Programming Assignment 3: Dead Code Detection

Course “Static Program Analysis” @Nanjing University
Assignments Designed by Tian Tan and Yue Li
Due: 23:00, Sunday, October 24, 2021

1 Assignment Objectives

- Implement a dead code detector for Java.

Dead code *elimination* is a common compiler optimization in which dead code is removed from a program, and its most challenging part is the *detection* of dead code. In this programming assignment, you will implement a dead code detector for Java by incorporating the analyses you wrote in the previous two assignments, i.e., *live variable analysis* and *constant propagation*. In this document, we will define the scope of dead code (in this assignment) and your task is to implement a detector to recognize them.

2 Introduction to Dead Code Detection

Dead code is the part of program which is unreachable (i.e., never be executed) or is executed but whose results are never used in any other computation. Eliminating dead code does not affect the program outputs, meanwhile, it can shrink program size and improve the performance. In this assignment, we focus on two kinds of dead code, i.e., *unreachable code* and *dead assignment*.

2.1 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. 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 code at lines 4 and 5 below is control-flow unreachable:

```
1 int controlFlowUnreachable() {
2     int x = 1;
3     return x;
4     int z = 42; // control-flow unreachable code
5     foo(z); // control-flow unreachable code
6 }
```

Detection: 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 and mark reachable statements. When the traversal finishes, the unmarked statements are control-flow unreachable.

Unreachable Branch. There are two kinds of branching statements in Java: *if* statement and *switch* statement, which could form unreachable branches.

For an if-statement, if its condition value is a constant, then one of its two branches is certainly unreachable in any executions, i.e., unreachable branch, and the code inside that branch is unreachable. 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 a unreachable branch.

```
1 int unreachableIfBranch() {
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 }
```

For a switch-statement, if its condition value is a constant, then case branches whose values do not match that condition may be unreachable in any execution and become unreachable branches. For example, in the following code snippet, the condition value (`x`) of switch-statement at line 3 is always 2, thus the branches "`case 1`" and "`default`" are unreachable. Note that although branch "`case 3`" does not match the condition value (2) either, it is still reachable as the control flow can fall through to it via branch "`case 2`".

```
1 int unreachableSwitchBranch() {
2     int x = 2, y;
3     switch (x) {
4         case 1: y = 100; break; // unreachable branch
5         case 2: y = 200;
6         case 3: y = 300; break; // fall through
7         default: y = 666; // unreachable branch
8     }
9     return y;
10 }
```

Detection: to detect unreachable branches, we need to perform a constant propagation in advance, which tells us whether the condition values are constants, and then during CFG traversal, we do not enter the corresponding unreachable branches.

2.2 Dead Assignment

A local variable that is assigned a value but is not read by any subsequent instructions is referred to as a dead assignment, and the assigned variable is *dead variable* (opposite to *live variable*). Dead assignments do not affect program outputs, and thus can be eliminated. For example, the code at lines 3 and 5 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 }
```

Detection: to detect dead assignments, we need to perform a live variable analysis in advance. 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 = m()`) 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 side-effects (described in Section 3.2). If so, you should not report it as dead code for safety, even `x` is not live.

3 Implementing a Dead Code Detector

3.1 Tai-e Classes You Need to Know

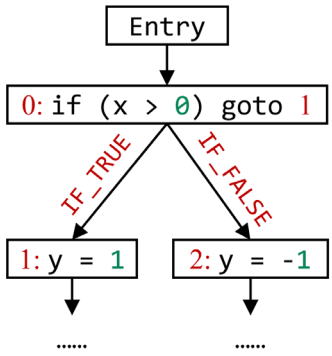
To implement a dead code detector, you need to know `CFG`, `IR`, and the classes about the analysis results of live variable analysis and constant propagation (such as `CPFact`, `DataflowResult`, etc.), which you have been used in previous assignments. Below we will introduce more classes about CFG and IR that you will use in this assignment.

- `pascal.taie.analysis.graph.cfg.Edge`
  This class represents the edges in CFG (as a reminder, the nodes of CFG are `Stmt`). It provides method `getKind()` to obtain the kind of the edges (to understand the meaning of each kind, you could read the comments in class `Edge.Kind`), and you could check the *edge kind* like this:
  ```
  Edge<Stmt> edge = ...;
  if (edge.getKind() == Edge.Kind.IF_TRUE) { ... }
  ```

In this assignment, you need to concern four kinds of edges: `IF_TRUE`, `IF_FALSE`, `SWITCH_CASE`, and `SWITCH_DEFAULT`. `IF_TRUE` and `IF_FALSE` represent the two out edges from if-statement to its branches, as shown in this example:

```
void ifBranch(int x) {
    int y;
    if (x > 0) {
        y = 1;
        …
    } else {
        y = -1;
        …
    }
}
```

Code

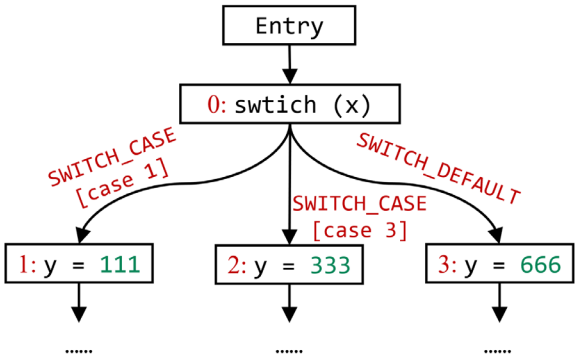


CFG

`SWITCH_CASE` and `SWITCH_DEFAULT` represent the out edges from switch-statement to its case and default branches, as shown in this example:

```
void switchBranch(int x) {
    int y;
    switch (x) {
        case 1: y = 111; …
        case 3: y = 333; …
        default: y = 666; …
    }
}
```

Code



CFG

For `SWITCH_CASE` edges, you could obtain their case values (e.g., `1` and `3` in the above example) by `getCaseValue()` method.

- `pascal.taie.ir.stmt.If` (subclass of `Stmt`)
  This class represents if-statement in the program.

Note that in Tai-e's IR, while-loop and for-loop are also converted to `If` statement. For example, this loop (written in Java):

```
while (a > b) {
    x = 233;
}
y = 666;
```

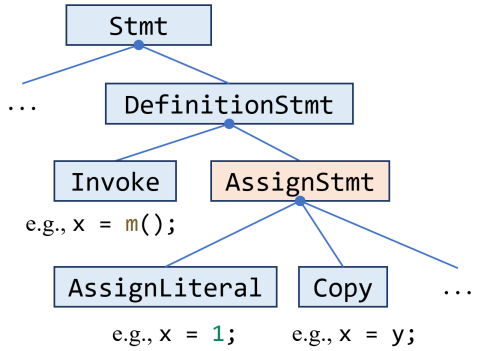
will be converted to Tai-e's IR like this:

```
0:  if (a > b) goto 2;
1:  goto 4;
2:  x = 233;
3:  goto 0;
4:  y = 666;
```

Hence, 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 that `x = 233` is dead code.

- `pascal.taie.ir.stmt.SwitchStmt` (subclass of `Stmt`)
  This class represents switch statement in the program. You could read its source code and comments to decide how to use it.

- `pascal.taie.ir.stmt.AssignStmt` (subclass of `Stmt`)
  This class represents assignment (i.e., `x = …;`) in the program. You might think it as similar to class `DefinitionStmt` that you have seen before. The relation between these two classes is shown in this partial class hierarchy:



  Actually, `AssignStmt` is one of two subclasses of `DefinitionStmt` (another one is `Invoke`, which represents method call/invocation in the program). It means that `AssignStmt` represents all assignments except the ones whose right-hand side expression is method call. `AssignStmt` is used in this assignment to recognize dead assignments. As mentioned in Section 2.2, method calls could have many side-effects, thus the statements like `x = m();` will not be considered as dead assignments even though `x` is not used later. Thus, you only need to consider `AssignStmt` as dead assignments.

- `pascal.taie.analysis.dataflow.analysis.DeadCodeDetection`
  This class implements dead code detection. It is incomplete, and you need to finish it as explained in Section 3.2.

3.2 Your Task [Important!]

Your task is to implement one API of `DeadCodeDetection`:

- `Set<Stmt> analyze(IR)`

This method takes an `IR` of a method as input, and it is supposed to return a set of dead code in the `IR`. Your task is to recognize two kinds of dead code described in Section 2, i.e., *unreachable code* and *dead assignments*, and add them into the resulting set.

Dead code detection depends on the analysis results of live variable analysis and constant propagation. Thus, to make dead code detection work, you need to finish `LiveVariableAnalysis.java` and `ConstantPropagation.java`. You could copy your implementation from previous assignments. Besides, you also need a complete worklist solver that supports both forward and backward data-flow analyses. You could copy your implementation of `Solver.java` and `WorkListSolver.java` from Assignment 2, and then complete `Solver.initializeBackward()` and `WorkListSolver.doSolveBackward()`. Note that you don't need to submit these source files in this assignment, so that even though your implementation of previous assignments is not entirely correct, it does not affect your score of this assignment.

- Hints: 1) In this assignment, Tai-e will automatically run live variable analysis and constant propagation before dead code detection. We have provided the code in `DeadCodeDetection.analyze()` to *obtain* the analysis results of these two analyses for the given `IR`, so that you could directly use them. Besides, this `analyze()` method contains the code to obtain the `CFG` for the `IR`.
  2) As mentioned in Section 2.2, the RHS expression of some assignments may have side-effects, and thus cannot be considered as dead assignments. We have provided an auxiliary method `hasNoSideEffect(RValue)` in `DeadCodeDetection` for you to check if a RHS expression may have side-effects or not.
  3) During the CFG traversal, you should use `CFG.outEdgesOf()` to query the successors to be visited later. This API returns the outgoing edges of each node in the CFG, so that you could use the information on the edges (introduced in Section 3.1) to help detect unreachable branches.
  4) You could use `ConstantPropagation.evaluate()` to evaluate the condition value of if- and switch-statements when detecting unreachable branches.

4 Run and Test Your Implementation

You can run the analysis as described in *Tai-e Manual for Assignments*. Tai-e performs live variable analysis, constant propagation, and dead code detection for each method of input class. To help debugging, it outputs the results of all three analyses:

```
-------------------- <DeadAssignment: void deadAssign()> (livevar) --------------------
[0@L4] x = 1; null
[1@L5] %intconst0 = 2; null
[2@L5] y = x + %intconst0; null
[3@L6] %intconst1 = 3; null
[4@L6] z = x + %intconst1; null
[5@L7] invokevirtual %this.<DeadAssignment: void use(int)>(z); null
[6@L8] a = x; null
[7@L8] return; null

-------------------- <DeadAssignment: void deadAssign()> (constprop) --------------------
[0@L4] x = 1; null
[1@L5] %intconst0 = 2; null
[2@L5] y = x + %intconst0; null
[3@L6] %intconst1 = 3; null
[4@L6] z = x + %intconst1; null
[5@L7] invokevirtual %this.<DeadAssignment: void use(int)>(z); null
[6@L8] a = x; null
[7@L8] return; null

-------------------- <DeadAssignment: void deadAssign()> (deadcode) --------------------
```

The OUT facts are `null` and no dead code is reported as you have not finished the analyses yet. After you implement the analyses, the output should be:

```
-------------------- <DeadAssignment: void deadAssign()> (livevar) --------------------
[0@L4] x = 1; [%this, x]
[1@L5] %intconst0 = 2; [%intconst0, %this, x]
[2@L5] y = x + %intconst0; [%this, x]
[3@L6] %intconst1 = 3; [%intconst1, %this, x]
[4@L6] z = x + %intconst1; [%this, x, z]
[5@L7] invokevirtual %this.<DeadAssignment: void use(int)>(z); [x]
[6@L8] a = x; []
[7@L8] return; []

-------------------- <DeadAssignment: void deadAssign()> (constprop) --------------------
[0@L4] x = 1; {x=1}
[1@L5] %intconst0 = 2; {%intconst0=2, x=1}
[2@L5] y = x + %intconst0; {%intconst0=2, x=1, y=3}
[3@L6] %intconst1 = 3; {%intconst0=2, %intconst1=3, x=1, y=3}
[4@L6] z = x + %intconst1; {%intconst0=2, %intconst1=3, x=1, y=3, z=4}
[5@L7] invokevirtual %this.<DeadAssignment: void use(int)>(z); {%intconst0=2, %intconst1=3, x=1, y=3, z=4}
[6@L8] a = x; {%intconst0=2, %intconst1=3, a=1, x=1, y=3, z=4}
[7@L8] return; {%intconst0=2, %intconst1=3, a=1, x=1, y=3, z=4}

-------------------- <DeadAssignment: void deadAssign()> (deadcode) --------------------
[2@L5] y = x + %intconst0;
[6@L8] a = x;
```

In addition, Tai-e outputs the control-flow graphs of the methods it analyzes to folder `output/`. The CFGs are stored as .dot files, and can be visualized by Graphviz (https://graphviz.org/download/).

We provide test driver `pascal.taie.analysis.dataflow.analysis.DeadCodeTest` for this assignment, and you could use it to test your implementation as described in *Tai-e Manual for Assignments*.

5 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!

6 Submission of Assignment

Your submission should be a zip file, which contains your implementation of

- `DeadCodeDetection.java`

The naming convention your submission is: `<STUDENT_ID>-<NAME>-A3.zip`

Please submit your assignment to 教学立方.

7 Grading

The points will be allocated for correctness. We will use your submission to analyze the given test files from the `src/test/resources/` directory, as well as other tests of our own, and compare your output to that of our solution.

Good luck!