Programming Assignment 2: Dead Code Detection

Course "Static Program Analysis" @Nanjing University
Assignments Designed by Tian Tan and Yue Li
Due: 23:00, Sunday, May 10, 2020

1 Goal

In this programming assignment, you will implement a dead code detection for Java based on Bamboo. To implement dead code detection, you will also need to finish a constant propagation and a live variable analysis in this assignment. We will describe the details about these analyses in Section 3. 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 only include the data-flow analysis framework of Bamboo, and the necessary classes for dead code detection. 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 three files 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.

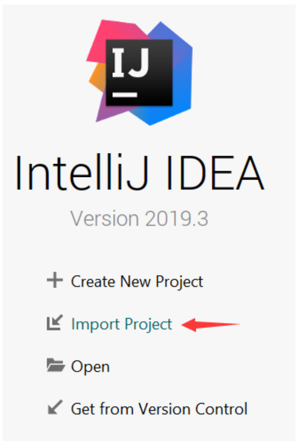
2.2 Setup Instructions (Same as Assignment 1)

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-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.

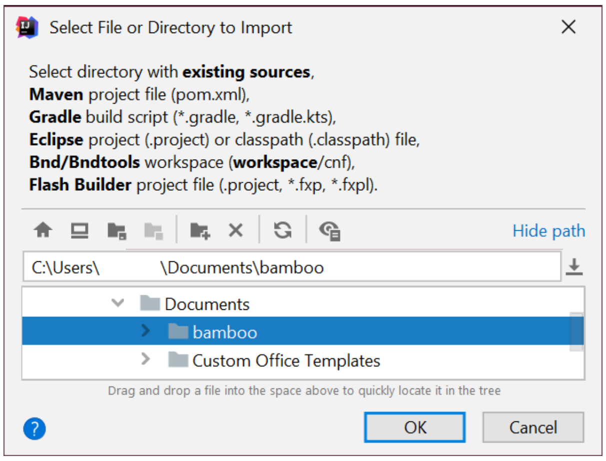
Step 1
Download IntelliJ IDEA from JetBrains (http://www.jetbrains.com/idea/download/)

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.)

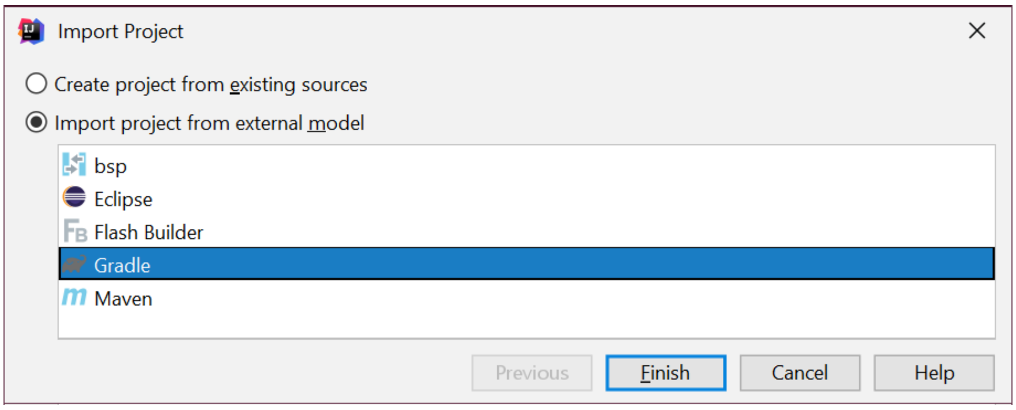
Step 3
Select the `bamboo/` directory, then click "OK".


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

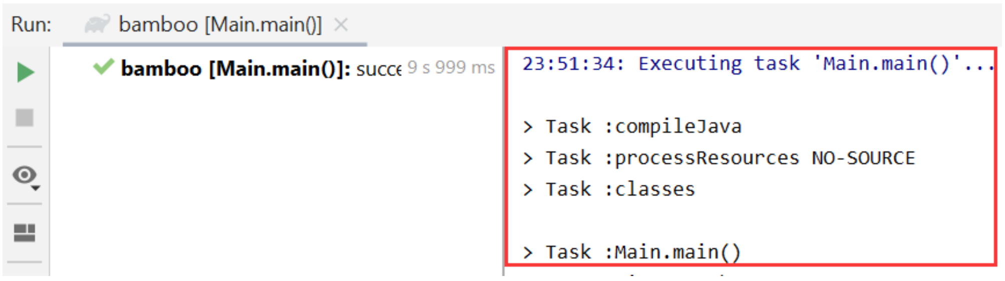
● Import project from external model

- 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:

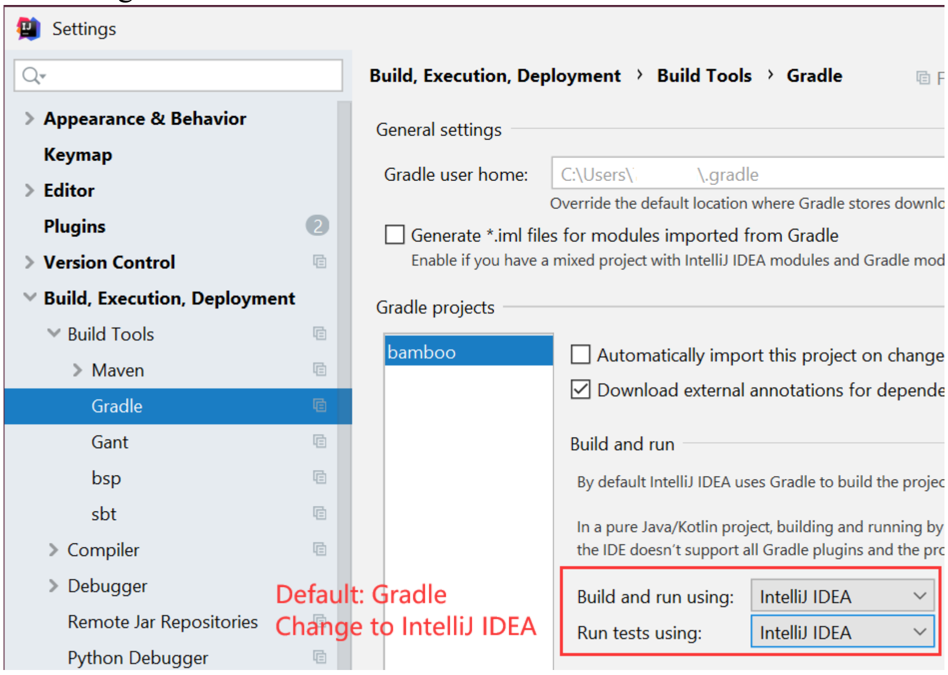

Run: bamboo [Main.main()]
bamboo [Main.main()]: succe 9 s 999 ms

```
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:



General settings

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

Override the default location where Gradle stores downlc

☐ 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

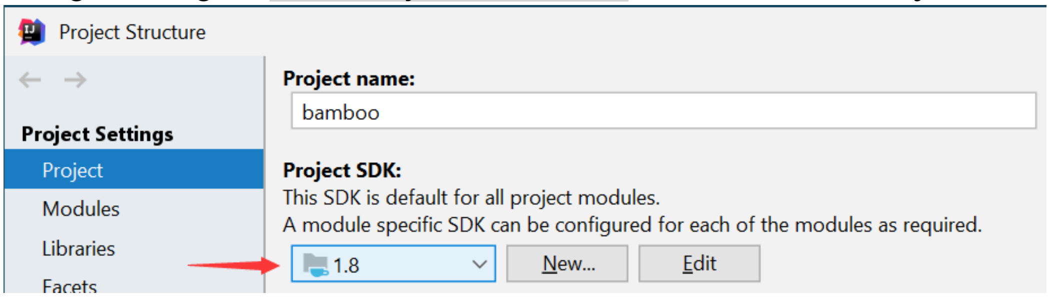
By default IntelliJ IDEA uses Gradle to build the projec

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

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 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 Dead Code Detection

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

3.1 Dead Code Detection

Dead code is the part of program which is unreachable (i.e., never be executed) or is executed but whose result is never used in any other computation. Eliminating dead code does not affect the program results, meanwhile, it can shrink program size and improve the performance. Thus, dead code elimination is a common compiler optimization. The hard part is finding dead code. You will implement an analysis to detect dead code in Java programs by incorporating two data-flow analyses, i.e., *constant propagation* and *live variable analysis*. In this assignment, we focus on two kinds of dead code, i.e., *unreachable code* and *dead assignment*.

3.1.1 Unreachable Code

Unreachable code in a program will never be executed. We consider two kinds of unreachable code, *control-flow unreachable code* and *unreachable branch*, as introduced below.

Control-flow Unreachable Code. In a method, if there exists no control-flow path to a piece of code from the entry of the method, then the code is control-flow unreachable code. An example is the code following return statements. Return statements are exits of a method, so the code following them is unreachable. For example, the lines 4 and 5 in following code snippet are control-flow unreachable code:

```
int controlFlowUnreachable() {
    int x = 1;
    return x;
    int z = 42; // control-flow unreachable code
    foo(z); // control-flow unreachable code
}
```

Such code can be easily detected with control-flow graph (CFG) of the method. We just need to traverse the CFG from the method entry node, and mark reachable nodes, then the statements in other nodes are control-flow unreachable.

Unreachable Branch. For an if-statement, if its condition value is a constant, then one of its two branches is certainly unreachable in any executions. We call such branch unreachable branch, and code inside the branch is unreachable code. For example, in the following code snippet, the condition of if-statement at line 3 is always true, so its false-branch (line 6) is unreachable branch.

```
1 int unreachableBranch() {
2     int a = 1, b = 0, c;
3     if (a > b)
4         c = 2333;
5     else
6         c = 6666; // unreachable branch
7     return c;
8 }
```

To detect unreachable branches, we need to at first perform a constant propagation, which tells us whether the condition values are constants, then during CFG traversals, we can skip the corresponding unreachable branches.

3.1.2 Dead Assignment

A local variable that is assigned a value but is not read by any subsequent instruction is referred to as a dead assignment, and assigned variable is *dead variable* (opposite to *live variable*). Dead assignments do not affect program results, thus they can be eliminated. For example, lines 3 and 5 in below are dead assignments.

```
1 int deadAssign() {
2     int a, b, c;
3     a = 0; // dead assignment
4     a = 1;
5     b = a * 2; // dead assignment
6     c = 3;
7     return c;
8 }
```

To detect dead assignments, we need to at first perform a live variable analysis. For an assignment, if its LHS variable is a dead variable (i.e., not live), then we could mark it as a dead assignment, except one case as discussed below.

There is a caveat about dead assignment. Sometimes an assignment `x = expr` cannot be removed even `x` is a dead variable, as the RHS expression `expr` may have some side-effects. For example, `expr` is a method call (`x = foo()`) which could have many side-effects. For this issue, we have provided an API for you to check whether the RHS expression of an assignment may have some side-effects (described in Section 3.4). If so, then you should not report it as dead code for safety, even `x` is not live.

3.2 Scope

In this assignment, we deal with a subset of Java features. The scope for constant propagation is the same as Assignment 1, except that this time we need to handle six additional comparison operators for `int`, i.e., `==`, `!=`, `>=`, `>`, `<=`, `<`, which are used in the conditions of if-statement.

To keep control-flow graph (CFG) simple, we do not concern exceptions.

For branch statements, we only consider if-statement, and switch-case is not concerned. Note that in Jimple IR, while-loop and for-loop are also converted to if-statement (`IfStmt`). For example, this loop (written in Java):

```
while (a > b) {
    x = 0;
}
y = 1;
```

will be converted to Jimple IR like this:

```
label1:
  if a > b goto label2;
  goto label3;
label2:
  x = 0;
  goto label1;
label3:
  y = 1;
```

So, your implementation should be able to detect dead code related to loops, e.g., if `a` and `b` are constants and `a > b`, then your analysis should report `y = 1;` as dead code.

3.3 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).

3.3.1 Soot Classes for Constant Propagation

As mentioned in Section 3.2, the only difference between constant propagation of this assignment and Assignment 1 is that you need to handle six more comparison operators. In Soot, they correspond to six subclasses of `soot.jimple.BinopExpr`, i.e., `EqExpr` (==), `NeExpr` (!=), `GeExpr` (>=), `GtExpr` (>), `LeExpr` (<=), and `LtExpr` (<).

3.3.2 Soot Classes for Live Variable Analysis

To implement live variable analysis, you need to obtain the variables that are defined and used in a statement. `soot.Unit` provides two convenient APIs for this:

- `List<ValueBox> getDefBoxes()`
- `List<ValueBox> getUseBoxes()`

The defined/used `soot.Value`s are wrapped in list of `ValueBox`s (i.e., return values). You could obtain the defined local variable of a `Unit` like this:

```
List<ValueBox> defBoxes = unit.getDefBoxes();
for (ValueBox vb : defBoxes) {
    Value value = vb.getValue();
    if (value instanceof Local) {
        Local var = (Local) value; // <- defined variable
        …
    }
}
```

The used variables can be obtained in similar way. If you are interested in more details about `Unit`, `Value`, and `ValueBox`, please refer to these documentations:
https://github.com/Sable/soot/wiki/Fundamental-Soot-objects
https://www.sable.mcgill.ca/soot/tutorial/pldi03/tutorial.pdf (pages 34--37)

3.3.3 Soot Classes for Dead Code Detection

- `soot.Body`
  This class represents Jimple bodies of the methods. A `Body` contains all `Unit`s in the corresponding method, and you could iterate these `Unit`s through API `getUnits()`:

  ```
  Body body = …;
  for (Unit unit : body.getUnits()) {
      …
  }
  ```

- `soot.toolkits.graph.DirectedGraph<Unit>`
  This interface is used to represent the control-flow graphs of the methods, and their nodes are the `Units` of the methods. This interface is iterable, so you could iterate all nodes (`Units`) over a control-flow graph like this:

  ```
  DirectedGraph<Unit> cfg = …;
  for (Unit unit : cfg) {
      …
  }
  ```

  When you traverse a CFG, you could obtain the successors of a `Unit` by API `List<Unit> getSuccsOf(Unit)` like this:

  ```
  for (Unit succ : cfg.getSuccsOf(unit)) {
      …
  }
  ```

- `soot.jimple.IfStmt` (subclass of `Unit`)
  This class represents if-statements in the program.
  - `soot.Value getCondition()`: returns the condition expression. You will need to use constant propagation to compute the value of the expression.
  - `Unit getTarget()`: returns the first `Unit` of its true branch.

  `IfStmt` does not provide API to directly obtain the first `Unit` of its false branch. You could obtain it through the `Body` containing the `IfStmt`:

  ```
  Unit falseBranch = body.getUnits().getSuccOf(ifStmt);
  ```

- `soot.jimple.AssignStmt` (subclass of `Unit`)
  This class represents assignment (i.e., `x = …;`) in the program. You could obtain its LHS variable through this API `soot.Value getLeftOp()`, for example:

  ```
  AssignStmt assign = …;
  Local l = (Local) assign.getLeftOp();
  ```

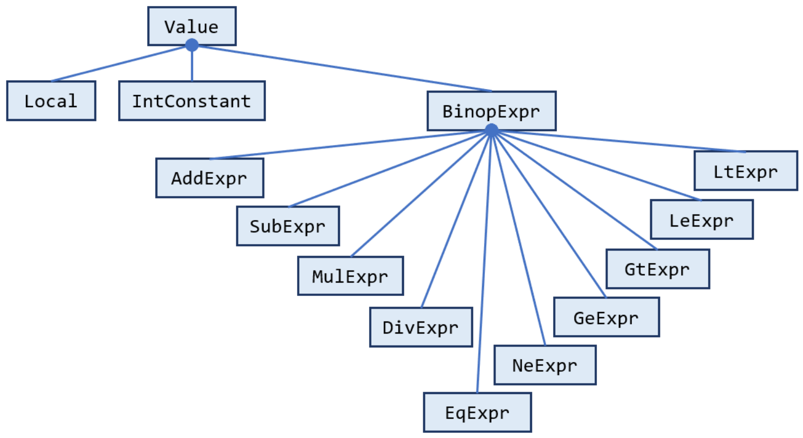
3.4 Bamboo Classes You Need to Know

To implement constant propagation, live variable analysis and dead code detection in Bamboo, you need to know the following classes.

3.4.1 Bamboo Classes for Constant Propagation

- `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 `Values`, as

shown in this partial class hierarchy:



Note that you need to handle six more comparison operators (all are the subclasses of `BinopExpr`) than Assignment 1.

For other classes, please refer to the documentation of Assignment 1.

3.4.2 Bamboo Classes for Live Variable Analysis

➢ `bamboo.dataflow.lattice.FlowSet<Local>`
This interface represents the sets of live variables (`soot.Local`), i.e., the data flows IN[s] and OUT[s] of each statement (`Unit`). You may use following APIs:

- `boolean add(Local)`: adds a local variable to this set, returns if the add operation changes the content of this set.
- `boolean remove(Local)`: removes a local variable from this set, returns if the remove operation changes the content of this set.
- `FlowSet<Local> union(FlowSet<Local>)`: unions another set into this set, returns this set.
- `FlowSet<Local> intersect(FlowSet<Local>)`: intersects another set and this set, returns this set.
- `FlowSet<Local> duplicate()`: returns a newly-created duplication of this set.
- `FlowSet<Local> setTo(FlowSet<Local>)`: sets the content of this set to the same as the given one, returns this set.

➢ `bamboo.dataflow.lattice.HashFlowSet<Local>`
This class is the HashSet-based implementation of `FlowSet`. When you need to create a new `FlowSet`, you could use constructor of `HashFlowSet`:

```
FlowSet<Local> set = new HashFlowSet<>();
```

➢ `bamboo.dataflow.analysis.livevar.LiveVariableAnalysis`
This class implements `DataFlowAnalysis`, and defines live variable analysis. It is incomplete, and you need to finish it as explained in Section 3.5.

3.4.3 Bamboo Classes for Dead Code Detection

➢ `bamboo.dataflow.analysis.deadcode.DeadCodeDetection`
This class implements dead code detection analysis. It is incomplete, and you need to finish it as explained in Section 3.5. We introduce some of its methods that you may need to use or understand.

- `Set<Unit> findDeadCode(Body b)`: starts dead code detection for the given method `Body`, returns the set of detected dead code. It invokes some several methods that you need to finish.
- `Unit getEntry(DirectedGraph<Unit>)`: returns the method entry `Unit` of the given CFG. You need this method for traversing the CFG.
- `boolean mayHaveSideEffect(AssignStmt)`: returns if an assignment may have some side-effects. You need this method to detect dead assignment.

➢ `bamboo.dataflow.analysis.deadcode.DeadCodeDetection.EdgeSet`
This class represents sets of control-flow edges. You will need this class to help detect unreachable branches.

- `void addEdge(Unit,Unit)`: adds a control-flow edge (represented by the given source node and target node) to this set.
- `boolean containsEdge(Unit,Unit)`: returns if this set contains the given control-flow edge (represented by its source node and target node)

➢ `bamboo.dataflow.analysis.deadcode.Main`
This is the main class of dead code detection, which performs the analysis for input Java program. We introduce how to run this class in Section 3.5.

3.5 Your Task [Important!]

In this assignment, you will need to finish three classes:
1. `bamboo.dataflow.analysis.constprop.ConstantPropagation`
2. `bamboo.dataflow.analysis.livevar.LiveVariableAnalysis`
3. `bamboo.dataflow.analysis.deadcode.DeadCodeDetection`
The details are given below.

➢ `bamboo.dataflow.analysis.constprop.ConstantPropagation`
Same as Assignment 1, you need to finish the following four APIs:

- `FlowMap meet(FlowMap,FlowMap)`

- `Value meetValue(Value,Value)`
- `boolean transfer(Unit,FlowMap,FlowMap)`
- `Value computeValue(soot.Value,FlowMap)`

The implementation for the first three APIs is the same as Assignment 1. As for `computeValue()`, you need to handle additional six comparison operators as mentioned in Section 3.2. So you could copy the relevant code from your Assignment 1, and then add code for handling comparison operators to `computeValue()`.

- Hint: the results of comparison expressions should be `boolean` value. For convenience, when the result is constant, you could use 1, i.e., `Value.makeConstant(1)`, to represent `true`, and 0 to represent `false`.

➢ `bamboo.dataflow.analysis.livevar.LiveVariableAnalysis`

You need to finish four APIs, which correspond to four parts of live variable analysis algorithm as shown below:

```
$IN[exit] = \emptyset$;        <- getEntryInitialFlow()
for (each basic block $B \setminus exit$)
    $IN[B] = \emptyset$;        <- newInitialFlow()
while (changes to any IN occur)
    for (each basic block $B \setminus exit$) {
        $OUT[B] = \boxed{\bigcup}^{\text{meet()}}_{S \text{ a successor of } B} IN[S]$;
        $\underbrace{IN[B] = use_B \cup (OUT[B] - def_B)}_{\text{transfer()}}$;
    }
```

Note that live variable analysis is a backward analysis. Bamboo implements backward analysis by performing forward analysis on a *reversed control-flow graph* (reversed CFG). In a reversed CFG, all control-flow edges of the original CFG are reversed, i.e., entry/exit nodes and IN/OUT flows of original CFG are swapped in the reversed one. Thus, for live variable analysis, entry node in Bamboo corresponds to exit node in the algorithm, and IN (OUT) flows in Bamboo correspond to OUT (IN) flows in the algorithm.

- `FlowSet<Local> getEntryInitialFlow(Unit)`

  This function returns the initial data flow for entry nodes.

- `FlowSet<Local> newInitialFlow()`

  This function returns the initial data flow for non-entry nodes.

- `FlowSet<Local> meet(FlowSet<Local>,FlowSet<Local>)`

  This is the **meet function** of live variable analysis, which meets two

`FlowSet`s of local variable and returns the resulting `FlowSet`. This function is used to handle control-flow influences.

- `boolean transfer(Unit,FlowSet<Local>,FlowSet<Local>)`

  This is the **transfer function** of live variable analysis.

➢ `bamboo.dataflow.analysis.deadcode.DeadCodeDetection`

You need to finish three methods to detect the dead code described in Section 3.1.

- `EdgeSet findUnreachableBranches(Body,DirectedGraph<Unit>)`

  You need to iterate the `Unit`s of given CFG to detect unreachable branches of if-statements (`IfStmt`). If the condition value of an if-statement is always true, then you should add false branch, i.e., the edge from the if-statement (`IfStmt`) to the first `Unit` of the false branch, to the resulting `EdgeSet`. Similarly, if the condition value is always false, add an edge for true branch.

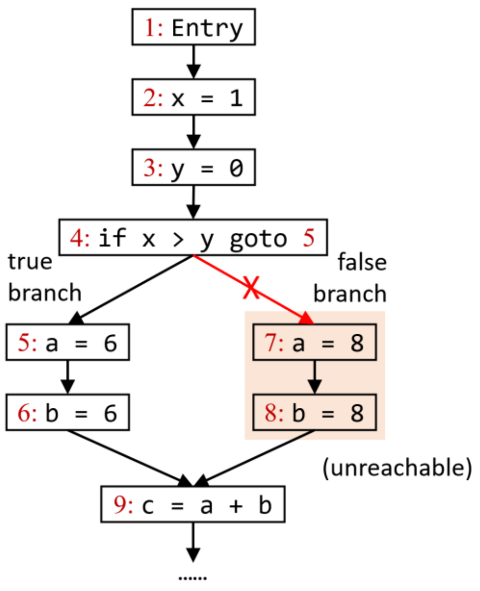
  ❖ Hint 1: there is a variable `constantMap` in this method, which holds the map from each `Unit` in the method to the `FlowMap`, i.e., the constant propagation result at the `Unit`.

  ❖ Hint 2: you could use `ConstantPropagtion.computeValue()` to compute the value for condition expression of if-statements. Since `computeValue()` is an instance method, and `ConstantPropagation` uses singleton pattern (the only instance can be obtained by calling `v()`), you could call `computeValue()` in this way:

  ```
  ConstantPropagation.v().computeValue(…);
  ```

- `Set<Unit> findUnreachableCode(DirectedGraph<Unit>,EdgeSet)`

  You need to iterate the `Unit`s in the given CFG and detect unreachable code by traversing the CFG from entry node (pointed by variable `entry`). During the traversal, you should skip unreachable branches in given `EdgeSet`.



Here we further explain the role of `EdgeSet` using the example in the above figure. In this example, the false branch is an unreachable branch as `x` is always larger than `y`. So, we should add edge 4→7 to the resulting `EdgeSet` of `findUnreachableBranches()`. Then, when we traverse the CFG to detect unreachable code in `findUnreachableCode()`, we could skip edge 4→7, and naturally discover that statements with labels 7 and 8 are unreachable from the entry node.

- ❖ Hint: you could examine the control-flow graph of each method in folders under `sootOutput/`, to help debug and test. The CFGs are stored as .dot files, and can be visualized by Graphviz (https://graphviz.org/download/).

◆ `Set<Unit> findDeadAssignments(Body)`

You need to iterate the `Units` in the given `Body` and detect dead assignments (`AssignStmt`) based on the results of live variable analysis.

- ❖ Hint 1: there is a variable `liveVarMap` in this method, which holds the map from each `Unit` in the method to a `FlowSet<Local>`, i.e., the set of live variables at the `Unit`.
- ❖ Hint 2: Do not mark the statements that may have side effects as dead assignment. Use `mayHaveSideEffect()` to ensure this.

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

3.6 Run Dead Code Detection as an Application

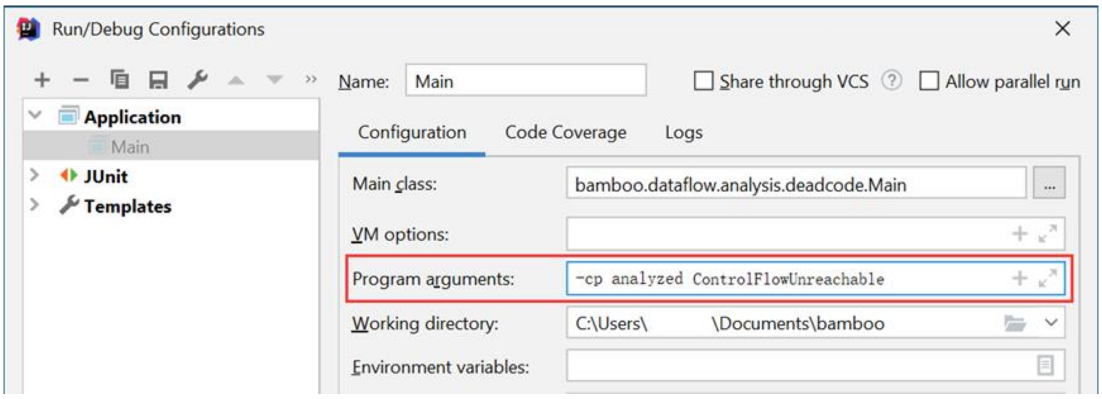
As mentioned in Section 3.4, the main class of dead code detection is

```
bamboo.dataflow.analysis.deadcode.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 `ControlFlowUnreachable.java` in class path `analyzed/`, just configure program arguments in IntelliJ IDEA as follows:



Name: Main

Main class: bamboo.dataflow.analysis.deadcode.Main

VM options:

Program arguments: -cp analyzed ControlFlowUnreachable

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

Environment variables:

For each method of input class, Bamboo performs constant propagation, live variable analysis, and dead code detection in order, and outputs the results of every analysis for each method. 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>”
```

3.7 Test Dead Code Detection 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 dead code detection (in form of line numbers and the corresponding Jimple `Units`) for some methods. You could analyze these test inputs by running test class (powered by JUnit):

```
bamboo.dataflow.analysis.deadcode.DCDTest
```

This test class analyzes all provided Java classes in `analyzed/`, and compares the given analysis results to the expected results. If your implementation of dead code detection 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`
- `LiveVariableAnalysis.java`
- `DeadCodeDetection.java`

The naming convention is of the zip file is:

```
<STUDENT_ID>-<NAME>-A2.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!