Programming Assignment 1: Constant Propagation

Course “Static Program Analysis” @Nanjing University
Assignments Designed by Tian Tan and Yue Li
Due: 23:00, Sunday, April 19, 2020

1 Goal

In the first programming assignment, you will implement a constant propagation analysis for Java. The assignment is based on Bamboo, a static program analysis framework designed for the assignments of course “Static Program Analysis” (software analysis) at Nanjing University. Bamboo includes a data-flow analysis framework and it means that you do not need to concern about how the data-flow problem is solved, and you could focus on implementing **meet** and **transfer** functions for constant propagation. As Bamboo is only for the education purpose (for students to get familiar with different static analysis approaches, rather than aiming to produce a practical analysis tool), for simplicity, Bamboo does not support full Java features. As a result, you only need to consider a small subset of Java features when you implement the assignments in this course.

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 only include the data-flow analysis framework of Bamboo, and the necessary classes for constant propagation. 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

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-jdk8-downloads.html

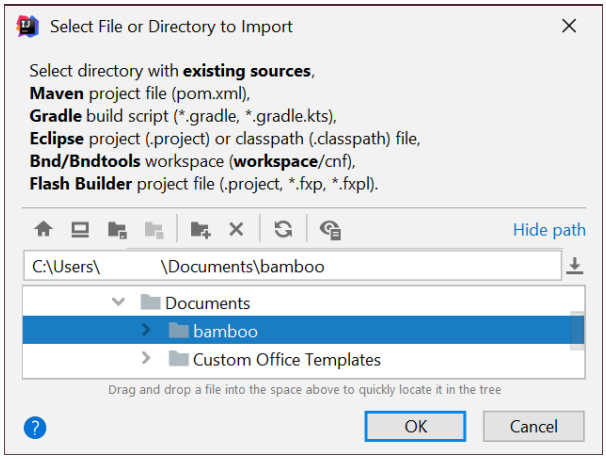
We highly recommend you to finish this (and the following) assignments with IntelliJ IDEA. Given the Gradle build script, it is very easy to import Bamboo to IntelliJ IDEA, as follows.

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

Step 2
Start to import a project

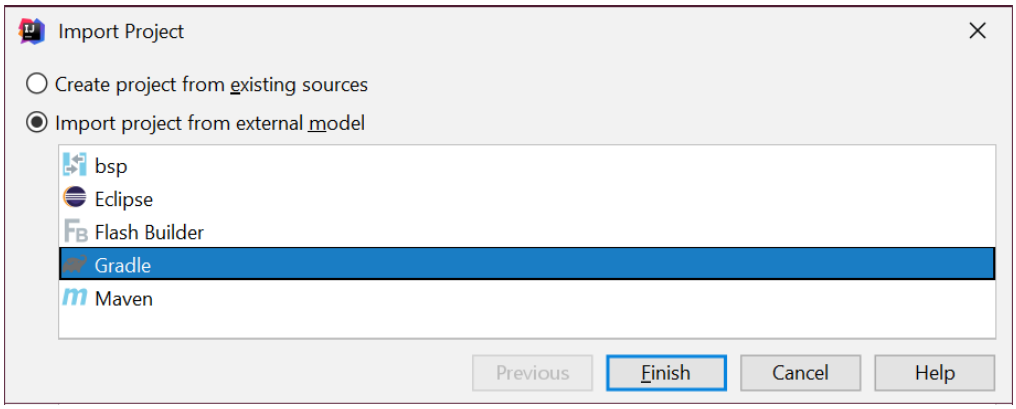
(Note: if you have already used IntelliJ IDEA, and opened some projects, then you could choose File > New > Project from Existing Sources… to open the same dialog for the next step.)

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



Step 4

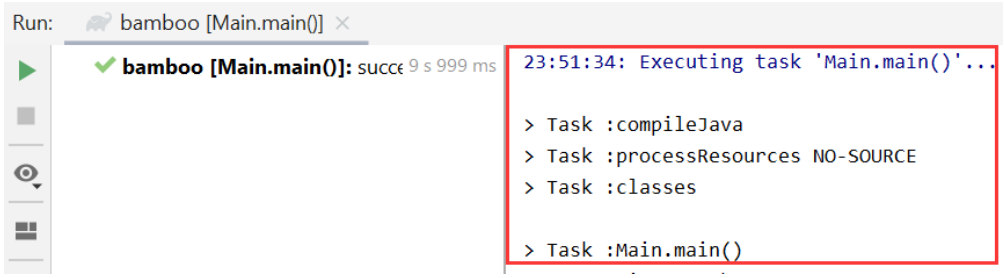
Import project from external model Gradle, then click “Finish”.



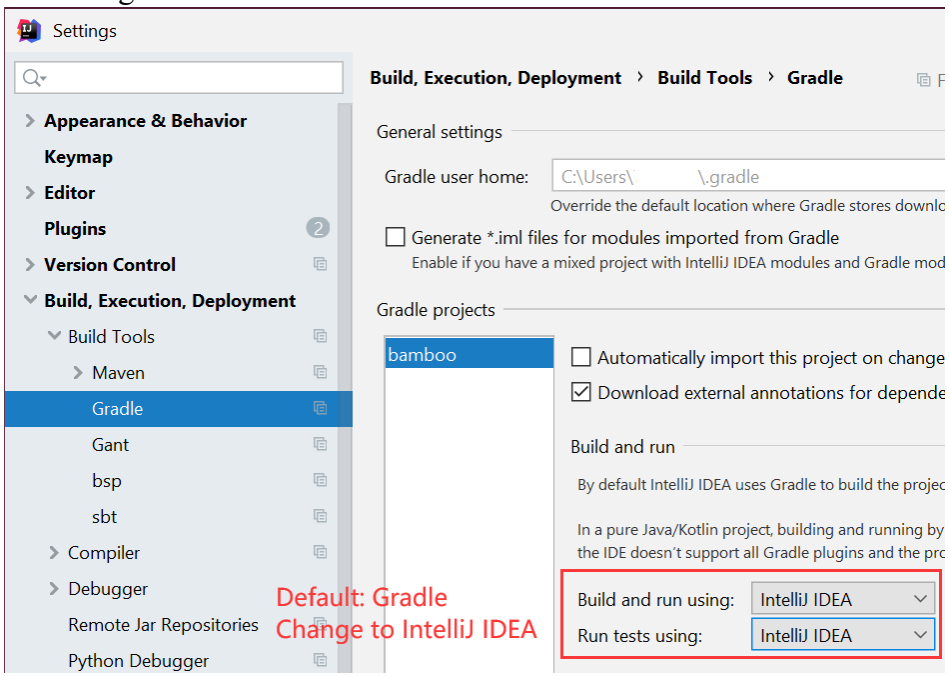
That’s it! You may wait a moment for importing Bamboo. After that, some Gradle-related files/folders will be generated in Bamboo directory, and you can ignore them.

Step 5

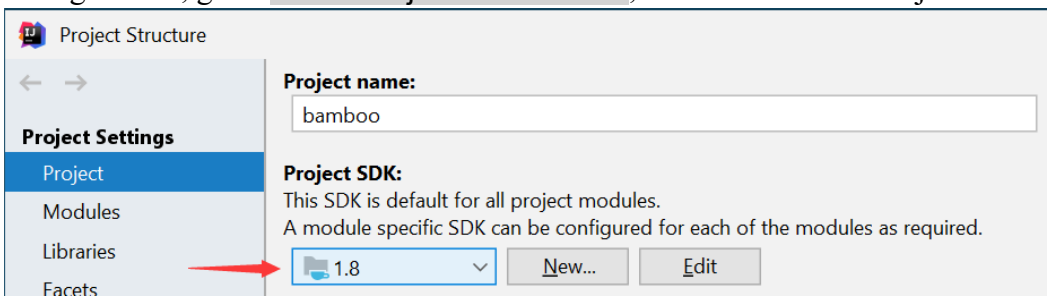
Since Bamboo is imported from Gradle model, IntelliJ IDEA always build and run it with Gradle, which makes it a bit slower and always output some annoying Gradle-related messages:



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:



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:



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 Constant Propagation

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

3.1 Scope

In this assignment, we deal with a subset of Java features. We only consider `int` type. Other primitive types and reference types (class types, array types, etc.) are out of scope. For `int`, only four arithmetic operators, i.e., `+`, `-`, `*`, `/`, are concerned.

For statements, as shown in page 241 of slides for Lecture 6, you only need to focus on assignment statements whose right-hand side expression are:

- Constant, e.g., `x = 1`
- Variable, e.g., `x = y`
- Binary expression, e.g., `x = a + b`, `x = 1 - c`, or `x = 2 * 3`

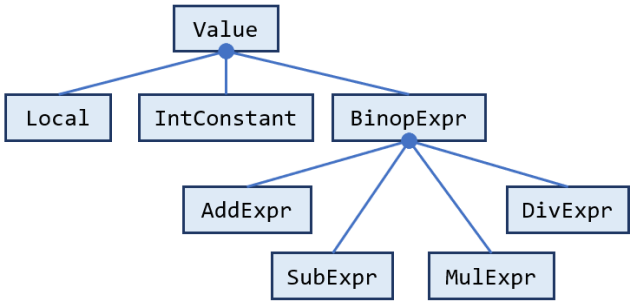
For other statements, the transfer function is identity function. You could check the Java programs we provided in `analyzed/`, to get concrete feeling about the scope.

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.Value`

  Soot provides this interface to represent the data in the program, which has many subclasses. In this assignment, you only need to operate a few kinds of `Value`s, as shown in this partial class hierarchy:



  Now we introduce the classes in this figure.

- `soot.Local`

  This class represents local variables in Jimple IR. Some local variables correspond to the ones in Java programs, and the others are temporary variables generated during IR construction.

- `soot.jimple.IntConstant`

  Each instance of this class represents an integer constant in the program, and you can obtain the constant value from its field named `value`. For example:

```
IntConstant c = …;
int n = c.value; // now n holds the constant value
```

- `soot.jimple.BinopExpr`

  This class represents binary expressions in the program. It has many subclasses, but you only need to handle four of them: `AddExpr` (`+`), `SubExpr` (`-`), `MulExpr` (`*`), and `DivExpr` (`/`). It provides two APIs to obtain the two operands:

  - `soot.Value getOp1()`
  - `soot.Value getOp2()`

  You could use `instanceof` to determine what kind a binary expression is. For example:

```
BinopExpr e = …;
Value op1 = e.getOp1();
if (e instanceof AddExpr) { // e is an add expression
    …
}
```

- `soot.jimple.DefinitionStmt`

  This class represents all definition statements (e.g., `x = y` or `x = a + b` where `x` is defined). It provides two APIs to obtain its LHS and RHS values:

- `soot.Value getLeftOp()`
- `soot.Value getRightOp()`

In this assignment, an LHS value must be local variable (as array and field accesses are out of scope), so you could use `DefinitionStmt` as follows:

```
DefinitionStmt s = ...;
Local l = (Local) s.getLeftOp();
```

3.3 Bamboo Classes You Need to Know

To implement constant propagation in Bamboo, you need to know the following classes.

- `bamboo.dataflow.analysis.DataFlowAnalysis`
  This is the interface between concrete data-flow analyses (e.g., constant propagation and reaching definition) and data-flow analysis solver in Bamboo. It has 5 APIs, corresponding to five key elements in a data-flow analysis (page 301 of slides for Lecture 4), i.e., direction, boundary, initialization, transfer function, and meet.

  The APIs in this class are invoked by data-flow analysis framework of Bamboo. Note that the transfer function manipulates statements, not basic blocks, which makes it simpler.

- `bamboo.dataflow.analysis.constprop.Value`
  This class represents the lattice values in constant propagation, i.e., (the lattice described in page 232 of slides for Lecture 6). Its code and comments are self-explaining. Note that to create the instance of this class, you will use these three static methods:

  - `Value getNAC()`: returns NAC
  - `Value getUndef()`: returns UNDEF
  - `Value makeConstant(int)`: returns a constant for the given integer

  Here are some examples:

  ```
  Value nac = Value.getNAC();
  Value undef = Value.getUndef();
  Value three = Value.makeConstant(3);
  ```

- `bamboo.dataflow.analysis.constprop.FlowMap`
  This class represents the maps from variables (`soot.Local`) to lattice values (`Value`), i.e., the data flows IN[s] and OUT[s] of each statement (`soot.Unit`). You may use following APIs:

  - `Value get(soot.Local)`: returns the associated value of given local

variable. If the local is absent in this map, then it returns UNDEF.

- `Value put(soot.Local,Value)`: associates the specified value with the specified local variable in this map. If the map previously contained a mapping for the variable, the old value is replaced by the specified value.
- `boolean update(soot.Local,Value)`: updates the local variable-value mapping in this map, returns if the update changes the content of this map.
- `Set<Local> keySet()`: returns set of all local variables in this map.
- `boolean copyFrom(FlowMap)`: copies the content from given map to this map, returns if the copy operation changes the content of this map.

➢ `bamboo.dataflow.analysis.constprop.ConstantPropagation`
This class implements `DataFlowAnalysis`, and defines constant propagation analysis. It is incomplete, and you need to finish it as explained in Section 3.4.

➢ `bamboo.dataflow.analysis.constprop.Main`
This is the main class of constant propagation 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 implement the **transfer function** and **meet function** for constant propagation (in class `ConstantPropagation`). Specifically, you will finish the following four APIs:

- `FlowMap meet(FlowMap,FlowMap)`

  This is the **meet function** of constant propagation, which meets two `FlowMap`s and returns the resulting `FlowMap`. This function is used to handle control-flow influences.

- `Value meetValue(Value,Value)`

  This corresponds to the meet operator ⊓ defined in page 232 of slides for Lecture 6. You should invoke this function from `meet()`.

- `boolean transfer(Unit,FlowMap,FlowMap)`

  This is the **transfer function** F defined in page 241 of slides for Lecture 6.

- `Value computeValue(soot.Value,FlowMap)`

  This function computes for the lattice value (`Value`) for a right-hand side expression (`soot.Value`). You will need to handle three cases in this function as described in page 241 of slides for Lecture 6. You should invoke this function from `transfer()`.

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

3.5 Run Constant Propagation as an Application

As mentioned in Section 3.3, the main class of constant propagation is

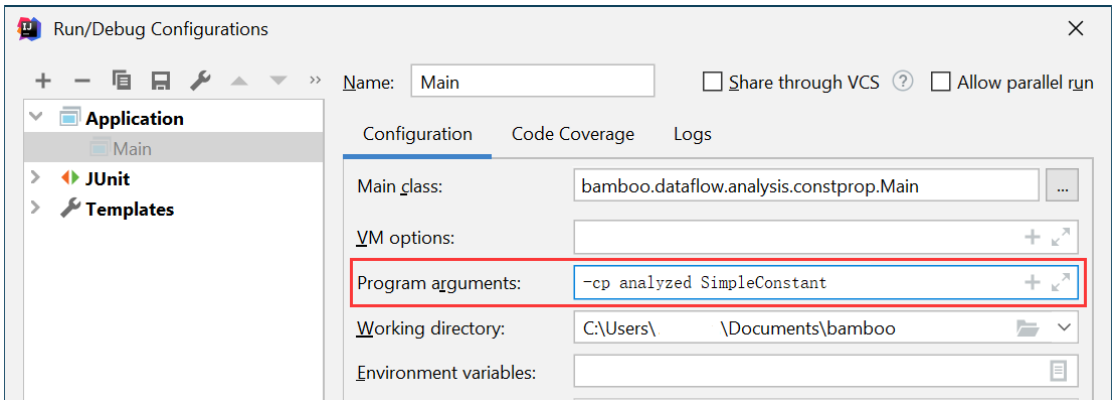
`bamboo.dataflow.analysis.constprop.Main`

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 `SimpleConstant.java` in class path `analyzed/`, just configure program arguments in IntelliJ IDEA as follows:



Bamboo performs constant propagation for each method of input class, and outputs the analysis results, i.e., the data-flow information in OUT flow of each Unit (Jimple IR). 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 Constant Propagation 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 partial expected results of constant propagation at some program points (line numbers). You could analyze these test inputs by running test class (powered by JUnit):

`bamboo.dataflow.analysis.constprop.CPTest`

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

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 `ConstantPropagation.java`, and its naming convention is:

```
<STUDENT_ID>-<NAME>-A1.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!