Programming Assignment 4: Pointer Analysis

Course "Static Program Analysis" @Nanjing University
Assignments Designed by Tian Tan and Yue Li
Due: 23:00, Thursday, December 17, 2020

1 Goal

In this programming assignment, you will implement whole-program context-insensitive pointer analysis for Java based on Bamboo. The pointer analysis builds a call graph on the fly. To show the usefulness of pointer analysis, we provide an interprocedural constant propagation, which uses the call graph constructed by your pointer analysis. If your implementation is correct, you can observe that pointer analysis can build a more precise call graph than class hierarchy analysis (CHA). Consequently, interprocedural constant propagation based on pointer analysis achieves better precision than CHA (please see Section 3.6 for more details). Again, 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 pointer analysis. In addition, we also include an interprocedural data-flow analysis framework and an interprocedural constant propagation to demonstrate the usefulness of pointer analysis.

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)

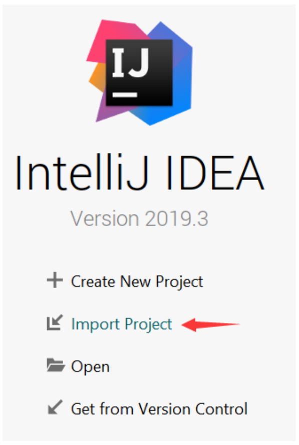
Bamboo is written in Java, so it is cross-platform. To build and run Bamboo, you need

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.

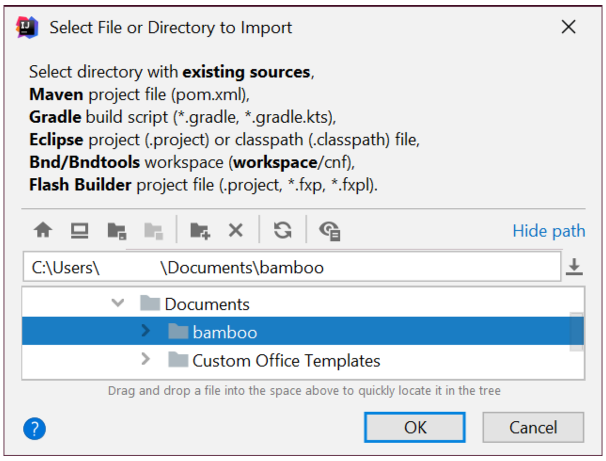
Step 1
Download IntelliJ IDEA from JetBrains (http://www.jetbrains.com/idea/download/)

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.)

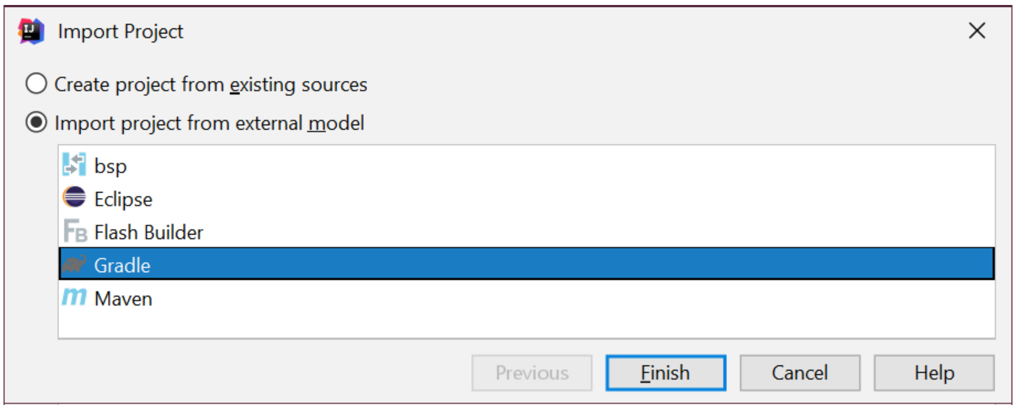
Step 3
Select the `bamboo/` directory, then click "OK".


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).

Hide path
C:\Users\ \Documents\bamboo
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

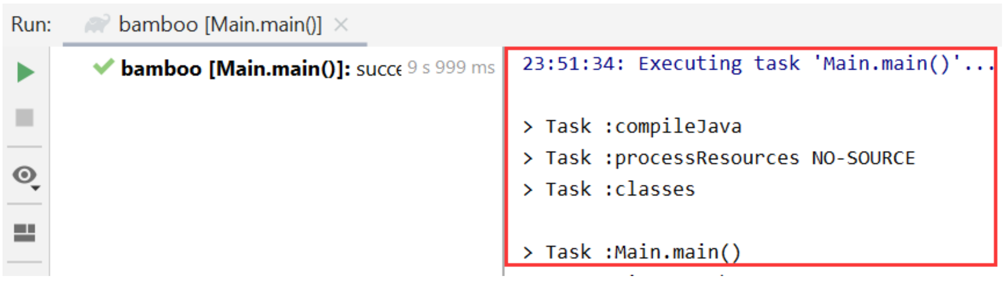
Create project from existing sources

[✓ Import project from external model]

bsp
Eclipse
Flash Builder
[✓ Gradle]
Maven

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:


Run: bamboo [Main.main()]

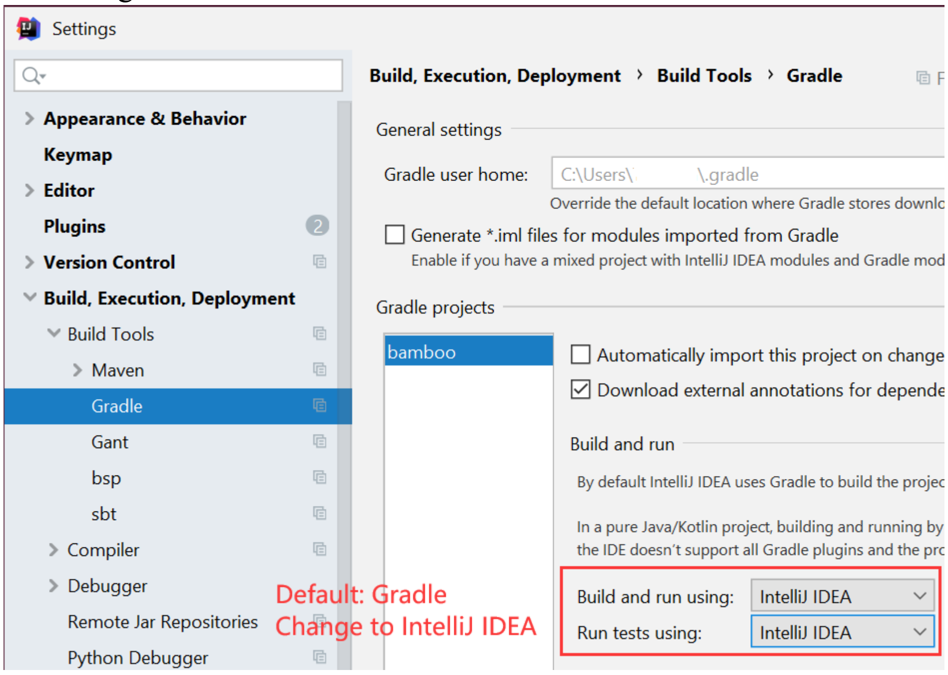
bamboo [Main.main()]: succe 9 s 999 ms

```
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

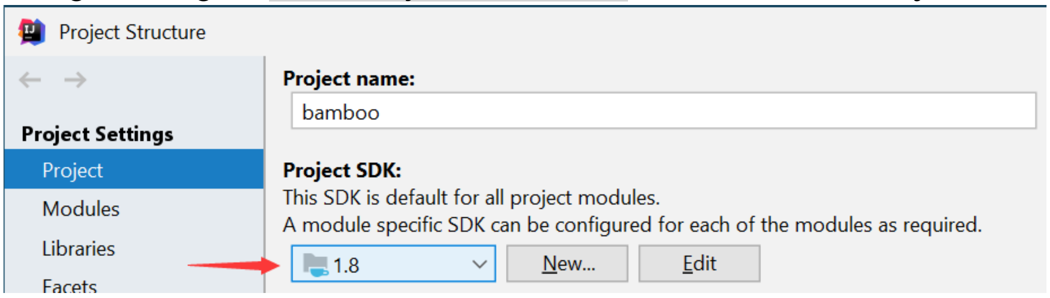
Build and run

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

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 Structure

Project Settings
- Project
- Modules
- Libraries
- Facets

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.

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 Pointer Analysis

This Section introduces the necessary knowledge about Bamboo and your task for this assignment. Note that Soot's Jimple IR is sophisticated and contains rich information, however, many of them are irrelevant to pointer analysis, and it is not that convenient to extract pointer-relevant information. To ease the implementation of pointer analysis, we have designed and implemented a new pointer analysis IR in Bamboo, which provides convenient APIs to obtain pointer-relevant information and excludes unnecessary details about the program statements. Our pointer analysis IR provides all information you need to implement pointer analysis, so in this assignment, you do not need to touch any Soot classes.

3.1 Scope

The scope of this assignment is the same as explained in Lecture 8. We deal with two kinds of pointer in Java (local variable and instance field), and five pointer-affecting statements, i.e., new, assign, store, load and call. Our pointer analysis handles not only virtual calls but also special and static calls.

3.2 Bamboo Classes You Need to Know

To implement pointer analysis in Bamboo, you need to know the following classes. We start with the classes for pointer analysis IR in Bamboo.

- `bamboo.pta.element.Variable`
  This class represents local variables. It provides some convenient APIs to obtain relevant statements of a variable, as explained below.
  - `Set<InstanceStore> getStores()`: returns the store statements (we will introduce `InstanceStore` later) whose base variable is this variable.

- `Set<InstanceLoad> getLoads()`: returns the load statements (we will introduce `InstanceLoad` later) whose base variable is this variable.

For example, suppose we are analyzing the following code snippet:

```
1 x = y;
2 x.h = a;
3 a.h = z;
4 a = x.f;
5 b = y.f;
6 c = x.g;
```

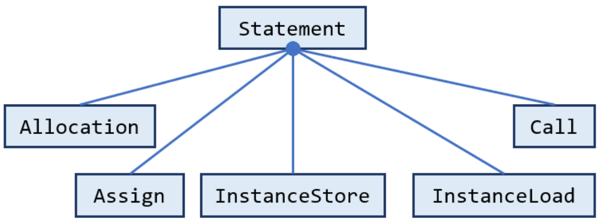
If `var` represents variable `x`, then `var.getStores()` returns the store statements at line 2, and `var.getLoads()` returns the load statements at lines 4 and 6.

- `bamboo.pta.element.Field`
  This class represents fields.

- `bamboo.pta.element.Obj`
  This class represents abstract objects.

- `bamboo.pta.element.CallSite`
  This class represents call sites.

- `bamboo.pta.element.Method`
  This class represents methods.
  - `Set<Statement> getStatements()`: returns the pointer-affecting statements in this method. `Statement` has five subclasses, corresponding to the five pointer-affecting statements.



  Below we introduce these classes.

- `bamboo.pta.statement.Allocation`
  This class represents allocation statements, e.g., `x = new T()`.
  - `Variable getVar()`: returns the LHS variable (`x`) of the allocation site.
  - `Object getAllocationSite()`: returns the identifier of the allocation site.

- `bamboo.pta.statement.Assign`
  This class represents assign statements, e.g., `x = y`.

- `Variable getTo()`: returns the LHS variable (`x`) of the assignment.
- `Variable getFrom()`: returns the RHS variable (`y`) of the assignment.

➢ `bamboo.pta.statement.InstanceStore`
This class represents assign statements, e.g., `x.f = y`.
- `Variable getBase()`: returns the base variable (`x`) of the store.
- `Field getField()`: returns the field (`f`) of the store.
- `Variable getFrom()`: returns the RHS variable (`y`) of the store.

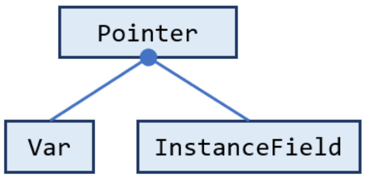
➢ `bamboo.pta.statement.InstanceLoad`
This class represents assign statements, e.g., `y = x.f`.
- `Variable getTo()`: returns the LHS variable (`y`) of the load.
- `Variable getBase()`: returns the base variable (`x`) of the load.
- `Field getField()`: returns the field (`f`) of the load.

➢ `bamboo.pta.statement.Call`
This class represents method calls. Since implementation of method call is not your concern in this assignment, we do not introduce this class in detailed.

Now we introduce the classes for pointer flow graph (PFG).

➢ `bamboo.pta.analysis.ci.Pointer`
This class represents the pointers in the analysis, i.e., the nodes in the PFG. Each pointer corresponds to a variable or an instance field in the program and is associated with a points-to set.
- `PointsToSet getPointsToSet()`: returns the points-to set of this pointer. Each pointer is automatically associated with an empty set on creation, so this method always returns a set (not `null`).

This class has two subclasses, as introduced below.



➢ `bamboo.pta.analysis.ci.Var`
This class represents variable nodes in the PFG, and each of its instance corresponds to a variable.
- `Variable getVariable()`: returns the corresponding variable of this node.

- `bamboo.pta.analysis.ci.InstanceField`
  This class represents instance field nodes in the PFG, and each of its instance corresponds to an instance field (e.g., $o_i.f$).
  - `Obj getBase()`: returns the corresponding base object of this node.
  - `Field getField()`: returns the corresponding field of this node.

- `bamboo.pta.analysis.ci.PointsToSet`
  This class represents points-to sets, i.e., sets of `Obj` in pointer analysis.
  - `boolean addObject(Obj)`: adds an object to this points-to set, if the given object is already in the set, then returns false, otherwise, returns true.
  - `boolean isEmpty()`: returns if this points-to set is empty.
  - `Iterator<Obj> iterator()`: returns an iterator over this points-to set for iterating its objects. Also, this method means that `PointsToSet` is iterable, i.e., you can iterate the objects in a points-to set in this way:
    ```
    PointsToSet pts = …
    for (Obj obj : pts) { … }
    ```

- `bamboo.pta.analysis.ci.PointerFlowGraph`
  This class represents a pointer flow graph of the program. It also maintains the mapping from variables/instance fields to corresponding pointers (PFG nodes).
  - `Var getVar(Variable)`: returns the corresponding variable node of the given variable (this method also adds the returned pointer to the PFG).
  - `InstanceField getInstanceField(Obj,Field)`: returns the corresponding instance field node of the given object and field (this method also adds the returned pointer to the PFG).
  - `boolean addEdge(Pointer s,Pointer t)`: adds an edge $s \rightarrow t$ to this PFG. If the edge is already in the PFG, returns false, otherwise, returns true.
  - `Set<Pointer> getSuccessorsOf(Pointer)`: returns the successors of given pointer (PFG node) on the PFG.

Now we introduce the classes for pointer analysis algorithm.

- `bamboo.pta.analysis.ci.WorkList`
  This class represents the worklist in pointer analysis algorithm.
  - `void addPointerEntry(Pointer,PointsToSet)`: adds a worklist entry, i.e., a pair of a pointer and a points-to set (whose objects should be propagated to the points-to set of the pointer) to the worklist.

- `bamboo.pta.analysis.ci.PointerAnalysis`
  This class implements the pointer analysis algorithms (i.e., the algorithm in page

of 115 for Lecture 10, please see the slides on the course website). It is incomplete, and you need to finish it as explained in Section 3.3.

- `void solve()`: starts the pointer analysis algorithm.
- `void initialize()`: implements the first two lines of the pointer analysis algorithm, i.e., initializes various data structures and analyzes entry methods.
- `void analyze()`: implements the big while-loop in the pointer analysis algorithm, which processes worklist entries until it is empty.

➢ `bamboo.pta.analysis.ci.Main`
This is the main class of pointer analysis, which performs the analysis for input Java program. We introduce how to run this class in Section 3.4.

3.3 Your Task [Important!]

In this assignment, you need to finish class `PointerAnalysis`, which implements pointer analysis algorithm. For simplicity, we have included all code for handling method calls, so you only need to implement the logic for handling new, assign, store, and load statements in Java. Specifically, you will finish the following six methods:

- `PointsToSet propagate(Pointer,PointsToSet)`

  This method implements difference set computation ($\Delta = pts - pt(n)$) and the Propagate function given in page 43 of the slides for Lecture 9, e.g., `propagate(p,pts)` propagates pts into $pt(p)$, and returns $pts - pt(p)$. We merge these two steps into one method for reducing redundant computation.

  ❖ Hint: You could uncomment the `println()` invocation in this method to observe how points-to sets are propagated to the pointers. This could help you debug and better understand pointer analysis algorithm.

- `void addPFGEdge(Pointer,Pointer)`

  This method implements the AddEdge function given in page 43 of the slides for Lecture 9.

- `void processAllocations(Method)`

  This method processes allocations (i.e., new statements) in a new reachable method, which corresponds to the first **foreach** loop in the AddReachable function given in page 118 of the slides for Lecture 10.

  ❖ Hint: You need heap model (heap abstraction) to obtain abstract object. We apply allocation-site abstraction, so each allocation site produces one abstract object. You can use `heapModel` (a field of `PointerAnalysis`) to do this. E.g., suppose `alloc` is an instance of `Allocation`, then you can obtain its corresponding abstract object in this way:

```
Object allocSite = alloc.getAllocationSite();
Obj obj = heapModel.getObj(allocSite, alloc.getType(), method);
```

where `method` is the given method.

- `void processLocalAssign(Method)`

  This method processes local assignments (e.g., `x = y`) in a new reachable method, which corresponds to the second **foreach** loop in the AddReachable function given in page 118 of the slides for Lecture 10.

- `void processInstanceStore(Var,PointsToSet)`

  This method processes instance stores (e.g., `x.f = y`), which corresponds to the **foreach** loop for handling store statements in the Solve function given in page 124 of the slides for Lecture 10. The first parameter (`Var`) is the variable node whose points-to set changes (i.e., `x`), and the second parameter (`PointsToSet`) is the changed part (i.e., Δ in the algorithm).

- `void processInstanceLoad(Var,PointsToSet)`

  This method processes instance loads (e.g., `y = x.f`), which corresponds to the **foreach** loop for handling load statements in the Solve function given in page 124 of the slides for Lecture 10. The first parameter (`Var`) is the variable node whose points-to set changes (i.e., `x`), and the second parameter (`PointsToSet`) is the changed part (i.e., Δ in the algorithm).

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

3.4 Run Pointer Analysis as an Application

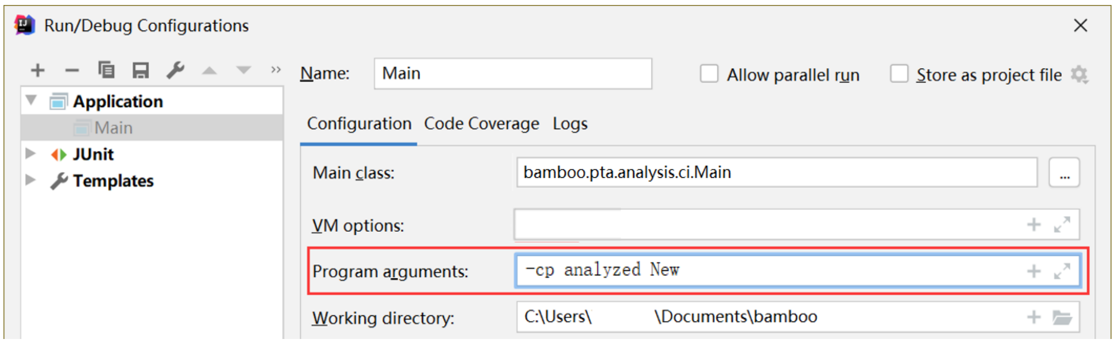
As mentioned in Section 3.2, the main class of pointer analysis is

```
bamboo.pta.analysis.ci.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 `New.java` in class path `analyzed/`, just configure program arguments in IntelliJ IDEA as follows:


Run/Debug Configurations

Name: Main

☐ Allow parallel run ☐ Store as project file

Application
Main
JUnit
Templates

Configuration Code Coverage Logs

Main class: bamboo.pta.analysis.ci.Main

VM options:

Program arguments: -cp analyzed New

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

where `<method>` is the method containing the allocation site of the object, `<Type>` is type of the object, and `<line>` is the line number of the allocation site. 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>”
```

3.5 Test Pointer 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 pointer analysis, i.e., the points-to sets of all variables and instance fields. You could analyze these test inputs by running test class (powered by JUnit):

```
bamboo.pta.PTATest
```

This test class analyzes all provided Java classes in `analyzed/`, and compares the given analysis results to the expected results. If your implementation of pointer 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.

3.6 Run Interprocedural Constant Propagation

As mentioned in Sections 1 and 2, to demonstrate the usefulness of pointer analysis (it can build more precise call graph than CHA), this assignment contains an interprocedural constant propagation, which uses your implementation of pointer 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.PTAMain
```

The format of its arguments is:

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

We also provide a test case `PTACP.java` in `analyzed/`, which comes from Lecture 8. After you finish the pointer analysis, we recommend you to run both pointer analysis based and CHA based 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 `PointerAnalysis.java`
The naming convention is of the zip file is:

```
<STUDENT_ID>-<NAME>-A4.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!