XYPointerAnalysis

Authors: Peiyi Sun && Dexin Liu

1. Introduction

This is a pointer analysis framework for Java, based on *Soot* analysis framework. Actually, it is the first project for course *Software Analysis in Fall 2018*.

2. Strategies

This work is mostly based on *Points-to Analysis for Java Using Annotated Constraints[1]*, published in 2001 on *Oopsula*. This is a general-purpose points-to analysis for Java based on Anderson's points-to analysis for C, with constraint-based approach that employs annotated inclusion constraints. It's shown in the paper that it models virtual calls and object fields precisely and efficiently.

In the subsection below, we'll illustrate some basic principles of the whole method.

2.1 Constraint Language and Annotated Constraint Graphs

To solve the problem, we first need annotated set-inclusion constraints of the form $L \subset_a R$. a is chosen from a given set of annotations.

The nodes in the graph can be classified as variables, sources, and sinks. The graph only contains edges of the following forms: $Source \subset_a Var$, $Var \subset_a Var$, $Var \subset_a Sink$

We use annotated constraint graphs based on the inductive form representation. The graphs are represented with adjacency lists $pred(n)$ and $succ(n)$.

2.2 Solving Systems of Annotated Constraints

The system is solved by computing the closure of the graph under the following transitive closure rule:

TRANS

$$If \begin{cases} < L, a > \in pred(v) \\ < R, b > \in succ(v) \\ Match(a, b) \end{cases}$$

The new transitive constraint is created only if the annotations of the two existing constraints “match”—that is, only if Match(a,b) holds.

Match Constraints

$$Match(a, b) = \begin{cases} true & if\ a\ or\ b\ is\ the\ empty\ annotation\ \epsilon \\ true & if\ a = b \\ false & otherwise \end{cases}$$

The annotation of the new constraint is :

$$a \circ b = \begin{cases} a & if\ b = \epsilon \\ b & if\ a = \epsilon \\ \epsilon & otherwise \end{cases}$$

2.3 Constraints for Assignment Statements

We use an example to show how to construct constraints for assignment statements.

$$< l = new\ o_i > \Rightarrow \{ref(o_i, v_{o_i}, \overline{v}_{o_i}) \subset v_l\}$$
$$< l = r > \Rightarrow \{v_r \subset v_l\}$$
$$< l.f = r > \Rightarrow \{v_l \subset proj(ref, 3, u), v_r \subset_f u\}, u\ fresh$$
$$< l = r.f > \Rightarrow \{v_r \subset proj(ref, 2, u), u \subset_r v_l\}, u\ fresh$$

Through **TRANS** operation, we can derivate that

$$ref(o_2, v_{o2}, \overline{v}_{o2}) \subseteq v_r$$

2.4 Handling of Virtual Calls

For every virtual call in the program, our analysis generates a constraint according to the following rule:

$$< l = r_0.m(r_1, \ldots, r_k) > \Rightarrow v_{r0} \subseteq_m lam(\overline{0}, \overline{v_{r1}}, \ldots, \overline{v_{rk}}, v_l)$$

We use a precomputed lookup table to determine the corresponding run-time target method, based on the class of the receiver object.

For more details about the method, please refer to the paper.

3. Code Structure

Here we illustrate the code structure of our program.

```
src
|-- main
    |--annotated_anderson_analysis              #The handler part
        |--constraint_graph_node                #Structure of Graph Node
            |--BasicConstraintGraphNode.java    #Structure
            |--ConstraintConstructor.java       #Structure
            |--ConstraintObjectConstructor.java #Structure
            |--ConstraintVariable.java          #Structure
        |--ConstraintAnnotation.java            #Structure for Annotation
        |--ConstraintConvertUtility.java        #Main part for convertion operation.
        |--ConstraintGraph.java                 #Main part for graph operation.
        |--LookUpTable.java                     #Structure for LookUpTable
```

4. How to use