On the Soot menagerie – fundamental Soot objects

Patrick Lam (plam@sable.mcgill.ca)

March 1, 2000

Soot has a large and complicated class hierarchy. This document will introduce the reader to some of the most important classes for developing extensions to Soot.

This document is meant to be read after the `createclass` document is understood. It builds on the knowledge gained from that example. We describe here the notions of `Body`, `Unit`, `Local`, `Value`, `UnitBox` and `ValueBox`.

1 All about `Bodys`

In the `createclass` tutorial, the concept of a `Body` was introduced briefly. This section will describe `Body` in more detail.

Recapping from the previous lesson, Soot uses a `Body` to store code for a method. There are three kinds of `Body` in Soot – namely, `BafBody`, `JimpleBody` and `GrimpBody` – one for each intermediate representation.

Also, recall that a chain is a list-like data structure providing constant-time access to chain elements, including insertion and removal.

The three principal chains in a `Body` are the `Units` chain, the `Locals` chain and the `Traps` chain. The following example will illustrate the role of each of these chains.

Consider the following Java method:

```
public static void main(String[] argv) throws Exception
{
    int x = 2, y = 6;

    System.out.println("Hi!");
    System.out.println(x * y + y);
    try
    {
        int z = y * x;
    }
    catch (Exception e)
    {
        throw e;
    }
}
```

After Jimplification, we have the following abbreviated jimple code:

```
public static void main(java.lang.String[]) throws java.lang.Exception
    {
        java.lang.String[] r0;
        int i0, i1, i2, $i3, $i4;
        java.io.PrintStream $r1, $r2;
```

```
        java.lang.Exception $r3, r4;

        r0 := @parameter0;
        i0 = 2;
        i1 = 6;
        $r1 = java.lang.System.out;
        $r1.println(``Hi!'');
        $r2 = java.lang.System.out;
        $i3 = i0 * i1;
        $i4 = $i3 + i1;
        $r2.println($i4);

    label0:
        i2 = i1 * i0;

    label1:
        goto label3;

    label2:
        $r3 := @caughtexception;
        r4 = $r3;
        throw r4;

    label3:
        return;

        catch java.lang.Exception from label0 to label1 with label2;
    }
```

1.1 Local variables

The locals in this method are seen at the top of the method:

```
        java.lang.String[] r0;
        int i0, i1, i2, $i3, $i4;
        java.io.PrintStream $r1, $r2;
        java.lang.Exception $r3, r4;
```

The collection of `Local`s is stored in the `localChain` and accessible via `body.getLocals()`. Each intermediate representation may define its own implementation of `Local`; however, it must always be possible, for every `Local r0`, to call `r0.getName()`, `r0.getType()`, `r0.setName()` and `r0.setType`. Note that local variables must be typed.

1.2 Traps

To support Java exception handling, Soot `Body`'s define the notion of *traps*. The idea is that in Java bytecode, exception handlers are represented by a tuple (exception, start, stop, handler); between the start and stop units (including start, but not including stop), if the exception is thrown, execution continues at handler.

In the example, there is one trap:

```
        catch java.lang.Exception from label0 to label1 with label2;
```

1.3 Units

The most interesting part of a `Body` is its chain of `Unit`s. This is the actual code contained in the `Body`. Jimple provides the `Stmt` implementation of `Unit`, while Grimp provides the `Inst` implementation. This reflects the fact that each IR has its own notion of statement.

An example of a Jimple `Stmt` is the `AssignStmt`, which represents a Jimple assignment statement. One `AssignStmt` would be:

```
x = y + z;
```

After we describe `Box`es, we will discuss the methods specified by `Unit`.

2 Value

Code always acts on data. To represent the data, Soot provides the `Value` interface. Some different types of `Value`s are:

- Locals
- Constants
- Expressions (`Expr`)
- ParameterRefs, CaughtExceptionRefs and ThisRefs.

The `Expr` interface, in turn, has a panoply of implementations; among them are `NewExpr` and `AddExpr`. In general, an `Expr` carries out some action on one or several `Value`s and returns another `Value`.

Here's a real live use of some `Value`s:

```
x = y + 2;
```

This is an `AssignStmt`. Its `leftOp` is "x" and its rightOp is "y+2", an `AddExpr`. The `AddExpr`, in turn, contains the `Value`s "y" and "2" as its operands; the former is a `Local` while the latter is a `Constant`.

In Jimple, we enforce the requirement that all `Value`s contain at most 1 expression. Grimp lifts this restriction, producing easier-to-read but harder-to-analyse code.

3 Boxes

Boxes are ubiquitous in Soot. The main idea to keep in mind is that *a Box is a pointer*. It provides indirect access to Soot objects.

A more descriptive name for `Box` would have been `Ref`. Unfortunately, `Ref` has a different meaning for Soot.

There are two kinds of `Box`es in Soot – the `ValueBox` and the `UnitBox`. Not surprisingly, a `UnitBox` contains `Unit`s while a `ValueBox` contains `Value`s. In C++, these would be simply `(Unit *)` and `(Value *)` respectively.

We now describe each type of `Box`.

3.1 The UnitBox

Some types of `Unit`s will need to contain references to other `Unit`s. For instance, a `GotoStmt` needs to know what its target is. Hence, Soot provides the `UnitBox`, a `Box` that contains a `Unit`.

Consider the following `jimp` code:

```
    x = 5;
    goto l2;
    y = 3;
l2: z = 9;
```

Each `Unit` must provide `getUnitBoxes()`. For most `UnitBoxes`, this returns the empty list. However, in the cast of a `GotoStmt`, then `getUnitBoxes()` returns a one-element list, containing a `Box` pointing to `l2`.

Note that a `SwitchStmt` will, in general, return a list with many boxes.

The notion of a Box comes in especially useful for modifying code. Say we have a statement `s`:

```
  s: goto l2;
```

and a stmt at `l2`:

```
l2:  goto l3;
```

It is clear that `s` can point to `l3` instead of `l2`, regardless of the actual type of `s`; we can do this uniformly, for all kinds of `Unit`s:

```
public void readjustJumps(Unit s, Unit oldU, Unit newU)
{
    Iterator ubIt = s.getUnitBoxes.iterator();
    while (ubIt.hasNext())
    {
        StmtBox tb = (StmtBox)ubIt.next();
        Stmt targ = (Stmt)tb.getUnit();

        if (targ == oldU)
            tb.setUnit(newU);
    }
}
```

Some code similar to this is used in `Unit` itself, to enable the creation of `PatchingChain`, an implementation of `Chain` which adjusts pointers to `Unit`s which get removed from the `Chain`.

3.2 The ValueBox

Analogously to `Unit`s, we often need a notion of a "pointer to a `Value`". This is represented by the `ValueBox` class. For a `Unit`, we can always get a list of `ValueBox`es, containing values *used* and *defined* in that `Unit`.

We can use these boxes to carry out constant folding: if an `AssignStmt` evaluates an `AddExpr` adding two constant values, we can statically add them and put the result into the `UseBox`.

Here is an example of folding `AddExpr`s.

```
public void foldAdds(Unit u)
{
    Iterator ubIt = u.getUseBoxes().iterator();
    while (ubIt.hasNext())
    {
        ValueBox vb = (ValueBox) ubIt.next();
        Value v = vb.getValue();
        if (v instanceof AddExpr)
        {
            AddExpr ae = (AddExpr) v;
            Value lo = ae.getOp1(), ro = ae.getOp2();
```

```
                if (lo instanceof IntConstant && ro instanceof IntConstant)
                {
                    IntConstant l = (IntConstant) lo,
                          r = (IntConstant) ro;
                    int sum = l.value + r.value;
                    vb.setValue(IntConstant.v(sum));
                }
            }
        }
    }
```

Note how this works for *any* `Unit`, regardless of type.

4 `Unit` revisited

We now discuss the different methods that any `Unit` must provide.

```
public List getUseBoxes();
public List getDefBoxes();
public List getUseAndDefBoxes();
```

These methods return `List`s of `ValueBox`es for values used, defined, or both, in this `Unit`. For the `getUseBoxes()` method, all values used are returned; this includes expressions as well as their constituent parts.

```
public List getUnitBoxes();
```

This method returns a `List` of `UnitBox`es for units pointed to by this method.

```
public List getBoxesPointingToThis();
```

This method returns a `List` of `UnitBox`es which contain this `Unit` as their target.

```
public boolean fallsThrough();
public boolean branches();
```

These methods have to do with the flow of execution after this `Unit`. The former method returns true if execution can continue to the following `Unit`, while the latter returns true if execution might continue to some `Unit` which isn't immediately after this one.

```
public void redirectJumpsToThisTo(Unit newLocation);
```

This method uses `getBoxesPointingToThis` to change all jumps to this `Unit`, pointing them instead at `newLocation`.

5 History

- March 1, 2000: Initial version.
- September 1, 2000: Fixed typo.
- January 7, 2005: Made .jimp example match actual Soot output