Report for *HW2 DeadCodeAnalysis*

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

***Abstract*—Report for *CS224 Homework2: Dead Code 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
│   │   │       ├── Util
│   │   │       ├── LiveVariable
│   │   │       │   └── LiveVariableAnalysis
│   │   │       ├── ConstantPropagation
│   │   │       │   ├── ConstantPropagation
│   │   │       │   ├── ConstantPropagationFlow
│   │   │       │   └── ConstantPropagationValue
│   │   │       └── DeadCodeDetection
│   │   └── 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. Specifically, **DeadCodeDetection** is the key file for the project, **LiveVariable** is based on LiveVariable-Analysis in HW1, **ConstantPropagation** is used to do constant propagation based on worklist algorithm, and **Util** contains some helper functions and classes.
- **src/main/resources:** Directory holds resource files like MANIFEST.MF.
- **src/test:** Directory holds test files of the project.

For the four key java file (i.e., **DeadCodeDetection.java**, **ConstantPropagation.java**, **ConstantPropagationFlow.java**, and **ConstantPropagationValue.java**), we describe them in detail in Sec. III.

II. Algorithm

Listing 1. Algorithm for DeadCodeDetection

```
HashSet deadCodeDetection(ControlFlowGraph cfg) {
    deadCodeSet = $\emptyset$;
    deadCodeSet += controlFlowUnreachableSet(cfg);
    deadCodeSet += unreachableBranchSet(cfg);
    deadCodeSet += deadAssignmentSet(cfg);
    return deadCodeSet;
}
```

As shown in List. 1, we divide the DeadCodeDetection into three parts, including control-flow unreachable detection, unreachable branch detection, and dead assignment detection.

Listing 2. Algorithm for ConstantPropagation

```
OUT[entry] = $\emptyset$
for (each basic block B is not entry)
    OUT[B] = $\emptyset$
Worklist $\leftarrow$ all basic blocks
while (Worklist is not empty)
    Pick a basic block B from Worklist
    old_OUT = OUT[B]
    IN[B] = $\cup_{\text{P a predecessor of B}}$ OUT[P];
    OUT[B] = gen$_B \cup$(IN[B] − kill$_B$);
    if (old_OUT $\neq$ OUT[B])
        Add all successors of B to Worklist
```

List. 2 shows the pseudocode of the constant propagation.

III. Implementation

Next we introduce the four key classes in the project, and we describe the important member functions and attributes. For the other classes, they are adopted from *HW1* or are helper classes. Thus, we don't describe them in detail.

A. DeadCodeDetection

This class contains three important functions corresponding to three kinds of analysis.

• **controlFlowUnreachableDetection().** We traverse the control flow graph and record whether each statement is reachable or not, and if so, add it to a hash set. Finally the statements that are not in this set but are in the analyzed method are stored in a new hash set and returned.

• **unreachableBranchDetection().** We first do the constant propagation, then we traverse the control flow graph and for each statement we determine if it is an if or $switch$ statement. if it is an if statement we further determine if its conditional expression is true and record the unreachable edges in the control flow graph . If it is a $switch$ statement, we determine if the conditional expression is a constant and record the unreachable edges in the control flow graph. Finally, we traverse the control flow graph again, and for each statement, if it is an endpoint pointed to by the previously recorded unreachable edge, we store it in a hash set.

• **deadAssignmentDetection().** We first do a live variable analysis, then we traverse the control flow graph and for each statement we determine if it is an assignment statement. If so, we further determine if its right value calls a function. If not, then we add the statement to a hash set and finally we return the set.

B. ConstantPropagationValue(CPValue)

CPValue is used to indicate whether a variable is a constant or not. Specifically, it includes NAC, UNDEF and specific constant values.

• **meet(CPValue v1, CPValue v2).** Depending on the values of the two CPValue, we perform different aggregation operations. This is mainly implemented to cope with one of the values being NAC or UNDEF.

C. ConstantPropagationFlow(CPFlow)

Our main job in this class is to maintain a map that keeps track of the variables and their corresponding CPValue. there are two main functions that ensure the proper behavior of this class.

• **meet(CPFlow f1, CPFlow f2).** We get the respective maps in f1 and f2, and then add the keys in both maps to a hash set. Then we iterate through the set, and for each key, we get its corresponding CPValue in f1 and f2, and call the *calculate* function to calculate the result. Finally, the new key-value pair is added to the map of the instance that called this function.

• **calculate(Value value).** This function will be called recursively. If value is a Local type, the map is queried directly to return the corresponding CPValue. if it is an IntConstant type, a new CPValue is created and returned. If BinopExpr, we get the values on both sides of the operator and recursively call the function to get the return value. Subsequently, if the values on both sides are constants (i.e., not NAC or UNDEF), the corresponding calculations are performed depending on the operator.

D. ConstantPropagation

We maintain a queue. First we add the entry of the analyzed function to this queue, and then we open a $while$ loop until the queue is empty. In each iteration, we fetch the first element of the queue. We call the calculate function in CPFlow to compute the updated CPFlow for each statement, and if there is nothing to update, we do not add its successor statements to the queue, and vice versa.