Soot's support for JSR292 (invokedynamic)

Eric Bodden (eric.bodden@ec-spride.de)

January 13th, 2012

Abstract

This document describes Soot's support for JSR292[1], also known as the invokedynamic bytecode. Since version 2.5.0, Soot supports the invokedynamic bytecode in all its intermediate representations. Users can use Soot to easily parse Java 7+ bytecode into these different representations, or can conversely use the representations to generate Java 7+ bytecode containing invokedynamic instructions.

1 Introduction: invokedynamic bytecode

JSR292 describes a new bytecode, invokedynamic, and a series of new constant-pool entries that, together, allow the execution of invokedynamic instructions. Unlike all other Java bytecode invoke instructions, the target method of an invokedynamic instruction is not defined by the language semantics, but rather by a user-defined method, called the bootstrap method.

The purpose of a bootstrap method is to associate an invokedynamic statement with a CallSite object. A call to this object's `getTarget()` method will then return a method handle to the invokedynamic statement's target method. Call sites may be constant or mutable, i.e., may or may not return the same method handle object on each call. Each invokedynamic instruction can be associated with its own bootstrap method, or multiple instructions can share the same method.

In addition to its bootstrap method, an invokedynamic instruction can take a number of static arguments. The arguments are called static because the bootstrap method is executed at load time. For the same reason, static arguments are restricted to the argument types that can be encoded in the constant pool.

Finally, an invokedynamic instruction states a method name and a number of regular invocation arguments. Note that, unlike for normal methods, this method name does not have to follow Java's naming conventions; instead it is an uninterpreted Utf8 String.

To summarize, an invokedynamic instruction includes:

- An unnamed type signature.
- A number of regular (dynamic) method arguments. Those arguments will be passed to the method call after resolution.
- A reference to a bootstrap method which initializes the dynamic call site, effectively binding the site to a concrete method handle.
- An uninterpreted Utf8 string, which can be seen as the method name. This string is passed as an argument to the bootstrap method.
- An arbitrary number of additional static arguments. Those arguments are passed to the bootstrap method as well; they must be constants.

[1] http://jcp.org/en/jsr/detail?id=292

The information is stored in the bytecode's constant pool in a rather complex way, using various constant pool entries and a new attribute, `BootstrapMethods`. Soot's bytecode parser Coffi now parses these entries to create Jimple expressions from invokedynamic bytecodes. The `BootstrapMethods` attribute is stripped during this process; it will not occur as a tag on the corresponding SootClass. Instead, the attribute is re-generated when Jimple is converted back to bytecode. (see below)

Further information on the bytecode format can be found in the official package documentation for java.lang.invoke.

2 Jimple representation of invokedynamic bytecodes

In the Jimple IR, a invokedynamic bytecode is represented as a `JDynamicInvokeExpr`, whose constructor expects four arguments:

SootMethodRef bootstrapMethodRef A reference to a (static) bootstrap method. This method can reside in the same class as the invokedynamic instruction or in any other class. The bootstrap method receives from the virtual machine three implicit arguments, in addition to the explicit static arguments provided. A standard bootstrap method would start with the following argument signatures:

java.lang.invoke.MethodHandles$Lookup A lookup object representing the current class context.

java.lang.String The uninterpreted Utf8 string stated at the invokedynamic site.

java.lang.invoke.MethodType A method type object representing the resolved method type for this invokedynamic site.

List bootstrapArgs A number of static bootstrap arguments. Those arguments are passed to the bootstrap method in addition to the three implicit arguments mentioned above. The bootstrap method's signature must be defined accordingly. Those arguments must all be Jimple Constant objects.

SootMethodRef methodRef A method reference stating the name and type signature of this invokedynamic call site. The method name may be any valid Utf8 string. The sole purpose of this name is to be passed, by the virtual machine, as an argument to the provided bootstrap method. The type signature is used for type checking the invokedynamic call site in its invocation context. This is necessary to guarantee the integrity of the bytecode. In this signature, the name of the declaring class must be `soot.dummy.InvokeDynamic`. Soot will handle this special class as a phantom class (even if phantom classes are disabled otherwise.) The dummy class name is only used internally by Jimple as a placeholder; it is stripped when the `JDynamicInvokeExpression` is converted to bytecode, because invokedynamic instructions do not have a declaring class.

List methodArgs A number of regular, dynamic, method arguments in the form of Jimple Immediate objects.

Textual representation

The pretty printer will print a `JDynamicInvokeExpression` as follows (line breaks added to enhance readability):

```
dynamicinvoke
"+"
<java.lang.Object (java.lang.Object)>(r1)
<jsr292.cookbook.binop.RT: java.lang.invoke.CallSite
   bootstrapOpLeft(java.lang.invoke.MethodHandles$Lookup,
                   java.lang.String,java.lang.invoke.MethodType,int)>(1)
```

Here the string following the keyword `dynamicinvoke` is the uninterpreted Utf8 string denoting the call site's method name. This string appears as an actual string constant so that it can be parsed again by the Jimple parser. (Again, note that any Utf8 string is allowed!) On the next line, the expression contains an unnamed method signature, stripped of any declaring class name and method name (because these were given before). In the above example, `r1` is the dynamic argument to this call; according to the signature, it may hold any kind of `Object`. The last three lines contain a regular staticinvoke expression, representing the bootstrap method and the static constant arguments to invoke it at load time.

The Jimple parser fully supports this syntax, which means that Soot users may directly write Jimple files in the above syntax to create invokedynamic bytecodes.

3 Grimp, Grimple and Shimple IRs

In addition to Jimple, Soot also features the Grimp and Grimple IRs, which contain aggregated Jimple expressions, and the Shimple IR for SSA form. For all those IRs, the representation of invokedynamic expressions is the same as in Jimple.

4 Baf IR

Before Jimple is converted to bytecode, it is first converted into the stack-based Baf IR. During this conversion, Soot generates Baf code to push the instruction's dynamic arguments onto the stack, followed by a `BDynamicInvokeInst` that holds the remainder of the information, i.e., the Utf8 method name, the reference to the bootstrap method and its static arguments. In analogy to the textual Jimple format, the textual Baf format would consist of the following two instructions:

```
load.r r0;

dynamicinvoke
"+"
<java.lang.Object (java.lang.Object)>
<jsr292.cookbook.binop.RT: java.lang.invoke.CallSite
    bootstrapOpLeft(java.lang.invoke.MethodHandles$Lookup,
                    java.lang.String,java.lang.invoke.MethodType,int)>(1);
```

Note that Baf is currently an output-only IR, i.e., there is no parser that could parse Baf instructions back into Soot.

5 Jasmin

The last step before the final conversion to bytecode is to convert Baf to Jasmin, a textual assembler for Java bytecode. Soot uses a fork of the Jasmin project; Jasmin is no longer maintained. Jasmin differs from Baf in that it is very close to the bytecode, e.g., uses the same format for class references. Jasmin was extended to accept invokedynamic instructions in the following textual format:

```
invokedynamic
"+"
(Ljava/lang/Object;)Ljava/lang/Object;
jsr292/cookbook/binop/RT/bootstrapOpLeft(Ljava/lang/invoke/MethodHandles$Lookup;
                                         Ljava/lang/String;Ljava/lang/invoke/MethodType;I)
                                            Ljava/lang/invoke/CallSite;
                                         ((I)1)
```

This is quite similar to Baf. The only noteworthy difference is that the static arguments to the bootstrap method are all preceded to a bracketed type identifier, in the example `(I)`. This is necessary to allow Jasmin to determine which kind of class constant to generate as a bytecode representation of that constant value. In the above example, `I` would indicate an int.

6 Current limitations and pitfalls

The current support for JSR292 in Soot is fairly complete. However, a few items are missing. First, JSR292 allows constant arguments to bootstrap methods in the form of method handle or method type constants. Soot does not currently support those constants; such support could be added. Adding the support would mostly complicate the Jimple and Jasmin parsers, which would need to cope with some form of textual representation of method handles in argument positions.

Bytecode constants of type `CONSTANT_MethodType_info` are not currently supported. However, those do not yet appear to be used anyway; they may become important with Java 8's support for lambda functions.

Current pitfalls include the odd argument passing to bootstrap methods. Remember that a bootstrap method always receives three implicit arguments from the virtual machine. Those arguments are of type `Lookup`, `String`, and `MethodType`. This does not mean, however, that one can rely on the bootstrap method's signature to start with those argument types. Instead, a bootstrap method could consume any number of those arguments through a varargs argument, as in:

```
CallSite bootstrap(Object caller, Object... nameAndTypeWithArgs)
```

For this reason, Soot currently does not check the correctness of the bootstrap method's arguments against the provided arguments. Users hence have to be careful to provide a correct signature. Note that we *do* check, however, that the correct return type, `java.lang.invoke.CallSite`, is provided.

Acknowledgements Thanks to Matthias Perner for providing an initial implementation of invokedynamic based on the JDK 7 beta. Thanks to Andreas Sewe and Rémi Forax for answering some questions on MethodType attributes.