Adding profiling instructions to applications with Soot

Feng Qian (fqian@sable.mcgill.ca)
Patrick Lam (plam@sable.mcgill.ca)

March 8, 2000

This tutorial is based on the `countgotos` example, written by Raja-Vallée-Rai and distributed with Ashes. It is located in `ashes.examples.countgotos` package. The code can be downloaded at

http://www.sable.mcgill.ca/soot/tutorial/profiler/Main.java

At this stage, the developer should have a basic knowledge of Soot, including the SootClass, SootMethod and Unit classes. They are described in the document about creating a class using Soot.

1 Goals

This tutorial describes how to write a `BodyTransformer` which annotates `JimpleBody`'s with a goto-counter. In particular, the developer will be able to write code to:

- Retrieve a desired method from the `Scene` by signature.
- Add a field to a class file.
- Differentiate between various types of Jimple statements.
- Insert Jimple instructions at a certain point.

The `GotoInstrumenter` example instruments a class or application to print out the number of `goto` bytecodes executed at run time.

2 Creating a GotoInstrumenter

We will instrument a class to print out the number of goto instructions executed at run time. The general strategy is:

1. Add a static field `gotoCount` to the main class.
2. Insert instructions incrementing `gotoCount` before each `goto` instruction in each method.
3. Insert `gotoCount` print-out instructions before each return statement in 'main' method.
4. Insert `gotoCount` print-out statements before each `System.exit()` invocation in each method.

Once we create a `BodyTransformer` class and add it to the appropriate `Pack`, it will be invoked for each `Body` in the program.

3 Subclassing `BodyTransformer`

This example works by creating a `Transformer` which is added to the appropriate pack. We thus declare a subclass of `BodyTransformer` to carry out our instrumentation:

```
public class GotoInstrumenter extends BodyTransformer
{
    private static GotoInstrumenter instance = new GotoInstrumenter();
    private GotoInstrumenter() {}

    public static GotoInstrumenter v() { return instance; }
```

The above code creates a private static instance and an accessor to that instance.

```
    protected void internalTransform(Body body, String phaseName, Map options)
    {
```

Every `BodyTransformer` must declare some `internalTransform` method, carrying out the transformation.

4 Adding a Field

We already know how to add locals to a method body; this is seen in the `createclass` example. We now show how to add a field to a class.

Here, we want to add a counter field to the main class.

4.1 Sanity check – find main() method

First of all, we check the 'main' method declaration by its subsignature.

```
    if (!Scene.v().getMainClass().
              declaresMethod("void main(java.lang.String[])"))
        throw new RuntimeException("couldn't find main() in mainClass");
```

A couple of notes about this snippet of code. First, note that we call `Scene.v().getMainClass()`. This returns the `Scene`'s idea of the main class; in application mode, it is the file specified on the command-line, and in single-file mode, it is the last file specified on the command-line. Also, note that if we fail, then a `RuntimeException` is thrown. It is not worthwhile to use a checked exception in this case.

The `main` class for a Java program will always have subsignature (the Soot word for a complete method signature) `.void main(java.lang.String[])`. The call to `declaresMethod` returns true if a method with this subsignature is declared in this class.

4.2 Fetching or adding the field

Now, if we've already added the field, we need only fetch it:

```
    if (addedFieldToMainClassAndLoadedPrintStream)
        gotoCounter = Scene.v().getMainClass().getFieldByName("gotoCount");
```

Otherwise, we need to add it.

```
else
{
    // Add gotoCounter field
    gotoCounter = new SootField("gotoCount", LongType.v(),
                                    Modifier.STATIC);
    Scene.v().getMainClass().addField(gotoCounter);
```

Here, we create a new instance of `SootField`, for a static field containing a `long`, named `gotoCount`. This is the field which will be incremented each time we do a `goto`. We add it to the main class.

```
    // Just in case, resolve the PrintStream SootClass.
    Scene.v().loadClassAndSupport("java.io.PrintStream");
    javaIoPrintStream = Scene.v().getSootClass("java.io.PrintStream");

    addedFieldToMainClassAndLoadedPrintStream = true;
}
```

We will use the `java.io.PrintStream` method, so we load it just in case.

5 Add locals and statements

Recall that a `BodyTransformer` operates on an existing method body. In this step, locals are added to the body, and profiling structions are inserted while iterating over the statements of the body.

We first use the method's signature to check if it is a `main` method or not:

```
boolean isMainMethod = body.getMethod().getSubSignature()
            .equals("void main(java.lang.String[])");
```

We could also check to see if `body.getMethod().getDeclaringClass()` is the main class, but we don't bother.

Next, a local is added; we already know how to do this.

```
Local tmpLocal = Jimple.v().newLocal("tmp", LongType.v());
body.getLocals().add(tmpLocal);
```

Here, we are inserting statements at certain program points. We look for specific statements by iterating over the `Units` chain; in Jimple, this chain is filled with `Stmts`.

```
Iterator stmtIt = body.getUnits().snapshotIterator();
while (stmtIt.hasNext())
{
    Stmt s = (Stmt) stmtIt.next();
    if (s instanceof GotoStmt)
    {
        /* Insert profiling instructions before s. */
    }
    else if (s instanceof InvokeStmt)
    {
        /* Check if it is a System.exit() statement.
         * If it is, insert print-out statement before s.
         */
    }
    else if (isMainMethod && (s instanceof ReturnStmt
```

```
                    || s instanceof ReturnVoidStmt))
        {
            /* In the main method, before the return statement, insert
             * print-out statements.
             */
        }
    }
```

The call to `getUnits()` is akin to that in the `createclass` example. It returns a Chain of statements.

The `snapshotIterator()` returns an iterator over the Chain, but modification of the underlying chain is permitted. A usual iterator would throw a `ConcurrentModificationException` in that case!

We can determine the statement type by checking its class with `instanceof`. Here, we are looking at four different statement types: `GotoStmt`, `InvokeStmt`, `ReturnStmt` and `ReturnVoidStmt`.

Before every `GotoStmt`, we insert instructions that increase the counter. The instructions in Jimple are:

```
tmpLong = <classname: long gotoCount>;
tmpLong = tmpLong + 1L;
<classname: long gotoCount> = tmpLong;
```

Creating a reference to a static field is done via a call to `Jimple.v().newStaticFieldRef(gotoCounter.makeRef())`. The entire assignment statement is created with the `newAssignStmt` method.

```
AssignStmt toAdd1 = Jimple.v().newAssignStmt(tmpLong,
                                Jimple.v().newStaticFieldRef(gotoCounter.makeRef()));
```

The new statements can then be added to the body by invoking the `insertBefore()` method. There are also some other methods that can add statements to a body. We have seen one of them in `createclass` example, `add()`. Note that above, we need to get a SootFieldRef from the SootField gotoCounter, in order to construct the Jimple StaticFieldRef grammar chunk properly.

```
units.insertBefore(toAdd1, s);
```

We have thus added profiling instructions before every `goto` statement.

It is quite dandy to keep counters; they are useless unless outputted. We add printing statements before calls to `System.exit()`. This is done similarly to what we did for `goto` statements, except that we will look more deeply into the Jimple statements and expressions.

```
InvokeExpr iexpr = (InvokeExpr) ((InvokeStmt)s).getInvokeExpr();
if (iexpr instanceof StaticInvokeExpr)
{
    SootMethod target = ((StaticInvokeExpr)iexpr).getMethod();
    if (target.getSignature().equals
                ("<java.lang.System: void exit(int)>"))
    {
        /* insert printing statements here */
    }
}
```

Every `InvokeStmt` has an `InvokeExpr`. The `InvokeExpr` must be able to return the target method. Again, we can use signatures to test for the wanted method.

We already saw how to make printing statements in `createclass` example. Here is the generated Jimple code.

```
tmpRef = <java.lang.System: java.io.PrintStream out>;
tmpLong = <test: long gotoCount>;
virtualinvoke tmpRef.<java.io.PrintStream: void println(long)>(tmpLong);
```

In the `main()` method, we must also insert the same statements before each `return` statement.

6 Outputting annotated code

Since we are providing a `BodyTransformer`, the modified `Body` is treated as input to the next phase of Soot, and outputted at the end, as per the Soot options.

7 Adding this transformation to Soot

The preferred method of adding a transformation to Soot is by providing a `Main` class in one's own package. This class adds transformers to `Packs`, as needed. It then calls `soot.Main.main` with the arguments it has been passed. This is demonstrated in `ashes.examples.countgotos.Main`.

8 Conclusions

In this tutorial, we have seen how to instrument class files in Soot. Usually, anything we want to do can be viewed as a transformer of class files. Here, we used more advanced methods than in the `createclass` example.

Where to find files

The `GotoInstrumenter`, as described in this document (in `BodyTransformer` form) is available in the `ashesBase` package. There is one Java file, `Main.java`, which contains the `Main` and `GotoInstrumenter` classes. Full class names are `ashes.examples.countgotos.Main` and `.GotoInstrumenter`.

Change Log

- March 9, 2000: Initial version.
- March 23, 2000: Changes reflecting that the gotocounter is distributed with ashes now.
- Feb 7, 2005: Update calls to Jimple constructors to pass SootFieldRef's and SootMethodRef's where appropriate, using makeRef().