Homework 3

Posted February 8, Due February 20

50 points

Now that you are (more) familiar with Soot and the Class analysis framework, you will build a more complex class analysis, 0-CFA. Add a new package `analysis.CFA` and add your 0-CFA implementation in `public class CFAAnalysis extends Analysis` in this package. You may add analogous drivers to the ones in RTA to test locally. Follow directory structure as Submitty pulls your `analysis/CFA/CFAAnalysis.java` to test.

Follow the declarative specification of 0-CFA we discussed in class. Recall that 0-CFA keeps sets for each (Jimple) reference variable in the program and although there are many ways to store analysis state, it is most convenient to store those sets as maps where keys are `String`s of fully-qualified variable names and values are sets of `SootClass`es. Keys are fully-qualified names as stored in `Node` objects in the framework. For example, the Jimple variable roughly corresponding to `a` in `p1`, gives rise to String `<A: void main(java.lang.String[])>@r2`. Hint: `getFromMap` can be especially useful when handling sets in `solve` functions.

Now you will need to collect information and schedule appropriate constraints in each one of the hooks exposed by the Class analysis framework:

(1) `allocStmt(SootMethod enclMethod, int allocSiteId, Node lhs, Node alloc)`
You will collect the allocated class as in RTA, however, you will need to grab the string representation of `lhs` as well as you will need to store the class into `lhs`'s set.

(2) `virtualCallStmt(SootMethod enclMethod, int callSiteId, Node lhs, SootMethod target, List<Node> args)`
You will need to grab argument and left-hand-side information as you will need to account for flow of values from arguments to formal parameters and from return variables to the left-hand-side of the call assignment in `solve`. Importantly, the receiver argument is stored in `args.get(0)`.

(3) `directCallStmt(SootMethod enclMethod, int callSiteId, Node lhs, SootMethod target, List<Node> args)`
Recall that this hook handles both `specialinvoke`'s and `staticinvoke`'s. For special invokes the receiver is stored into `args.get(0)`. As in the previous case, you will need to collect argument and left-hand-side information in order to account for flow into parameters and from returns in `solve`.

(4) `assignStmt(SootMethod enclMethod, Node lhs, Node rhs)`
You will now need to collect and solve assignment statement constraints as well. An important note here is that the framework captures static field reads and static fields writes as assignment statements. For example, a static field read `local_var = A.static_fld` is exposed as an assignment statement where the `rhs` `Node` object is of kind `STATIC_FIELD`. You do not need special handling of these cases as the string representation of the static field is sufficient to identify the field for the purposes of our 0-CFA analysis.

(5) `fieldWriteStmt(SootMethod enclMethod, Node lhs, SootField f, Node rhs)` and `fieldReadStmt(SootMethod enclMethod, Node lhs, Node rhs, SootField f)`
Recall that our 0-CFA associates a set with each field, where a field is identified by

the (class, field_identifier) tuple. You can use the string representation provided by `SootField` to identify the field for the purposes of our analysis.

(6) `arrayWriteStmt(SootMethod enclMethod, Node lhs, Node rhs)` and `arrayReadStmt(SootMethod enclMethod, Node lhs, Node rhs)`
Finally, you will now need handling of array reads and array writes. A standard (though generally imprecise) handling is to treat the array subscript operator `[]` as a special field, and we resort to this handling in 0-CFA. Therefore, for our purposes there is a single "blob" *array field* `[]`. Each array write will cause flow into the array field `[]`, and each array read will cause flow out of array field `[]`. Note that while this is safe, it is imprecise and will cause spurious flow in general.

Finally, as with RTA, display your result in `showResult`. Specifically, display all reachable methods m in alphabetical order followed by all *virtual calls* in reachable methods printed in alphabetical order with dynamic targets at each call. In cases with more than one dynamic target, print them alphabetically as well. For example, the expected output for `p1` is as follows:

```
Reachable methods:
<A: void <init>()>
<A: void m()>
<A: void main(java.lang.String[])>

Virtual calls:
<A: void main(java.lang.String[])> --> <A: void main(java.lang.String[])>@a.<A: void m()>
=== <A: void m()>
```

When you are done, push into your `hw02` repository and click Grade my Repository in Submitty. Submitty pulls your `analysis/CFA/CFAAnalysis.java` to test. Make sure you commit/push the entire `CFA` directory.