Design of IFDS Analysis

Hampus Balldin, Emil Hammarström

October 18, 2018

1 Design

We have implemented a May analysis in SOOT. The analysis is implemented as a whole-program-analysis transformer. It is based on the IFDS framework; we subclass DefaultJimpleIFDSTabulationProblem.

May Analysis
Given that uses of remove or contains on instances of ArrayList or LinkedList are to be considered bad practice, any code that may use that pattern should be flagged (it is a static property, not a dynamic one). Thus we use a May-analysis and not a Must-analysis.

1.0.1 IFDS

Background
IFDS computes so called facts associated with each node; the facts at a given node hold only at that node. The transfer functions in IFDS propagates these facts to successor nodes. A fact is thus a property (at a node) and the transfer functions describe what implications between properties hold. A merge in IFDS then corresponds to logical OR. The expressible transfer functions are implications from (NODE, PROPERTY) to (NODE', {PROPERTY}); i.e. a single property at a given node can give multiple properties at a successor node. Since we may not combine multiple properties at a node to form new properties, we can only compute problems that fit into a distributive framework. The always true property is denoted **0**.

Tracking References
The property that we use is: P :: Value -> Bool, $P\ v$ = "Value v is a reference to ArrayList or LinkedList".

For example, at a node $n : [v \leftarrow newArrayList()]$ then $\mathbf{0} \implies P\ v$ at n. Similarly, at a definition statement, $[x \leftarrow y]$ then $P\ y \implies P\ x$.

Similar tracking is done across procedure calls where implications are performed for the procedure arguments. Similarly, returned references from procedure calls are tracked.

At a use of a value v to perform a call to **contains** or **remove** s.t. $P\ v$ holds, then we add the value v to a set of found statements.

Limiting size of CallGraph
By default, the callgraph includes all methods that could be invoked. This leads to an unreasonably large graph. To limit the propagation to only the packages that we are interested in, we implemented a simple hack as shown below:

```
public class SimpleCallGraph extends JimpleBasedInterproceduralCFG {
    private String[] targetClasses = {
         "eda045f.exercises" ,"java.util"
        ,"org.javacc","org.apache"
    };

    public SimpleCallGraph() { super(false, false); }

    @Override
    public Collection<SootMethod> getCalleesOfCallAt(Unit u) {
        return super.getCalleesOfCallAt(u).stream().filter(sm -> {
            for(int i = 0; i != targetClasses.length; ++i)
                if(sm.getDeclaringClass().getName().startsWith(targetClasses[i]))
                    return true;
                return false;
            }).collect(Collectors.toList());
    }
}
```