Homework 4
Posted Tuesday February 19, Due Monday February 25
20 points Extra homework credit
HW4 adds on following HW3. HW4 is OPTIONAL and you may work in teams.

1. Overview of Class Analysis Framework

Starter code in package analysis builds a Java class analysis framework on top of Soot. Code in Analysis.java abstracts away Jimple into 8 kinds of statements relevant to class analysis. Since we are interested in class analysis, i.e., flow of values of reference type, we ignore all statements and expressions on primitive types. All exposed variables (essentially all, there is one caveat) are of reference type.

1. Assignment: \( x = y \)
2. Field read: \( x = y.f \)
3. Field write: \( x.f = y \)
4. Array read: \( x = y[] \)
5. Array write: \( x[] = y \)
6. Object allocation: \( x = \text{new} A \)
7. Direct call: either static invoke \( x = \text{sm}(\text{args}) \) or special invoke \( x = y.m(\text{args}) \)
8. Virtual call: \( x = y.m(\text{args}) \)

Analysis.java includes a worklist-like algorithm for solving constraints/transfer functions. Your task is to encode constraints as needed for a specific analysis then fire up the algorithm to compute the fixpoint solution. In the first assignment, HW2, you will implement Rapid Type Analysis, which entails the simplest constraints (i.e., transfer functions) and dataflow information (i.e., lattice). In HW3 you will implement XTA, a more precise and slightly more complex analysis. (If you need an extra challenge you can code 0-CFA or PTA, i.e., points-to analysis, as well.) In HW4 you will move on from toy programs to larger Java programs and compare RTA to XTA with respect to call graph construction, essentially trying to reproduce the results of Tip and Palsberg’s OOPSLA’00 paper: “Scalable Propagation-Based Call Graph Construction Algorithms”.

Folder programs contains 7 toy Java programs. Run the analysis on each of these and carefully examine the Jimples created by Soot. The autograder will test your analysis on these programs, as well as some other simple programs.

2. HW2

Your first task is to code Rapid Type Analysis which we covered in class. Place your code in package analysis.RTA. Starter code for this package is already there in your repo. Study the code, as well as the rest of the framework. Places where you will be adding your code are marked as TODO: YOUR CODE HERE. (There are other TODOs scattered throughout the code; these are reminders for me to fix, eventually.)

Once you are done, print the results of RTA on the console. Your analysis.RTA.showResults should print all RTA-reachable methods, by full name in alphabetical order, followed by all instantiated classes, also in alphabetical order. For example, the expected output from one of the toy programs looks like this:
Reachable methods:
=== <A: int add(A)>
=== <A: void <init>()>
=== <A: void m()>
=== <A: void main(java.lang.String[])>
=== <A: void sm()>
=== <B: void <init>()>

Instantiated classes:
=== A
=== B

When you are done, push into your repository and click Submit in Submitty. I will be running your RTAAnalysis with a slightly different driver to compare your output with the expected output.

3. HW3

Now that you are (more) familiar with Soot and the Class Analysis framework, you will build a more complex class analysis, XTA.

Update your hw02 repository. I have committed 3 more toy programs to test with; they should be showing in directory in programs/p8, programs/p9 and programs/p10. Add a new package analysis.XTA and add your XTA implementation in public class XTAAnalysis 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/XTA/XTAAnalysis.java to test.

Below is a rehash of the XTA constraints we discussed in class.

1. Allocation:
   1: for each new A in m s.t. m ∈ ReachableMethods do
   2:     A ∈ S_m
   3: end for

2. Virtual call:
   1: for each x = y.n(z) in m s.t. m ∈ ReachableMethods do
   2:     for each C in S_m ∩ SubTypes(StaticType(y)) do
   3:         n’(this, p, ret) = resolve(C, n)
   4:         n’ ∈ ReachableMethods
   5:     C ∈ S_{n’} // add receiver class to S_{n’}
   6:     S_m ∩ SubTypes(StaticType(p)) ⊆ S_{n’} // propagate types to S_{n’}
   7:     S_{n’} ∩ SubTypes(StaticType(ret)) ⊆ S_m // propagate types from S_{n’} due to return
   8: end for
   9: end for

I have simplified the presentation stating that each method has exactly one formal parameter p. Of course, a method may have 0 or more parameters and you do need to handle this. Note that the class analysis framework passes all actual arguments to the analysis, including ones that are of primitive type. You will have to do some extra work to filter out parameters of primitive type. Soot API methods getParameterType and getReturnType in SootMethod may be of use.
3. Field Read:
1: for each x = y.f in m s.t. m ∈ ReachableMethods do
2: \( S_f \subseteq S_m \)
3: end for

4. Field Write:
1: for each x.f = y in m s.t. m ∈ ReachableMethods do
2: \( S_m \cap \text{SubTypes(StaticType(f))} \subseteq S_f \)
3: end for

In addition, you must handle direct calls, static fields and array reads/writes.
Handling direct calls that are static calls is unambiguous. When handling direct calls that are instance calls, use the following constraint to pass the type of the receiver to the callee n:

\[ S_m \cap \text{SubTypes(StaticType(this))} \subseteq S_n \]

Static field reads local = static_field and writes static_field = local are abstracted as assignStmt in the class analysis framework, where either the right-hand-side or left-hand-side node is of kind STATIC_FIELD. This design choice may demand separate Constraints for static fields even though handling of static and instance fields is exactly the same.

Finally, make sure you handle arrays. If you ignore arrays the XTA analysis will be unsound. Consider

```java
void m(X[] a) {
    X x = a[0];
    x.n();
}
```

and note that in general, the array argument may have been written anywhere in the program.

Finally, as with RTA, display your result in showResult. Specifically, display all reachable methods m in alphabetical order with all classes in \( S_m \) in alphabetical order. For example, the expected output for p2 is the following

Reachable methods:

```
<A: int add(A)>
  === A

=== B

<A: void <init>()>
  === A

=== B

<A: void m()>
  === A

=== B

<A: void main(java.lang.String[])>

<A: void sm()>
  === A

=== B

<B: void <init>()>
  === B
```
4. HW4

In this part, you will apply your RTA and XTA implementations on large(r) programs and measure precision improvement of XTA over RTA with respect to call graph construction.

4.1. Part 1. Download the 3 larger programs I have selected:

- **javad** is a Java disassembler
- **bloat** is a byte code optimizer and is part of the standard DaCapo benchmark suite
- **eclipssec** is the Eclipse Java compiler

Download the jars from the course website:

- [www.cs.rpi.edu/~milanova/csci4450/javad.jar](http://www.cs.rpi.edu/~milanova/csci4450/javad.jar)
- [www.cs.rpi.edu/~milanova/csci4450/bloat.jar](http://www.cs.rpi.edu/~milanova/csci4450/bloat.jar)
- [www.cs.rpi.edu/~milanova/csci4450/bloat-deps.jar](http://www.cs.rpi.edu/~milanova/csci4450/bloat-deps.jar)
- [www.cs.rpi.edu/~milanova/csci4450/eclipssec.jar](http://www.cs.rpi.edu/~milanova/csci4450/eclipssec.jar)

Note: These programs do not use reflection/dynamic loading (they do, but minimally, and its presence only minimally affects soundness of RTA/XTA). You have probably noticed that our implementation of RTA/XTA is unsound in the presence of reflection/dynamic loading. For example, we ignore statements

```
A x = (A) Class.newInstance(Class.forName(someString))
```

but they may create arbitrary subtypes of A! Reflection is a thorny issue in static analysis, and I have purposefully left it outside the scope of our class.

4.2. Part 2. Run your RTA and XTA implementations on these programs using the following command line arguments:

```java
String[] args = new String[9];
args[0] = "-app";
// 3 new arguments force Soot to include all java.* libraries in the analysis
args[1] = "-include";
args[2] = "java";
args[3] = "-allow-phantom-refs";
args[4] = "-f";
args[5] = "J";
args[6] = "-cp";
// (I have placed bloat into directory programs/bench)
// If you are on Windows, don't forget to change the path separator to ";"!
args[8] = "dacapo.bloat.Main2";
XTA.main(args);
```

The corresponding arguments for **eclipssec** and **javad** are respectively

```java
args[7] = ".src/programs/bench/eclipssec.jar:"+RT_HOME;
args[8] = "org.eclipse.jdt.internal.compiler.batch.Main";
```
and

```java
args[7] = ".src/programs/bench/javad.jar:"+RT_HOME;
args[8] = "main";
```
Now analysis scope includes not only the application itself, but also, all Java libraries that the application pulls in. You may encounter unexpected issues when switching from toy programs to larger ones. I had to optimize my implementation of XTA, particularly I had to implement more efficient computation of successor constraints, and more efficient computation of the subtyping relation. It is reasonable if your analysis completes within 200 seconds per benchmark.

4.3. **Part 3.** Once your implementation scales, run RTA and XTA and collect statistics over the 3 programs. Specifically, show the improvement (if any) of XTA over RTA in the number of reachable methods. In addition, count \( \text{numCallGraphEdges} \), the number of call graph edges according to RTA and XTA, and again, show the improvement of XTA over RTA. (XTA was a lot more work, so we have to have a decent improvement in call graph precision, right?)

The number of call graph edges is computed as follows. For each virtual call \( c_i: x = y.n(z) \) in reachable method \( m \), \( \text{numCallGraphEdges} = \text{numCallGraphEdges} + \text{numTargets}(c_i) \) where \( \text{numTargets}(c_i) \) is computed based on \( \mathcal{S}_m \) for XTA and the set of instantiated classes \( \mathcal{I} \) for RTA. As expected, each direct call in a reachable method contributes 1 edge to \( \text{numCallGraphEdges} \).

Write a 1-2 page paper summarizing your findings.

Perhaps the easiest way to run the analyses is to tack two transformers, then when Soot is done, solve each, and then collect statistics.

```java
// Tack two transformers and run Soot.
Analysis rtaAnalysis = new RTAAnalysis();
PackManager.v().getPack("jtp").add(new Transform("jtp.rta", rtaAnalysis));
Analysis xtaAnalysis = new XTAAnalysis();
PackManager.v().getPack("jtp").add(new Transform("jtp.xta", xtaAnalysis));
soot.Main.main(args);

// Run RTA then XTA
rtaAnalysis.worklistSolve();
xtaAnalysis.worklistSolve();

// Collect stats
```

4.4. **Part 4.** Submission. Work in one of your hw02 repositories, continuing on your RTA and XTA implementation from HW2 and HW3. Create a new directory, `src/reports/`, and place your report, titled `RTAvsXTA.pdf` in that directory. Finally, email the report to me; I won’t be setting up Submitty page for this homework.