine (JVM), the ability for the language customizations shown in the previous section could be easily implemented. However, this implementation may make it difficult to apply optimization techniques on method dispatch and field access to the JVM since methods and fields may be dynamically altered, removed, and added. In general, those optimization techniques improve execution performance assuming that all the methods and fields are statically given. Modifying the JVM for implementing the ability for language customizations is naive but may make the JVM inefficient especially if it includes a just-in-time (JIT) compiler or other complex optimizers.

To give the ability for language customization to Java, we do not modify the JVM but modify the Java compiler with a technique called *compile-time reflection*. With this technique, most of the customizations are applied at compile time and the compiled Java program is executed by the regular JVM. This means that runtime overheads due to the language customization are significantly limited. Although compile-time reflection has been already employed by other systems such as OpenC++ [1], this paper discusses the application of compile-time reflection to the metaobjects providing the ability for introspection at runtime. The metaobjects of the previous systems work only at compile time but they do not provide any ability for reflection at runtime.

3.1 `openjava.mop.OJClass`

We are currently implementing a class `openjava.mop.OJClass` for class metaobjects and the associate classes such as `OJMethod`. These classes are reimplementations of `java.lang.Class` and `Method` although their names begin with `OJ` for easy distinction, and they provide the ability for not only introspection but also language customization. The ability for language customization is implemented with compile-time reflection in cooperation with our Java compiler, called *the OpenJava compiler*.

Our `OJClass` supplies all the methods that Java's `Class` does. In addition, `OJClass` supplies a method `transformClass()`, which is an empty method but overridable by a subclass. The meta-level programmers who want to customize the structure of a class can define a subclass of `OJClass` and override this method. Since `transformClass()` is called on every base-level class associated with that subclass, the programmers can customize the structure of that base-level class in that method, for example, they can add a new field:

```
void transformClass(Environment e){
    Class type = Class.forName("double");
    OJField f = new OJField(type, "z");
    addField(f);    // add this field z to the base-level class
}
```

The method `addField()` is supplied by `OJClass`. It adds a given field represented by a `OJField` object to the class.

A unique feature is that our Java compiler internally creates a class metaobject for every base-level class in the processed source program. If the programmer associates a subclass of `OJClass` with a base-level class, then the compiler instantiates that subclass for that base-level class at *compile time*. Then the compiler calls `transformClass()` on all the class metaobjects and translates the source program according to the results of that method. In the case of the example above, the result of `addField()` is reflected on the source program so that the definition of that base-level class is modified to include the added field `z`. After the source-level translation, the program is compiled by the regular Java compiler. Since the resulting byte code includes a class that the customization of the class structure has been already applied to, it does not imply any runtime overheads due to the reflection.

3.2 The behavior of operations on objects

Reflecting changes of the structure of a class on the source program at compile time is easy and straightforward but how should we deal with changes of the behavior of operations on objects, such as a method call? In the previous section, we showed an example of meta-level methods representing new behavior of the operations. The method `fieldRead()` in that example is apparently meaningless if it is execute at compile time; it must be executed at runtime.

To solve this problem, our OpenJava compiler translates an expression executing an operation on an object so that the class metaobject intercepts that operation. Suppose that you want to alter the behavior of field accesses on `Person` objects. To do this, you define a subclass of `OJClass` to override `fieldRead()`, and associate that subclass to the class `Person`. If a variable `p` is a `Person` object, then this expression:

```
    p.name
```

is translated by the OpenJava compiler into this:

```
    (String)(OJClass.getClass(p).notifyRead(p, "name"))
```

At runtime, this expression calls `notifyRead()` on the class metaobject for `p`. Note that the class metaobject exists at both compile-time and runtime. The implementation of `notifyRead()` is supplied by `OJClass`:

```
Object notifyRead(Object o, String field_name){
    OJField f = getField(field_name);
    return readField(o, f);
}
```

This method looks up the field metaobject and calls `readField()` with it, which implements the new behavior of the base-level method call.

The other operations such as method calls are also processed in the same technique. All the expressions including operations on objects are translated at compile time so that the class metaobjects can intercept the operations at runtime. This technique enables changes of the behavior of those operations without modifying the JVM.

3.3 Variable types

The OpenJava compiler also applies changes of the interpretation of variable types to the source program at compile time. The implementation is simple — the compiler replaces the type name of the declaration of the variable with the type returned by the class metaobject. Recall the example in the previous section. The method `realTypeVariable()` returned the class metaobject for the type of a given variable according to the modifier.

3.4 Syntax extensions

Finally, we mention how the OpenJava compiler deals with syntax extensions. The technique used for changing the behavior of operations on objects cannot be effective for syntax extensions because it requires closures. In the example of an `unless` statement in the previous section, the method `unlessStatement()` took two arguments, `exp` and `blk`, and interpreted the statement with them. These arguments are closures and calling `evaluate()` on these closures must return the resulting value of executing the code. The closures cannot be implemented with a simple program translation in Java.

We enable syntax extensions in Java with a different technique. We use macro expansion like Lisp's to implement syntax extensions. For example, the `unless` statement:

```
unless (b > 0) {
    b = -b;
}
```

is expanded by a class metaobject into a regular Java statement:

```
if (!(b > 0)) {
    b = -b;
}
```

To specify the macro expansion, our `openjava.mop.OJClass` supplies an overridable method `expandNewStatement()`. This is called by the OpenJava compiler at compile time to perform the macro expansion. To expand an `unless` statement, the meta-level programmers can override `expandNewStatement()` as follows:

```
ParseTree expandNewStatement(Environment e, ParseTree statement)
    throws SyntaxError
{
    if (statement.first().equals("unless")) {
        ParseTree expr = statement.second();
        ParseTree block = statement.third();
        return new ParseTree("if(!" + expr.toString() + ")"
                             + block.toString());
    }
    else
        throw new SyntaxError();
}
```

This method takes a compile-time environment and a parse tree representing the `unless` statement. The parse tree is a linked list similar to Lisp's list data structure. The method `expandNewStatement()` transforms the given parse tree and returns a different parse tree representing regular Java code. The OpenJava compiler substitutes the resulting parse tree for the original `unless` statement in the source program.

Unfortunately, this implementation requires changing the protocol of the class metaobjects. The meta-level programmers cannot define the semantics of the `unless` statement in the interpretation style shown in the previous section. Instead, they have to define it in the macro-expansion style. The drawbacks of this change is controversial; some say they are serious but we do not think so. According to our experiences with OpenC++, defining a new syntax in the macro-expansion style is not difficult but rather it is often intuitive.

4 Concluding remarks

We are implementing a reflection mechanism for Java, which replaces Java's original `java.lang.Class`. Our reflection mechanism provides the ability for language customization that is more comprehensive than Java's original one and similar extensions [3].

To avoid performance drawbacks, we employ a technique called compile-time reflection. Our reflection mechanism is implemented by a pair of our OpenJava compiler and a class library substituting `java.lang.Class` and `java.lang.reflect.*`. The Java virtual machine (JVM) is not modified at all. The OpenJava compiler applies language customizations written in meta-level programs to the base-level programs at compile time so that extra costs due to reflection are not necessary at runtime. On the other hand, our approach implies limitations on the ability for customization. For example, the meta-level programmers cannot change the structure of a class dynamically at runtime. We think that addressing this problem needs to modify the JVM and involves too much overheads to justify its benefits for the time being. This issue is our future work.

References

[1] Chiba, S., "A Metaobject Protocol for C++," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, no. 10 in SIGPLAN Notices vol. 30, pp. 285–299, ACM, 1995.

[2] Java Soft, *Java^{TM} Core Reflection API and Specification.* Sun Microsystems, Inc., 1997.

[3] Kleinöder, J. and M. Golm, "MetaJava: An Efficient Run-Time Meta Architecture for Java," in *Proc. of the International Workshop on Object Orientation in Operating Systems (IWOOS'96)*, IEEE, 1996.

[4] Kramer, D., *JDK 1.1.6 Documentation.* Sun Microsystems, Inc., 1998.