Report for *HW1 LiveVariableAnalysis*

Hongchen Cao
Shanghaitech University
Shanghai, China
caohch1@shanghaitech.edu.cn

***Abstract*—Report for *CS224 Homework1: Live Variable Analysis*, including project structure, algorithm introduction, and Implementation description.**

I. Project Structure

Following is the directory tree of the source code of the project:

```
LiveVariableAnalysis
 ├─src
 │  ├─main
 │  │  ├─java
 │  │  │  └─indi
 │  │  │     ├─CFGNode.java
 │  │  │     ├─MyCFG.java
 │  │  │     └─LiveVariableAnalysis.java
 │  │  └─resources
 │  └─test
 ├─out
 └─pox.xml
```

Here we give a brief description of each part of the project:

- **pom.xml:** Maven configuration of the project.
- **out:** Directory holds built artifacts(i.e., jar package).
- **src/main/java/indi:** Directory holds key codes for the project.
- **src/main/resources:** Directory holds resource files like MANIFEST.MF.
- **src/test:** Directory holds test files of the project.

For the three key java file (i.e., **CFGNode.java**, **MyCFG.java**, and **LiveVariableAnalysis.java**), we describe them in detail in Sec. III.

II. Algorithm

Listing 1. Algorithm for LiveVariableAnalysis

```
1  IN[exit] = $\emptyset$;
2  for (each basic block B\exit)
3      IN[B] = $\emptyset$;
4  while (changes to any IN occur)
5      for (each basic block B\exit) {
6          OUT[B] = $\cup_{\text{S a successor of B}}$ IN[S];
7          IN[B] = use$_B$$\cup$(OUT[B] - def$_B$);
8      }
```

List. 1 shows the pseudocode of the live variable analysis. There are three key points in the algorithm need to be implemented:

- **MeetInto:** Compute the merge set of IN sets of a node's successors, which corresponding to Line#6.
- **TransferNode:** Compute the IN set of a node based on transfer function, which corresponding to Line#7.
- **Solver:** Main iterable solver, which corresponding to Line#4&Line#5.

III. Implementation

The main idea of my implementation is to first create a control flow graph (CFG) and then iterably compute the IN and OUT set of each node while traversing CFG. The immaterial member methods (e.g., getter and setter methods) and attributes (e.g, Logger from org.slf4j) will be ignored in the following sections.

A. CFGNode

CFGNode is the data structure for every node in CFG. Each node contains five member attributes as follows:

- **Set<Local> inSet:** A HashSet contains all Locals belong to IN set of a node.
- **Set<Local> outSet:** A HashSet contains all Locals belong to OUT set of a node.
- **Unit unit:** The Jimple statement.
- **boolean isTail:** Used to indicate if it is the entry node of the CFG.
- **boolean isHead:** Used to indicate if it is the exit node of the CFG.

B. MyCFG

MyCFG is fine-tuned from **UnitGraph** which is a built-in class of **Soot**. It has two member attributes as follows:

- **UnitGraph unitGraph:** Built-in class of Soot which generated from a JimpleBody.
- **ArrayList<CFGNode> cfgNodes:** An arrayList contains all **CFGNode**s of the CFG.

MyCFG implementes has three key functions as follows:

- **CFGNode searchNodeByUnit(Unit unit):** This function traverse the **cfgNodes** and return the **CFGNode** whose **uint** equals to the paramater.
- **ArrayList<CFGNode> getPredsOf(CFGNode cfgNode):** This function return the predecessor nodes of the paramater, which is fine-tuned from function **List<Unit> getPredsOf(Unit u)** of the class **UnitGraph**.
- **ArrayList<CFGNode> getSuccsOf(CFGNode cfgNode):** This function return the successors nodes of the

paramater, which is fine-tuned from function **List<Unit> getSuccsOf(Unit u)** of the class **UnitGraph**.

C. LiveVariableAnalysis

This is the main class to implemente the live variable analysis. We now describe the key member methods one by one according to the order in which they are called to complete the analysis.

1) **void setupSoot()** is used to indicate the environment variable $JAVA_HOME and process directory for Soot. We invoke **Options.v().set_prepend_classpath()** and **Options.v().set_process_dir()** to complete two tasks. So make sure $JAVA_HOME is set correctly in your environment.
2) **LiveVariableAnalysis(String className, String methodName, String dir)** is the constructor method of the class. Firstly, it invokes **void setupSoot()**. Secondly, it load the class and its main method. Note that your class (i.e., java source code) must be compilable or an error will occur. Thirdly, generating a **MyCFG** and store it into the member attribute.
3) **void newBoundaryFact()** sets **inSet**s of all **CFGNode**s whose **isTail** equals true in **MyCFG** to empty.
4) **void newInitialFact()** sets **inSet**s of all **CFGNode**s in **MyCFG** to empty except for whose **isTail** equals true.
5) **void meetInto(CFGNode upNode, CFGNode downNode)** merges the **inSet** of **downNode** into the **outSet** of **upNode**.
6) **void transferNode(CFGNode node)** is is slightly more complex. Firstly, we use **unit.getDefBoxes()** and **unit.getUseBoxes()** to get **Set<Local> defLocals** and **Set<Local> useLocals**. Secondly, we remove all Locals in **defLocals** from **outSet** of the paramater. Thirdly, we add all Locals in **useLocals** into **outSet** of the paramater.
7) **int doAnalysis()** is the most important member method. Firstly, it invokes **newBoundaryFact()** and **newInitialFact()**. Secondly, we create a **Queue<CFGNode>** to help us traverse CFG and add all **CFGNode** whose **isTail** equals true. Thirdly, we create a double nested while loop. For the inner loop, we stop it until the queue is empty. In each iteration, we first get the first element in queue and invoke **MyCFG.getPredsOf(node)** for further pushing the return values into queue. Then we invoke **meetInto(node, succNode)** and **transferNode(node)** to update **inSet** and **outSet** of each node. Here we also record whether these sets are changed. If no change happened after one whole inner loop, we break the outer loop. Otherwise, we start a new turn of the inner loop.
8) Finally, we utilize **void writeFile(String filePath, String[] data)** to write the .txt file into the output directory.

Through the above efforts, for a single java class, we can do the live variable analysis by only one line of code **new LiveVariableAnalysis(”test0”, ”main”, ”./test/”).doAnalysis()**. There are also member methods like **void doAnalysisAndShow()** and **void doAnalysisAndShowWithArg(String filePath)**. They are based on **int doAnalysis()** and the difference is they display some informations about the preocess of the analysis, which may be more user-friendly and useful for debuging. For further details on the implementation, you can read the source code of the submission.