Using the Soot flow analysis framework

Patrick Lam (plam@sable.mcgill.ca)

March 17, 2000

Slides from a talk on the Soot flow analysis framework are at http://www.sable.mcgill.ca/soot/notes.

1 Goals

By the end of this lesson, the student should be able to:

- understand the Soot Flow Analysis class hierarchy
- use an auxiliary class to package flow analysis results for use
- use the hooks in Soot to output the results of custom analyses

2 Flow Sets

In dataflow analysis, we seek to associate some data with each node in the control-flow graph. In Soot, we represent this data as flow sets.

Typically, a flow set represents a set of facts. For reaching definitions, the flow sets are the sets of pairs (variable, program point). Soot defines the `FlowSet` interface to be the canonical flow object.

Soot also provides an implementation of `FlowSet`, `ArraySparseSet`. This represents the `FlowSet` using an array.

Often, we want a `FlowSet` with complementation; this is a `BoundedFlowSet`, and implemented by `ArrayPackedSet`. To speak of complementation, there must be some universe with respect to which we complement. When the universe is not implicit in the definition of the flow set itself, then Soot provides the `FlowUniverse` set; this is used by `ArrayPackedSet`.

The `FlowSet` interface requires the following methods:

- `clone()`
- `clear()`
- `isEmpty()`
- `copy (FlowSet dest)`
- `union (FlowSet other, FlowSet dest)`
- `intersection (FlowSet other, FlowSet dest)`
- `difference (FlowSet other, FlowSet dest)`

The first 3 methods are clear. The `copy()` method will put the contents of ‘this’ `FlowSet` into the destination `FlowSet`.

For the last 3 methods, the following rule is used:

```
dest <- this op other
```

Note that these operations give `FlowSet` the necessary structure to be a lattice element.

Usually, we just need to represent sets for flow analysis. In that case, the following optional methods are used:

- `size()`
- `add (Object obj, FlowSet dest)`
- `remove (Object obj, FlowSet dest)`
- `contains (Object obj)`
- `toList()`

The `add()` and `remove` objects use the following rule:

```
dest <- this op obj
```

The reference implementation, `ArraySparseSet`, implements all of the methods for `FlowSet`.

3 Graph creation

In order to do dataflow analysis, a control-flow graph is required. We will now describe how to create such a graph.

The `SootMethod` and `Body` classes have been described in the example showing how to create a class from scratch. These concepts should now be understood.

Soot provides the `UnitGraph` class to describe the notion of a control-flow graph. There are two types of `UnitGraphs`: `ExceptionalUnitGraph` and `BriefUnitGraph`. The exceptional graph contains all of the edges in the brief graph, plus edges corresponding to potential exceptions being thrown. It should be used for analysis.

The `UnitGraph` class implements the `DirectedGraph` interface, which captures the essential features of directed graphs (on `Directed` objects).

Given a `Body`, say `b`, we can create a `ExceptionalUnitGraph` by invoking `new ExceptionalUnitGraph(b)`.

4 Flow Analysis

Now that we are armed with graphs and flow sets, we can proceed to carry out the actual analysis.

Soot provides `FlowAnalysis` classes. All we need to plug in are a flow function, merge, copy, and initial flow sets, and we're all set!

The following abstract methods must be implemented by a `FlowAnalysis`:

`newInitialFlow()`: return the initial value for `FlowSet`s in the graph.

e.g. "`return emptySet.clone();`"

`customizeInitialFlowGraph()`: overriding `customizeInitialFlowGraph()` will permit the user to give different flow sets to different graph nodes. This method can adjust anything it needs to; it is called at the end of the constructor for `Analysis`. For instance, an all-paths analysis will often make the initial objects the full set, except at the start.

`merge(inSet1, inSet2, outSet)`: combine two `FlowSet`s to produce an out-`FlowSet`.

e.g. "`inSet1.union(inSet2, outSet);`"

`copy(sourceSet, destSet)`: put the source into the destination.

e.g. "`sourceSet.copy(destSet);`"

`flowThrough(inSet, s, outSet)`: given `inSet` and `s`, put the correct OUT value into the `outSet`.

If we have pre-computed the gen and preserve sets, this code could implement `flowThrough()`:

```
inSet.intersection(unitToPreserveSet.get(s), outSet);
outSet.union(unitToGenerateSet.get(s), outSet);
```

In appendix 9 we show a complete example of a simple analysis, for detecting live locals.

4.1 Pre-computing gen and preserve Sets

We made a passing reference to pre-computing sets in the above. Often, in a flow analysis, the flow-through function is actually quite simple:

$$\mathrm{OUT}(s) = \mathrm{IN}(s) \cup \mathrm{gen}(s) \cap \mathrm{prsv}(s)$$

In such a case, we can pre-compute the gen and preserve sets in the constructor. Hopefully, this speeds up the analysis.

We illustrate the pre-computation of a preserve set for live locals:

```
Unit s = (Unit) unitIt.next();

BoundedFlowSet killSet = emptySet.clone();
Iterator boxIt = s.getDefBoxes().iterator();

while(boxIt.hasNext())
{
    ValueBox box = (ValueBox) boxIt.next();

    if(box.getValue() instanceof Local)
        killSet.add(box.getValue(), killSet);
}

// Store complement
    killSet.complement(killSet);
    unitToPreserveSet.put(s, killSet);
```

Note that all `Unit` objects provide a `getDefBoxes` method. This returns a list of values defined in the `Unit`.

In order to compute the `gen` sets, we use `getUseBoxes` instead; it is a fairly simple change.

5 Packaging Flow Analyses

Typically, we don't want to provide access to the underlying Analysis classes. For instance, we would much rather pass around `Lists` of live locals, rather than `FlowSets`; we can make the `Lists` unmodifiable, while we're at it!

In order to do that, we wrap the `Analysis` object. After running the Analysis, we run through the units and map the `Units` in question to `Lists` of results. Then, we can provide accessor methods:

```
List lla = liveLocals.getLiveLocalsAfter(s);
```

An example wrapper is in appendix A.

We now have clean access to the analysis results.

6 Transforming Jimple

Often, we want to transform all of the JimpleBody objects for a program. This can be done, first, by creating a BodyTransformer object.

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

    public static NaiveCommonSubexpressionEliminator v() { return instance; }

    /** Common subexpression eliminator. */
    protected void internalTransform(Body b, String phaseName, Map options)
    {
```

The most important part of this class is the internalTransform method. It carries out the work of the transformer. There are also *declared* options – those that the transformer claims to understand; and *default* options.

The code fragment above also has code to provide a singleton object, so that we may refer to the common subexpression eliminator as NaiveCommonSubexpressionEliminator.v(), which is a Java object.

Once we have done this, we want to ensure that the transformation is triggered at the appropriate times. Soot runs a number of Packs at different stages of its execution; they are built in the PackManager's constructor. One notable Pack is the Jimple transformation pack (jtp); the user may wish to add transformations to this pack:

```
        PackManager.v().getPack("jtp").add(
             new Transform("jtp.gotoinstrumenter", GotoInstrumenter.v()));
```

The Transform object just keeps track of the pair (phase name, transformation object). Phases are described in more detail in the document about phase options.

7 Extending Soot

The approved way of extending Soot is to provide a Main class file in a custom package, say plam.Main. This class can make adjustments to the Packs contained in the PackManager, for instance adding a goto instrumentor:

```
    public static void main(String[] args)
    {
        if(args.length == 0)
        {
            System.out.println("Syntax: java "+
                "soot.examples.instrumentclass.Main --app mainClass "+
                "[soot options]");
            System.exit(0);
        }

        PackManager.v().getPack("jtp").add(
             new Transform("jtp.gotoinstrumenter", GotoInstrumenter.v()));
        soot.Main.main(args);
    }
```

8 Conclusions

In this lesson, we have described the Soot flow analysis framework, described how to write a Body transformer, and how to integrate all of this into the Soot framework.

9 History

- March 17, 2000: Initial version.
- May 31, 2003: Initial version.

Simple Live Locals Analysis

```
/* Soot - a J*va Optimization Framework
 * Copyright (C) 1997-1999 Raja Vallee-Rai
 *
 * Licensed under LGPL. */

package soot.toolkits.scalar;

import soot.*;
import soot.util.*;
import java.util.*;
import soot.jimple.*;
import soot.toolkits.graph.*;

class SimpleLiveLocalsAnalysis extends BackwardFlowAnalysis
{
    FlowSet emptySet;
    Map unitToGenerateSet;
    Map unitToPreserveSet;

    protected Object newInitialFlow()
    {
        return emptySet.clone();
    }

    protected void flowThrough(Object inValue, Directed unit, Object outValue)
    {
        FlowSet in = (FlowSet) inValue, out = (FlowSet) outValue;

        // Perform kill
            in.intersection((FlowSet) unitToPreserveSet.get(unit), out);

        // Perform generation
            out.union((FlowSet) unitToGenerateSet.get(unit), out);
    }

    protected void merge(Object in1, Object in2, Object out)
    {
        FlowSet inSet1 = (FlowSet) in1,
            inSet2 = (FlowSet) in2;

        FlowSet outSet = (FlowSet) out;

        inSet1.union(inSet2, outSet);
    }

    protected void copy(Object source, Object dest)
    {
        FlowSet sourceSet = (FlowSet) source,
            destSet = (FlowSet) dest;

        sourceSet.copy(destSet);
    }

    SimpleLiveLocalsAnalysis(UnitGraph g)
    {
        super(g);

        // Generate list of locals and empty set
        {
            Chain locals = g.getBody().getLocals();
            FlowUniverse localUniverse = new FlowUniverse(locals.toArray());

            emptySet = new ArrayPackedSet(localUniverse);
        }

        // Create preserve sets.
        {
            unitToPreserveSet = new HashMap(g.size() * 2 + 1, 0.7f);
            Iterator unitIt = g.iterator();

            while(unitIt.hasNext())
            {
                Unit s = (Unit) unitIt.next();

                BoundedFlowSet killSet = (BoundedFlowSet) emptySet.clone();
                Iterator boxIt = s.getDefBoxes().iterator();

                while(boxIt.hasNext())
                {
                    ValueBox box = (ValueBox) boxIt.next();

                    if(box.getValue() instanceof Local)
                        killSet.add(box.getValue(), killSet);
                }

                // Store complement
                    killSet.complement(killSet);
                    unitToPreserveSet.put(s, killSet);
            }
        }

        // Create generate sets
        {
            unitToGenerateSet = new HashMap(g.size() * 2 + 1, 0.7f);
            Iterator unitIt = g.iterator();

            while(unitIt.hasNext())
            {
                Unit s = (Unit) unitIt.next();

                FlowSet genSet = (FlowSet) emptySet.clone();
                Iterator boxIt = s.getUseBoxes().iterator();

                while(boxIt.hasNext())
                {
                    ValueBox box = (ValueBox) boxIt.next();

                    if(box.getValue() instanceof Local)
                        genSet.add(box.getValue(), genSet);
                }

                unitToGenerateSet.put(s, genSet);
            }
        }

        doAnalysis();
    }
}
```

A Simple Live Locals Analysis Wrapper

```
/* Soot - a J*va Optimization Framework
 * Copyright (C) 1997-1999 Raja Vallee-Rai
 *
 * Licensed under LGPL. */

package soot.toolkits.scalar;

import soot.*;
import soot.util.*;
import java.util.*;
import soot.jimple.*;
import soot.toolkits.graph.*;

/** Wrapper for Analysis class. */
public class SimpleLiveLocals implements LiveLocals
{
    Map unitToLocalsAfter;
    Map unitToLocalsBefore;
    public SimpleLiveLocals(ExceptionalUnitGraph graph)
    {
        SimpleLiveLocalsAnalysis analysis = new SimpleLiveLocalsAnalysis(graph);

        // Build unitToLocals map
        {
            unitToLocalsAfter = new HashMap(graph.size() * 2 + 1, 0.7f);
            unitToLocalsBefore = new HashMap(graph.size() * 2 + 1, 0.7f);

            Iterator unitIt = graph.iterator();

            while(unitIt.hasNext())
            {
                Unit s = (Unit) unitIt.next();

                FlowSet set = (FlowSet) analysis.getFlowBefore(s);
                unitToLocalsBefore.put(s,
                                   Collections.unmodifiableList(set.toList()));

                set = (FlowSet) analysis.getFlowAfter(s);
                unitToLocalsAfter.put(s,
                                   Collections.unmodifiableList(set.toList()));
            }
        }
    }

    public List getLiveLocalsAfter(Unit s)
    {
        return (List) unitToLocalsAfter.get(s);
    }

    public List getLiveLocalsBefore(Unit s)
    {
        return (List) unitToLocalsBefore.get(s);
    }
}
```