Programming Assignment 3: Class Hierarchy Analysis

Course “Static Program Analysis” @Nanjing University
Assignments Designed by Tian Tan and Yue Li
Due: 23:00, Wednesday, December 2, 2020

1 Goal

In this programming assignment, you will implement a class hierarchy analysis and call graph construction for Java based on Bamboo. To show the usefulness of call graph, we provide an interprocedural constant propagation, which uses the call graph constructed by your analysis. If your implementation is correct, you can see that interprocedural constant propagation achieves better precision than intraprocedural counterpart (please see Section 3.7 for more details). Same as in Assignment 1, you only need to consider a small subset of Java features.

2 Introduction to Bamboo

Bamboo is a static program analysis framework developed by the two instructors of this course, and it supports multiple static analyses (e.g., data-flow analysis, pointer analysis, etc.) for Java. Bamboo leverages Soot as front-end to parse Java programs and construct IRs (Jimple). In this assignment, we include the necessary classes for class hierarchy analysis and call graph construction. In addition, we also include an interprocedural data-flow analysis framework and an interprocedural constant propagation to demonstrate the usefulness of call graph and interprocedural analysis. You will see other parts of Bamboo in the following assignments.

2.1 Content of Assignment

The content resides in folder `bamboo/`, which includes:

- `analyzed/`: The folder containing test input files.
- `libs/`: The folder containing Soot classes with its dependencies.
- `src/`: The folder containing the source code of Bamboo. **You will need to modify a file in this folder to finish this assignment**.
- `test/`: The folder containing test classes.
- `build.gradle`: The Gradle build script for Bamboo.
- `copyright.txt`: The copyright of Bamboo.

2.2 Setup Instructions (Same as Assignment 1)

Bamboo is written in Java, so it is cross-platform. To build and run Bamboo, you need to have Java **8** installed on your system (other Java versions are currently not supported). You could download the Java Development Kit 8 from the following link: https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html

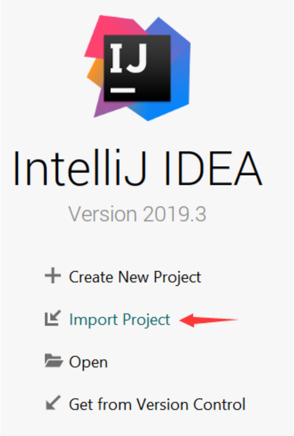
We highly recommend you to finish this (and the following) assignment(s) with IntelliJ IDEA. Given the Gradle build script, it is very easy to import Bamboo to IntelliJ IDEA, as follows.

Step 1

Download IntelliJ IDEA from JetBrains (http://www.jetbrains.com/idea/download/)

Step 2

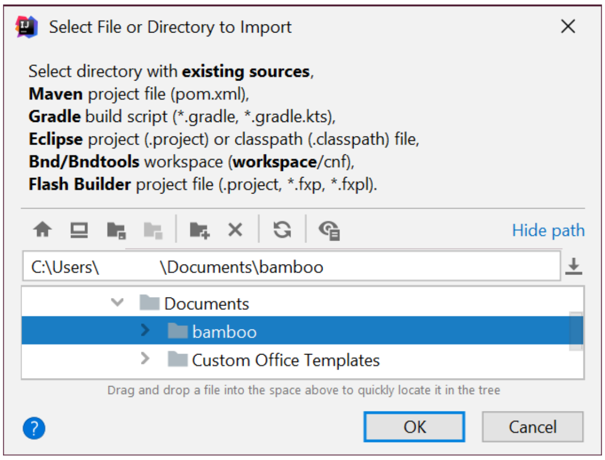
Start to import a project



(Note: if you have already used IntelliJ IDEA, and opened some projects, then you could choose File > New > Project from Existing Sources… to open the same dialog for the next step.)

Step 3

Select the `bamboo/` directory, then click "OK".



Select File or Directory to Import

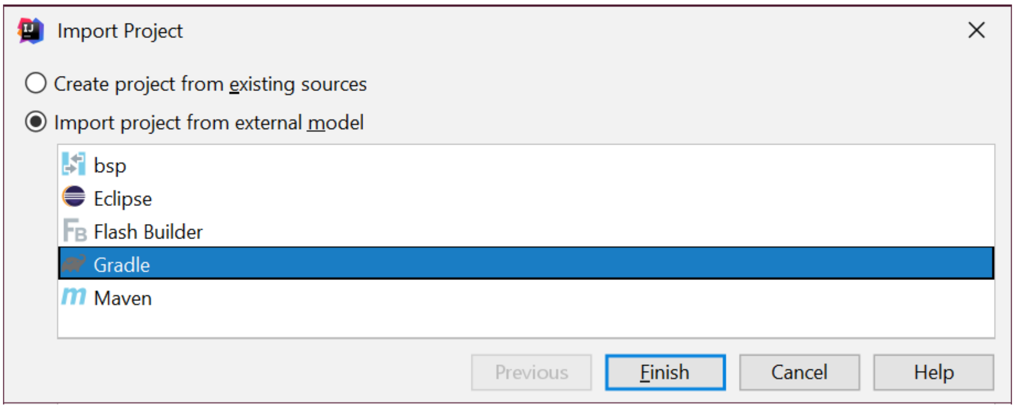
Select directory with **existing sources**,
Maven project file (pom.xml),
Gradle build script (*.gradle, *.gradle.kts),
Eclipse project (.project) or classpath (.classpath) file,
Bnd/Bndtools workspace (**workspace**/cnf),
Flash Builder project file (.project, *.fxp, *.fxpl).

C:\Users\ \Documents\bamboo

Hide path
Documents
bamboo
Custom Office Templates
Drag and drop a file into the space above to quickly locate it in the tree
?
OK
Cancel

Step 4

Import project from external model Gradle, then click "Finish".



Import Project

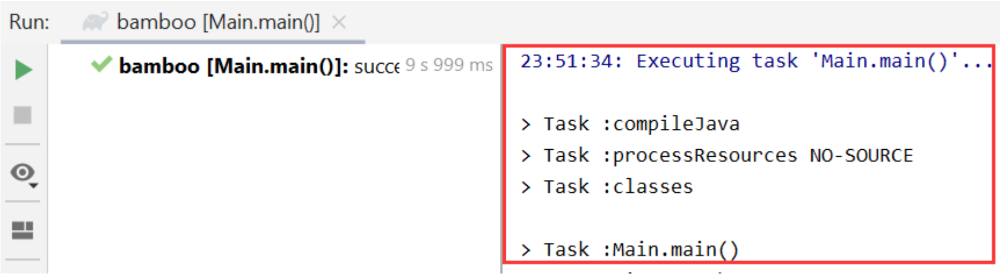
○ Create project from existing sources

● Import project from external model

That's it! You may wait a moment for importing Bamboo. After that, some Gradle-related files/folders will be generated in Bamboo directory, and you can ignore them.

Step 5

Since Bamboo is imported from Gradle model, IntelliJ IDEA always build and run it with Gradle, which makes it a bit slower and always output some annoying Gradle-related messages:



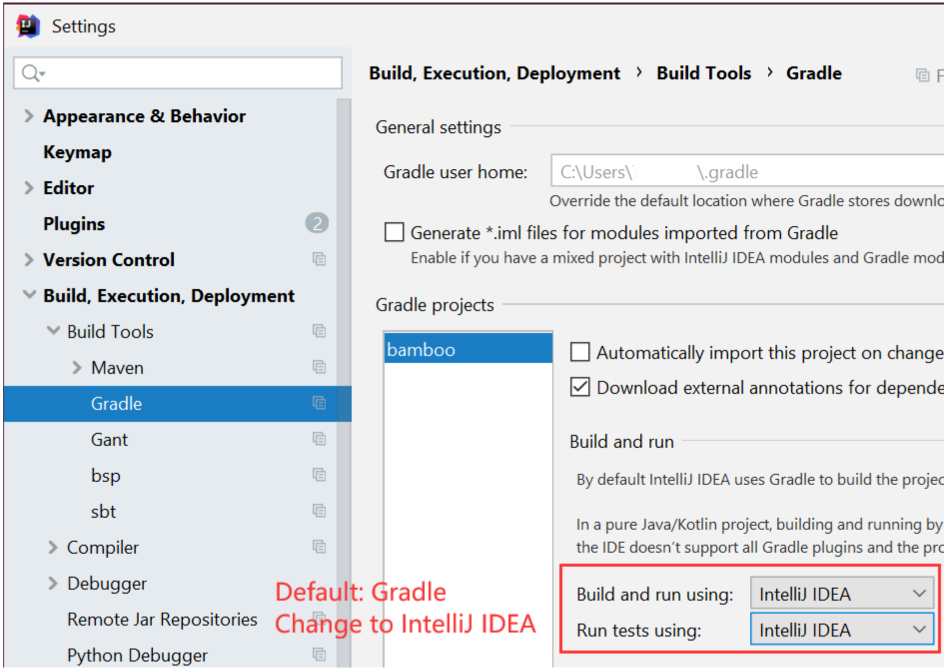
✓ **bamboo [Main.main()]:** succe 9 s 999 ms

```
23:51:34: Executing task 'Main.main()'...

> Task :compileJava
> Task :processResources NO-SOURCE
> Task :classes

> Task :Main.main()
```

Thus, we suggest you disable the Gradle in IntelliJ IDEA. Just go to File > Settings, and change the *build and run* tool from Gradle to IntelliJ IDEA as shown:



Settings

Build, Execution, Deployment › Build Tools › Gradle

General settings

Gradle user home: C:\Users\ \.gradle

Override the default location where Gradle stores downlo

□ Generate *.iml files for modules imported from Gradle

Enable if you have a mixed project with IntelliJ IDEA modules and Gradle mod

Gradle projects

bamboo

□ Automatically import this project on change

☑ Download external annotations for depende

Build and run

By default IntelliJ IDEA uses Gradle to build the projec

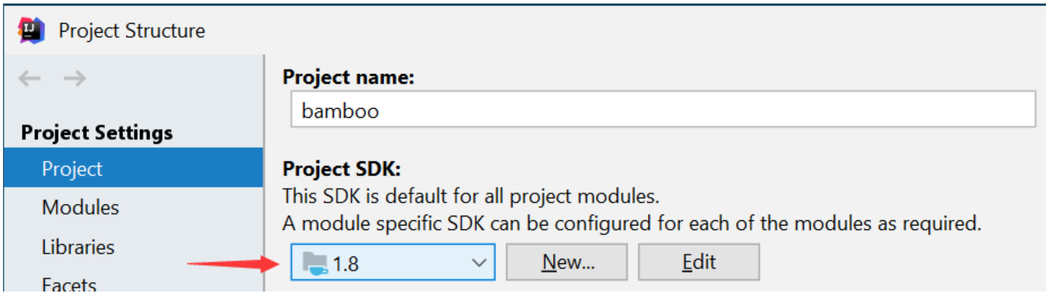
In a pure Java/Kotlin project, building and running by the IDE doesn't support all Gradle plugins and the pro

Build and run using: IntelliJ IDEA

Run tests using: IntelliJ IDEA

Default: Gradle
Change to IntelliJ IDEA

Notice: If your system has multiple JDKs, make sure that IntelliJ IDEA uses Java 8 (otherwise you may experience `NullPointerException` thrown by Soot). To configure this, go to File > Project Sturcture…, and select **1.8** for Project SDK:



Project name: bamboo

Project SDK:
This SDK is default for all project modules.
A module specific SDK can be configured for each of the modules as required.

1.8

Alternatively, if you (really :-)) want to build Bamboo from command line, you could change working directory to Bamboo folder, and build it with Gradle:

```
$ gradle compileJava
```

3 Implementation of Class Hierarchy Analysis

This Section introduces the necessary knowledge about Bamboo, Soot, and your task for this assignment.

3.1 Scope

In this assignment, we deal with four kinds of method invocations in Java, i.e., `invokestatic`, `invokespecial`, `invokeinterface`, and `invokevirtual`.

3.2 Soot Classes You Need to Know

Bamboo uses Jimple IR from Soot, and this section introduces several Soot classes you need to know for finishing this assignment. If you are interested in more details of Soot, you could check its online documentation (https://github.com/Sable/soot/wiki).

- ➢ `soot.SootClass`
  Soot representation of a Java class. Each instance of this class corresponds to a class in the Java program and contains various information about the class.
  - ◆ `soot.SootClass getSuperclassUnsafe()`: returns the superclass of this class. If this class is the top of class hierarchy, i.e., `java.lang.Object`, null is returned.
  - ◆ `soot.SootMethod getMethodUnsafe(String)`: returns the method of this class with the given subsignature (represented by string). If no method with the given subsignature can be found, null is returned. The subsignature of a method is its method name plus its descriptor, as shown in page 24 of Lecture 7. For example, the subsignature of the following method `foo` is "`T foo(P,Q,R)`":

```
class C {
    T foo(P p, Q q, R r) { … }
}
```

- `boolean isInterface()`: returns if this class is an interface.

- `soot.SootMethod`
  Soot representation of a Java method. Each instance of this class corresponds to a method in the Java program and contains various information about the method.
  - `soot.SootClass getDeclaringClass()`: returns the class which declares the this method.
  - `String getSubSignature()`: returns the string representation of subsignature of this method.
  - `boolean isConcrete()`: returns if this method is a concrete method which has method body. For the methods declared in interfaces and the abstract methods declared in abstract classes, this method returns false.

- `soot.FastHierarchy`
  This class provides class hierarchy information.
  - `Set<SootClass> getAllImplementersOfInterface(SootClass)`: For a given interface (the given `SootClass` MUST be an interface), returns set of all implementers of it but NOT their subclasses. For example, for the following program:

    ```
    interface I {…}
    interface II extends I {…}
    class A implements I {…}
    class B implements II {…}
    class C extends B {…}
    ```

    Suppose `i` is an instance of SootClass representing interface `I`, then `hierarchy.getAllImplementorsOfInterface(i)` returns classes `A` and `B`, but not `C`.
  - `Collection<SootClass> getSubclassesOf(SootClass)`: returns the direct subclasses of a given class.

3.3 Bamboo Classes You Need to Know

To implement class hierarchy analysis in Bamboo, you need to know the following classes.

- `bamboo.callgraph.CallKind`
  This enumeration type represents the kind of call graph edges. It defines four constants, `INTERFACE`, `VIRTUAL`, `SPECIAL` and `STATIC`, corresponding to the four kinds of invocations in Java.

- `bamboo.callgraph.JimpleCallGraph`
  This class represents a call graph. Its call sites are `Units` and methods are `SootMethods`.
  - `Collection<SootMethod> getEntryMethods()`: returns the entry methods of the program being analyzed. In this assignment, this API returns the main method.
  - `Collection<Unit> getCallSitesIn(SootMethod)`: returns all call sites in a given method.
  - `boolean contains(SootMethod)`: returns if a given method is reachable in this call graph. The entry method is reachable from the beginning.
  - `boolean addEdge(Unit,SootMethod,CallKind)`: adds a call edge to this call graph. It has three parameters, where `Unit` is the call site of the edge, `SootMethod` is the call target (i.e., callee), and `CallKind` is kind of this call. You could use `getCallKind(Unit)` to obtain the `CallKind` of a given call site. Once a call edge from a call site *c* to a method *m* is added to this call graph, the method *m* becomes reachable.

- `bamboo.callgraph.cha.CHACallGraphBuilder`
  This class implements class hierarchy analysis and call graph construction. It is incomplete, and you need to finish it as explained in Section 3.4.

- `bamboo.callgraph.cha.Main`
  This is the main class of class hierarchy analysis, which performs the analysis for input Java program. We introduce how to run this class in Section 3.5.

3.4 Your Task [Important!]

In this assignment, you need to finish class `CHACallGraphBuilder`, which implements class hierarchy analysis and uses it to build call graph. Specifically, you will finish the following three APIs:

- `SootMethod dispatch(SootClass,SootMethod)`

  This method implements the Dispatch function given in page 26 of the slides for Lecture 7. If no satisfying method is found, returns null.

- `Set<SootMethod> resolveCalleesOf(Unit)`

  This method implements the Resolve function given in page 33 of the slides for Lecture 7.

  - Hint: You only need to handle interface and virtual calls. The method at the call site is stored in variable `method`.

- `void buildCallGraph(JimpleCallGraph)`

  This method implements the BuildCallGraph algorithm given in page 52 of the slides for Lecture 7.

  - Hint: You should complete this method by adding call edges to the call graph. As introduced in Section 3.3, you can obtain the entry method from `JimpleCallGraph`. Besides, `JimpleCallGraph.addEdge()` method will automatically mark the target method of given call edge as reachable method, so you do not need to maintain reachable methods by yourself.

We have provided code skeletons for the above three APIs, and your task is to fill the part with comment "`TODO – finish me`".

3.5 Run Class Hierarchy Analysis as an Application

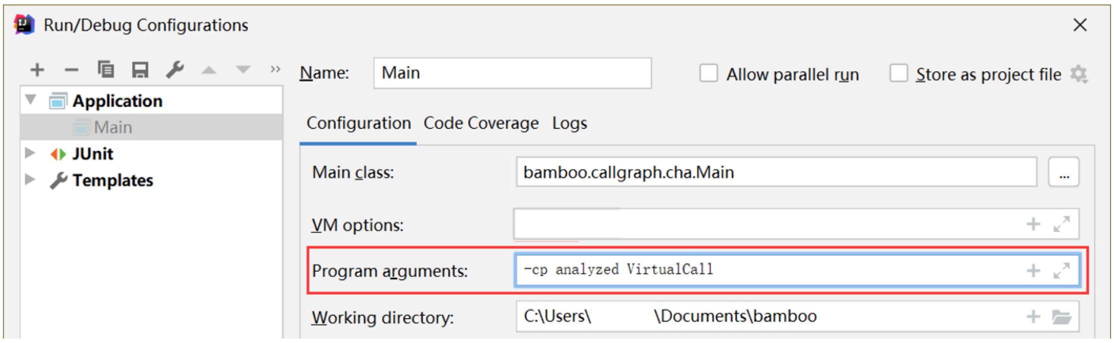
As mentioned in Section 3.4, the main class of class hierarchy analysis is

```
bamboo.callgraph.cha.Main
```

The format of its arguments is:

```
-cp <CLASS_PATH> <CLASS_NAME>
```

`<CLASS_PATH>` is the class path, and `<CLASS_NAME>` is the name of the input class to be analyzed. Bamboo locates input class from given class path. For example, to analyze the `VirtualCall.java` in class path `analyzed/`, just configure program arguments in IntelliJ IDEA as follows:



Name: Main

□ Allow parallel run

□ Store as project file

Main class: bamboo.callgraph.cha.Main

VM options:

Program arguments: -cp analyzed VirtualCall

Working directory: C:\Users\ \Documents\bamboo

For each input program, Bamboo performs class hierarchy analysis to build call graph, and outputs the callees of every call site in each reachable method (if the method contains any call sites). You can use this information to help develop and debug. We encourage you to write some Java classes and analyze them.

Of course, you could also run the analysis using Gradle, with the following command:

```
$ gradle run --args="-cp <CLASS_PATH> <CLASS_NAME>"
```

3.6 Test Class Hierarchy Analysis with JUnit

To make testing convenient, we have prepared some Java classes as test inputs in folder `analyzed/`. Every class has an associated file named `*-expected.txt`, which contains the expected results of class hierarchy analysis (in form of call sites and their callees) for each reachable method. You could analyze these test inputs by running test class (powered by JUnit):

```
bamboo.callgraph.cha.CHATest
```

This test class analyzes all provided Java classes in `analyzed/`, and compares the given analysis results to the expected results. If your implementation of class hierarchy analysis is correct, the tests will pass, otherwise it fails and outputs the differences between expected and given results.

Again, you could run tests with Gradle, just type:

```
$ gradle clean test
```

This command will delete the build directory, rebuild Bamboo, and run tests.

3.7 Run Interprocedural Constant Propagation

As mentioned in Sections 1 and 2, to demonstrate the usefulness of call graph construction and interprocedural analysis, this assignment contains an interprocedural constant propagation, which uses your implementation of class hierarchy analysis to build call graph, interprocedural control-flow graph (ICFG), and performs constant propagation on the ICFG. The main class of the analysis is:

```
bamboo.dataflow.analysis.constprop.IPMain
```

The format of its arguments is:

```
-cp <CLASS_PATH> <CLASS_NAME>
```

We also provide a test case `CHACP.java` in `analyzed/`, which comes from Lecture 7. After you finish the class hierarchy analysis, we recommend you to run both interprocedural and intraprocedural constant propagation for the test case, to observe their analysis results and precision differences.

Note that before you run intra- and interprocedural constant propagation, please replace `bamboo.dataflow.analysis.constprop.ConstantPropagation.java` in this Assignment package by your implementation for Assignment 2.

4 General Requirements

- In this assignment, your only goal is correctness. Efficiency is not your concern.
- DO NOT distribute the assignment package to any others.

- Last but not least, do not plagiarize. The work must be all your own!

5 Submission of Assignment

Your submission should be a zip file, which contains your implementation of `CHACallGraphBuilder.java`
The naming convention is of the zip file is:

```
<STUDENT_ID>-<NAME>-A3.zip
```

Please submit your assignment through 教学立方.

6 Grading

The points will be allocated for correctness. We will use your submission to analyze the given test files from the `analyzed/` directory, as well as other tests of our own, and compare your output to that of our solution.

Good luck!