Flow Based Bug Detection

Oskar Kari

January 2021

Abstract

Program analysis is an important tool in modern software developement. In this paper a program analysis tool that solves the redundant assignment problem will be described. This tool will be tested on a real program as well as on smaller test programs. The performance will be analysed and some suggestions for further work will be done.

1 Introduction

Program analysis is useful for code optimization, detecting bugs, increase code quality and reduce developement costs. There are many different forms of program analysis [2]. There is Dynamic programing analysis that analyses the code at runtime and there is static program analysis that analysis the code during the compilation. There are also different kinds of static and dynamic program analysis. For example there are interactive program analysis were the user must interact with the program analysis tool, like debuggers. Then there are automatic program analysis tools where the user calls the tool and it returns some result.

In this project I will create an automatic static program analysis tool that detects a common bug pattern in the Java programming language. The tool will find redundant assignments. For example in the following java code

$$b = a;$$

$$c = b;$$

$$a = c;$$

the last assignment is redundant and can be removed. This is useful both for optimization and also for detecting possible bugs. To do this analysis the tool will analyse how data flows in the program. The Java compiler that I will be using is ExtendJ [1] and the tool that extracts the control flow graph is IntraJ, the excecution pipeline is summarized in Figure 1.

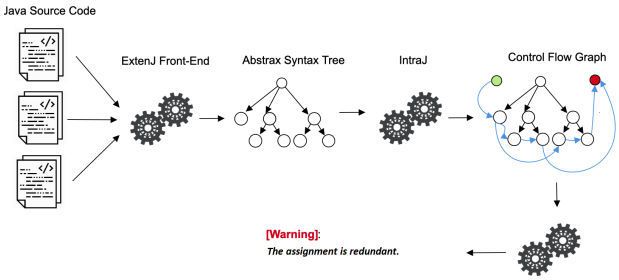


Figure 1: An overview of how ExtendJ and IntraJ works together with the analysis described in this paper. First the compiler ExtendJ creates an abstract syntax tree. From this abstract syntax tree the tool IntraJ creates a control flow graph. From this control flow graph it is then possible to build an extension of your own. This tool can give different warnings or information.

2 Dataflow Analysis

2.1 Control Flow Graph

A control flow graph (CFG) is a graph $G(V, E)$ that shows how a program executes. Normally the CFG graph is built from the intermediate code but IntraJ extracts the CFG directly from the abstract syntax tree. The benefit of this is speed because the compiler does not have to execute as many steps before the CFG can be made.

In the graph that IntraJ extracts each statement and expression in the program is represented by a node V in the CFG. The edges E are directed and are drawn between statements/expressions that can be executed sequentially. So if statement B can follow from statement A then there is an edge from node A to node B. There can be many ingoing edges to a node if there are many statements that can reach the node. Likewise there can be many outgoing edges if many statements can follow from a node.

Every node in the CFG defines three different sets; the set of data from ingoing edges, the set of data from outgoing edges and a transfer function that changes the data. The ingoing edges provides information to the node about what happened earlier in the program. The transform can make some kind of modification to this information and then this information is sent to the sequential nodes by the outgoing edges. What kind of data the node gets as input, transforms and then outputs depends on the program analysis problem that is being solved.

2.2 Using the Control Flow Graph in Dataflow Analysis

When the CFG is used in dataflow analysis the information that is of interest flows through the edges of the CFG. In this project the information traverses the CFG starting from the beginning of a method and ending when it reaches an exit from the method.

3 Method

3.1 Redundant Assignments

To solve the redundant assignment problem four sets will be defined for every node in the CFG. These are $\mathbf{in}_n$, $\mathbf{out}_n$, $\mathbf{gen}_n$, $\mathbf{kill}_n$. The $\mathbf{gen}_n$ set contains all the substitutions generated in this node if any. For example if the code in the node is

$$a = b;$$

where both a and b are compatible types then the $\mathbf{gen}_n$ set contains the substitution element $Sub(a, b)$. The $\mathbf{in}_n$ set is the intersection of the subsitutions in the $\mathbf{out}_a$ nodes that have edges to the current node

$$\mathbf{in}_n = \bigcap_{(a,n)\in\epsilon} \mathbf{out}_a.$$

The $\mathbf{out}_n$ set is the union between $\mathbf{in}_n$ and $\mathbf{gen}_n$ sets except the elements that exists in the $\mathbf{gen}_n$ set. So the $\mathbf{out}_n$ set is the

$$\mathbf{out}_n = (\mathbf{in}_n \cup \mathbf{gen}_n) \setminus \mathbf{kill}_n.$$

The $\mathbf{kill}_n$ set is all substitutions that are no longer relevant after this node. If we have a variable x and something new is assigned to this variable. Then the $\mathbf{kill}_n$ are all the substitutions in the $\mathbf{in}_n$ set than contains this variable.

To determine whether a substitution is redundant graphs are being built. Two different graphs are built each time the $\mathbf{gen}_n$ contains an substitution to check whether the new substitution is redundant. One of the graphs are built from the union of the $\mathbf{gen}_n$ and $\mathbf{in}_n$ and the second graph is built from only the $\mathbf{in}_n$ set. The nodes are the variables in the substitutions in the sets that the graphs are being built from. The edges are directed and shows the relationship between the variables. For example if we have

$$a = b$$

then there is an a node and a b node and an edge that goes from a to b. The first graph is used to check for circularities. If an circularity appeared when the last subsitution was added then the last subsitution was redundant. The second graph is used to check if there is any other way in the graph to reach b from a. An example of this follows in the next section.

3.2 Example Execution

For the following example program

$$b = a$$
$$c = b$$
$$a = c$$

the four sets at each step in the programs execution can be seen in the table 1.

Assign	b = a	c = b	a = c
$\mathbf{in}_n$		$S(b,a)$	$S(b,a),\ S(c,b)$
$\mathbf{gen}_n$	$S(b,a)$	$S(c,b)$	$S(a,c)$
$\mathbf{kill}_n$			$S(b,a)$
$\mathbf{out}_n$	$S(b,a)$	$S(b,a),\ S(c,b)$	$S(c,b),\ S(a,c)$

Table 1: The different sets when the different assignment expressions are executed in the example program above.

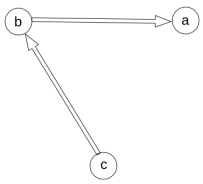
At every step a graph is built from the union of the $\mathbf{in}_n$ and $\mathbf{gen}_n$ set. The graph at different points in the programs excecution can be seen in Figure 2. In Figure 2c we can see that the graph is circular. This circular property did not exist previously. Because of this it follows that the last assignment is redundant.

As previously mentioned it is not always enough to check for circularity. It is also necessary to check for a reachable property. Assume that the gen set is $Sub(x, y)$. Then it is also necessary to check if it is possible to reach node y from x in the graph formed from the $\mathbf{in}_n$ set. Assume that the example program instead is

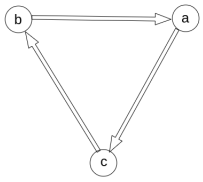
$$b = a$$
$$c = b$$
$$c = a$$

The last graph formed by the union of the $\mathbf{in}_n$ and $\mathbf{gen}_n$ set can be seen in Figure 3a. As can be seen it is not circular. In Figure 3b the graph formed by the $\mathbf{in}_n$ set can be seen. As can be seen it is it is possible to reach c from a in this graph. Therefore the last assignment is redundant.

(a) The graph after assignment $b = a$.

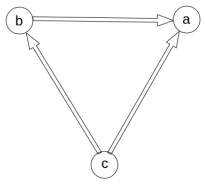


(b) The graph after assignment $c = b$.

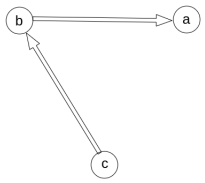


(c) The graph after assignment $a = c$.

Figure 2: The graph that is built on the union of the $\mathbf{in}_n$ and $\mathbf{gen}_n$ sets at different points in the program execution.



(a) The graph built by the union of the $\mathbf{in}_n$ and $\mathbf{gen}_n$ sets after assignment $c = a$.



(b) The graph built from the $\mathbf{in}_n$ set after assignment $c = a$.

Figure 3: Graphs for the second example program.

3.3 Loss of Precision

Sometimes there can be many ingoing edges in the CFG. For example after an if statement there are two ingoing edges, one if the condition is true and one if it isn't. An subsitution is only added to the $\mathbf{in}_n$ set in these cases if the substitution exists in the **out** set for all previous nodes. This causes a loss of precision in the analysis because there might be redundancies that are missed.

3.4 Intraprocedual versus Interprocedual

Intraprocedual program analysis means that the analysis only cares about the content in the specific method. If there is a method call to an other method the control flow does not go into the other method. Interprocedual program analysis on the other hand is not restricted to a single method but follows the flow of inspiration into other methods.

In this project the program analysis is intraprocedual. This can give errors in the analysis if for example an array that a variable points to is changed in an other method.

4 Results

4.1 Redundant Assignments

To measure the how well the tool finds redundant assignments there is a set of testfiles. Futher more the tool is also applied to the `fop0.95` program which is a print formatter written in Java. The `fop0.95` program is useful because it is a real application that contains several hundreds of thousands of lines of java code.

The tool finds all redundant assignments in the testfiles. On the `fop0.95` program the tool flags four errors. However of these four errors two are false positives.

To have something to compare with the `fop0.95` program is also analysed using SonarQube which is a professional software used to find bugs. This software finds one bug that my tool could not.

4.2 Execution Time

To calculate the execution time for the tool the tool did analyze the `fop0.95` program 10 times. The mean time is 58.6 seconds and a 95% confidence interval gives the interval [55.5, 61.7] seconds. The SonarQube took less than 10 seconds to analyze the `fop0.95` program which is clearly outside the confidence interval.

5 Discussion

SonarQube found one bug that my tool did not. The reason for this is that my tool does not support primitive types at the moment. So for example

$$int \quad a;$$
$$a = 9;$$
$$a = 9;$$

will not return a warning when analysed using my tool. In ExtendJ there is an class in the abstract grammar called *variable*. My tool only supports this class.

SonarQube was faster than my tool when analysing the `fop0.95` program. The reason for this is probably because my tool used ArrayLists for storing the sets. When building the graphs from these ArrayLists the find() method is called a lot and this has the complexity O(n).

My tool found two false positives when analysing the `fop0.95` program. The reason for this false positives is that my tool is not interpocedual. So for example if we have the code

$$a = b;$$
$$function\_call();$$
$$a = b;$$

and $function\_call()$; somehow changes the variable a then my tool will not find out that a was changed and flag the second assignment as redundant.

6 Futher Work

There are at least three things than could be done to improve the tool. The program could be rewritten to use HashMap instead of ArrayList. This should make the tool significantly faster. An other improvement would be to include support for primitive types. The third improvement would be to make the analysis interprocedual. The third improvement however would probably require a major rewrite of the tool.

7 Conclusion

In this paper a intraprocedual tool that solves the redundant assignment problem was described. This tool was analysed using two different performance measurements; how good it is at finding errors and the time it takes to do the analysis. This tool is not as good as SonarQube but some suggestions for further work was done that would probably make the tool as good.

References

[1] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[2] Valentina Lenarduzzi, Alberto Sillitti, and Davide Taibi. *A Survey on Code Analysis Tools for Software Maintenance Prediction*, pages 165–175. 09 2018.