**02242 Program Analysis**

**Preliminary Report**

To change: Interval arithmetic processing discussed at lecture. Highly likely it will require a change of table with example program run.

Anusha Sivakumar s124571@student.dtu.dk

Nikita Martynov s124570@student.dtu.dk

Zhen Li s124568@student.dtu.dk

Table of Contents

[1. Program representation 1](#_Toc373658196)

[1.1. Representation of the AST 1](#_Toc373658197)

[1.2. Flow graphs 3](#_Toc373658198)

[1.2.1. Data structure 3](#_Toc373658199)

[1.2.2. Algorithm for transforming 4](#_Toc373658200)

[1.3. Program graphs 6](#_Toc373658201)

[1.3.1. Data structure 6](#_Toc373658202)

[1.3.2. Rules for constructing program graphs 6](#_Toc373658203)

[1.3.3. Implementation 7](#_Toc373658204)

[2. Program Slicing 10](#_Toc373658205)

[2.1. Data Flow Equations 10](#_Toc373658206)

[2.2. Manual computation of solution 12](#_Toc373658207)

[2.3. Algorithms for program slices 13](#_Toc373658208)

[2.3.1. Algorithm for generation of program slices 13](#_Toc373658209)

[2.3.2. Algorithm to compute free variables 15](#_Toc373658210)

[2.3.3. Algorithm to compute reaching definition 15](#_Toc373658211)

[2.3.4. Algorithm to compute killRD and genRD 16](#_Toc373658212)

[2.3.5. Algorithm to compute ud-chains 17](#_Toc373658213)

[2.3.6. Data Structures 17](#_Toc373658214)

[2.4. Program slice using proposed algorithm 18](#_Toc373658215)

[3. Buﬀer overﬂow – detection of signs 20](#_Toc373658216)

[3.1. Detection of signs analysis 20](#_Toc373658217)

[3.1.1. Definition 20](#_Toc373658218)

[3.1.2. Discussion regarding correctness of the analysis 22](#_Toc373658219)

[3.1.3. Proof 23](#_Toc373658220)

[3.2. Algorithm for array bound checking for the lower bounds 24](#_Toc373658221)

[3.2.1. Algorithm for solving detection of sign constraints 24](#_Toc373658222)

[3.2.2. Algorithm for array bound checking for lower bounds 25](#_Toc373658223)

[3.3. Constraints and solutions for the example program 25](#_Toc373658224)

[4. Buffer overflow – interval analysis 28](#_Toc373658225)

[4.1. Interval Analysis 28](#_Toc373658226)

[4.1.1. Definition of Interval Analysis 28](#_Toc373658227)

[4.1.2. Discussion regarding correctness of the analysis 31](#_Toc373658228)

[4.1.3. Proof 33](#_Toc373658229)

[4.1.4. Implementation 34](#_Toc373658230)

[4.2. Algorithm for array bound checking 34](#_Toc373658231)

[4.2.1. Algorithm for solving Interval analysis equations 34](#_Toc373658232)

[4.2.2. Algorithm: Array Bound Checking 35](#_Toc373658233)

[4.3. Constraints and solutions for the example program 36](#_Toc373658234)

[4.4. Discussion on precision of analysis 37](#_Toc373658235)

[5. Security analysis 38](#_Toc373658236)

[5.1. Security analysis 38](#_Toc373658237)

[5.1.1. Definition 38](#_Toc373658238)

[Appendix 40](#_Toc373658239)

[References 45](#_Toc373658240)

# Program representation

Given a piece of program written in while language, the abstract syntax tree (AST) of the program is built when the parser parses it. In our system, the AST is maintained in a tree structure which preserves the original tree structure of an AST obtained directly after parsing. Based on the AST, the flow graph and the program graph are then constructed transforming the data structure from a tree into graphs. Then the graph representations of the program could be used for static program analysis.

## Representation of the AST

Our system deploys a tree to represent the AST. It keeps the primary tree structure of the parsing result. The root of the tree is a node called program. The program node has two sub-trees whose roots are a declaration node and a statement node respectively.

The declaration node consists of declarations of variables in program. If the program has no declaration, a *null* node is recorded. Otherwise, the program could have declarations of variable declaration, array declaration, or sequential declaration.

If the program has only one declaration which could be a variable declaration or an array declaration, the declaration is stored directly in the declaration node making this node to be a leaf node. However if the program has more than one declaration, a reference to a sequential declaration sub-tree is stored in the declaration node.

The sequential declaration sub-tree is a binary tree. The left child of the tree is a leaf node which stores a declaration in the program, and the right child is either a leaf node if only one declaration is left in the program or another declaration sub-tree if more than one declaration is left, so on and so forth. If we perform in-order traversal on the declaration tree and print all the leaf nodes, the declarations are printed following the same order as the order presented in the program. The design of the statement sub-tree is the same as the declaration sub-tree.

We implement the design in Java. Based on the syntax of the while language (), we design classes *Program, Declaration, Statement, BoolExpr* and *ArithExpr* for *P, D, S, b* and *a* correspondingly. The UML diagrams for each of them are shown in Figure 1.1– Figure 1.5. The “has-a” relationship (the program node has a declaration node and a statement node) is represented as a member in a class definition and the “is-a” relationship (a sequential declaration is a declaration) is represented as subclasses.

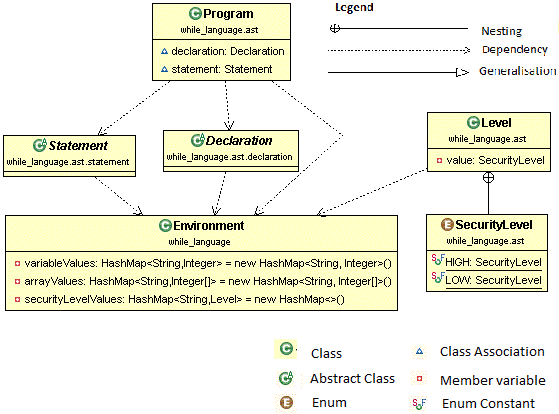


Figure .1 The UML diagram for *Program, Declaration* and *Statement*



Figure .2 The UML diagram for *Declaration*



Figure .3 The UML diagram for *Statement*



Figure .4 The UML diagram for *ArithExpr*



Figure .5 The UML diagram for *BoolExpr*

## Flow graphs

This subsection deals with basic data structures that are employed in our system to store the flow graph and the algorithm for deriving the flow graph from the AST.

### Data structure

The pseudo code below shows the instance field of the data structure for flow graphs.

abstract class FlowGraph {

// instance field

static Vector<Block> blocks; // all the blocks in the whole graph

Vector<int> labels; // the block labels in this graph

Vector<Flow> flow; // the flows in this graph

int init; // init label of this graph

Vector<int> final; // final set of this graph

int ancestorBoolLabel; // used in ud-chain

}

class Flow{

int pri; // the block label which the flow flows from

int next; // the block label which the flow flows to

}

Code 1.1 The *FlowGraph* class for flow graphs

Figure 1.6 which is originally from [1], illustrates an example to use the data structure to represent the flow graph for a piece of code. The flow graph for this whole piece of code contains six *blocks* and they are labeled from 1 to 6. Labels are put in pairs in the *flow* set to represent the flow from one block to another. Finally, the *init* field marks that the starting point is block 1 and *final* set records that the ending set contains one block - that is block 6 in this example.



*blocks:*s

Figure .6 An example for the data structure of the flow graph

### Algorithm for transforming

One thing that needs to be pointed out is that the data structure *FlowGraph* is an abstract class. Classes that extend the abstract class *FlowGraph* are *SimpleFlowGraph, SeqFlowGraph, IfFlowGraph, WhileFlowGraph*. Giving an AST, the statement classes are mapped to *FlowGraph* classes following the transformation rules which are shown in Table 1.1. In practice, the mapping is carried out in a Factory class named *FlowGraphFactory.* When an instance of *Statement* is presented to the Factory, the Factory detects the input *Statement* and returns the desired *FlowGraph.*

As some of the *Statement*s might contain *Statement*s in their field, the mapping could be carried out recursively. In other words, the graph for a piece of program which is represented by an AST is constructed when all the flow graphs for all of the nodes in the AST are constructed.

Table .1 The transformation rules for mapping from Statement classes to *FlowGraph* classes

|  |  |
| --- | --- |
| Classes extending *Statement* | Classes extending *FlowGraph* |
| *AssignmentStatement, SkipStatement,*  *ArrayAssignStatement, ReadArrayStatement,*  *ReadStatement, WriteStatement* | *ElementaryFlowGraph* |
| *SeqStatement* | *SeqFlowGraph* |
| *IfStatement* | *IfFlowGraph* |
| *WhileStatement* | *WhileFlowGraph* |

Table 1.2 and Table 1.3 define rules to fill the instance field in each *FlowGraph* during the mapping from Statement classes to *FlowGraph* classes. A concrete example for *IfFlowGraph* is given in

Appendix B where we demonstrate how the flow graph for *IfStatement* is created recursively (the fields of the flow graph are filled) using the rules in Table 1.2 and Table 1.3.

Table .2 Rules for constructing flow graph (part1)

|  |  |  |  |
| --- | --- | --- | --- |
| ***S*** | ***labels(S)*** | ***init(S)*** | ***final(S)*** |
|  | *{}* |  | *{}* |
|  |
|  |
|  |
|  |
|  |
|  |  |  |  |
|  |  |  |  |
|  |  |  | *{}* |

Table .3 Rules for constructing flow graph (part2)

|  |  |  |
| --- | --- | --- |
| ***S*** | ***flow(S)*** | ***blocks(S)*** |
|  | *∅* |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## Program graphs

This subsection provides the data structure for program graphs as well as the algorithm for generation of the program graph from AST.

### Data structure

The Java-like pseudo code of the program flow data structure is presented in Code 1.2.

Class Edge{

Int qs;

String block;

Int qt;

}

Code 1.2 The *Edge* class for program graphs

The data structure for the program graph is represented by a sequence of edges. Each edge consists of an initial node *qs,* an actual *block* and a final node *qt.* Edges of the program graph are stored in a container which is represented by a vector. This vector contains a sequence of program graph edges.

### Rules for constructing program graphs

The rules for constructing the program graph from the AST are presented in the Table 1.4.

All the statements, except for sequence, conditional and loop statements, simply add corresponding edges containing a block itself and initial and final nodes to the vector of edges. The sequence statement produces two corresponding program graphs for both statements generating a new node between the two statements.

The conditional statement, first, produces two edges for the case where the then Boolean expression holds and for the case where the then Boolean expression does not hold with a fresh final node in each case. Next, the program graphs are generated for then and else branch respectively.

For the case of the while loop statement, firstly an edge is added, for the case where the then Boolean expression holds, producing a fresh node. Next, program graph is generated for the body. Finally, an edge for the case in which the then Boolean expression does not hold is added.

Any appearance of at the right hand side of the Table 1.4 denotes recursion in the algorithm.

Table .4 Rules for transforming the AST to the program graph where *S* is a statement and represents the program graph for statement *S* with initial node *qs*and final node *qt*.

|  |  |
| --- | --- |
| ***S*** |  |
| *[x:=a]l* | *{(qs, [x:=a]l, qt)}* |
| *[skip]l* | *{(qs, [skip]l, qt)}* |
| *S1;S2* | *, where q is a fresh node* |
| *If [b]l then S1 else S2* | *{(qs, [b]l, q1), (qs, [˥b]l, q2)} , where q1 and q2 are fresh nodes* |
| *While [b]l do S* | *{(qs, [b]l, q), (qs, [˥b]l, qt)} , where q is a fresh node* |
| *[A[a1]:=a2]l* | *{(qs, [A[a1]:=a2]l, qt)}* |
| *[read x]l* | *{(qs, [read x]l, qt)}* |
| *[write a]l* | *{(qs, [wrire a]l, qt)}* |
| *[read A[a]]l* | *{(qs, [read A[a]]l, qt)}* |

### Implementation

For implementation the vector of edges is represented in the abstract class *ProgramGraph* and it is shown in Code 1.3. This class is a base class for any statement class and making the vector of edges static provides accessibility for any object of statement classes. Since the graph is built sequentially, each particular program graph will sequentially add its part to the whole program graph.

abstract class ProgramGraph{

Static Vector < edge > edges = new Vector < edge >();

}

Code 1.3 The *ProgramGraph* class pseudo code

Each of the statements from the Table 1.4 is represented in a separate class. The rules of the algorithm for transformation from the AST to the program graph are presented in constructors in the corresponding classes. Constructor of any statement class has three input parameters: a statement from the AST, an initial node for this part of the program graph and a final node.

The statement is used to fill the block field of an edge. *InitialNode* parameter gives the information about starting point for the particular edge. *FinalNode* parameter is used for the special cases then a final node of an edge is not the number incrementing the initial node number by 1. For example, the final node of the statement in the loop body will be the initial node of the edge containing Boolean expression block. Furthermore, in the *else* branch of the conditional statement neither initial nor final node are easy to deduce from the accessible static vector of the full program graph. Thus, *initialNode* and *finalNode* parameters are used to simplify the creation of new nodes. Here the *finalNode* will be the same node as the *finalNode* of the *then* branch and the *initialNode* will be the final node of the edge containing the condition for entering the *else* branch.

The Java-like pseudo code is presented in Code 1.4. The constructor of the class contains the algorithm for creating a program graph for the conditional statement which essentially implements the corresponding line from the Table 1.4 in different notations. The only additional part is that if the vector is empty, then the first edge should be created with *qs* equal 1 and *qt* equal 2, rather than calculating final node value from previous entry of the vector. The function *create* invoked in the constructor is explained in Code 1.5.

class IfProgramGraph extends ProgramGraph {

//constructor

public IfProgramGraph (IfStatement st, int initialNode, int finalNode) {

String boolBlock = st.getBoolExpr().toString();

If (edges.empty()== false)

// add “then” branch

edges.add(new edge(initialNode, boolblock, edges.last().qt +1);

Else

//add “else” branch

{

edges.add( new edge(1, boolblock,2));

edges.add(new edge(

edges.last().qs,“!”+ boolblock,edges.last().qt+1);

}

// graph is created recursively for each branch separate recursion

Int qsElseBranch = edges.last().qt;

ProgramGraphFactory.create(st.getThenStatement (), edges.prelast().qt, finalNode);

finalNode = edges.last().qt;

ProgramGraphFatrory.create(st.getElseStatement(),qsElseBranch, finalNode);

}

}

Code 1.4 The *IfProgramGraph* class pseudo code

The *ProgramGraphFactory* class implements a static function *create*. The *create* function essentially recognizes the input statement and invokes the constructor of the corresponding class which generates the part of the program graph presented by the statement. The pseudo code is presented in Code 1.5.

Code 1.6 shows the starting point for generation of the program graph from the AST. The create function is invoked with the topmost statement of the AST as an input parameter.

The pseudo code for rest of the classes is presented in Appendix C.

class ProgramGraphFactory {

public static void create(Statement st, int initialNode, int finalNode) {

if(st instanceof IfStatement)

new IfProgramGraph(st, initialNode, finalNode);

else if( st instanceof WhileStatement)

new IfProgramGraph(st, initialNode, finalNode);

else … //for all the statements

}

}

Code 1.5 The *ProgramGraphFactory* class pseudo code

void Main(){

ProgramGraphFactory.create(st.getStatement(), 1, 0);

}

Code 1.6 Main function

# Program Slicing

Example program for reaching deﬁnitions analysis:

program

int n;

int x;

int f1;

int f2;

int ans;

n := 20;

f1 := 0;

f2 := 1;

x := 2;

ans = 0;

while x <= n do

ans := f1 + f2;

f1 := f2;

f2 := ans; (\* The point of interest \*)

x := x + 1

end

The program presented at the beginning of this section is used as an example program to demonstrate the eﬀects of program slicing. The program slice for the point of interest which is marked in the example program is given in Code 2.1. Basically, it contains all the statements in the example program except for the assignment statement for *ans*. In the following parts of this section, we will present how this program slice is computed.

n := 20;

f1 := 0;

f2 := 1;

x := 2;

while x <= n do

ans := f1 + f2;

f1 := f2;

f2 := ans; (\* The point of interest \*)

x := x + 1

od

Code 2.1 The program slice for the point of interest in the example program

## Data Flow Equations

The data flow equations makes use of two functions – namely *kill* and *gen*. *kill* function represents information that becomes invalid after execution of a program label. This information needs to be removed from data flow equations. *gen* function represents information generated in a particular label. This information needs to be added to data flow equations. The *gen* and *kill* functions vary for each construct of while language as processing of each construct may result in different information being generated or killed. The following table represents the data flow equations for the constructs of the while language. Table 2.1 is constructed with the assumption that input program has isolated entries.

Table .1 Data Flow Equations for Reaching Definition Analysis

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

All constructs of the while language which modify a variable’s value such as the assignment statement and the read statement have specification of *kill* function mentioned as all previous definitions to a variable. By executing the current assignment/read the information on previous definitions becomes invalid. Their corresponding *gen* specifications have the variable name to which a value is assigned or read into and the current statement’s label, as this is the information about the pair which causes a new definition. For all other language constructs such as Boolean expressions, skip statements and write statements, both *gen* and *kill* functions are defined to be   
 or no value. This is because these non assignment statements do not modify any definitions that reach that point. The reaching definition entry and exit information and of label *l* are defined as below:

## Manual computation of solution

The *kill* and *gen* equations for the example program presented at the start of this chapter are tabulated below:

Table .2 *kill* and *gen* equations for example program

|  |  |  |
| --- | --- | --- |
| Label | (l) | (l) |
| 1 | {(n, ?), (n, 1)} | {(n, 1)} |
| 2 | {(f1, ?), (f1, 2), (f1, 8)} | {(f1, 2)} |
| 3 | {(f2, ?), (f2, 3), (f2, 9)} | {(f2, 3)} |
| 4 | {(x, ?), (x, 4), (x, 10)} | {(x, 4)} |
| 5 | {(ans, ?), (ans, 5), (ans, 7)} | {(ans, 5)} |
| 6 |  |  |
| 7 | {(ans, ?), (ans, 5), (ans, 7)} | {(ans, 7)} |
| 8 | {(f1, ?), (f1, 2), (f1, 8)} | {(f1, 8)} |
| 9 | {(f2, ?), (f2, 3), (f2, 9)} | {(f2, 9)} |
| 10 | {(x, ?), (x, 4), (x, 10)} | {(x, 10)} |

The equations are generated based on the specifications in section 2.1. It can be seen that all statements except statement with label 6, are assignment statements. In each of these statements, all previous information about a variable to which a value is assigned gets killed and current definition is generated as the new value. The statement with label 6 is a Boolean statement and hence no value is modified. Consequently, there is no information that is being killed nor is any information being generated.

The and information for the example program is presented in the table below.

Table .3 and information for the example program

|  |  |
| --- | --- |
|  |  |
| (1) = {(n, ?), (f1, ?), (f2, ?), (x, ?), (ans, ?)}  (2) = RD⦁(1)  (3) = RD⦁(2)  (4) = RD⦁(3)  (5) = RD⦁(4)  (6) = RD⦁(5) U RD⦁(10)  (7) = RD⦁(6)  (8) = RD⦁(7)  (9) = RD⦁(8)  (10) = RD⦁(9) | RD⦁(1) = (1) \ ( (1)) U (1);  RD⦁(2) = (2) \ ( (2)) U (2);  RD⦁(3) = (3) \ ( (3)) U (3);  RD⦁(4) = (4) \ ( (4)) U (4);  RD⦁(5) = (5) \ ( (5)) U (5);  RD⦁(6) = (6);  RD⦁(7) = (7) \ ( (7)) U (7);  RD⦁(8) = (8) \ ( (8)) U (8);  RD⦁(9) = (9) \ ( (9)) U (9);  RD⦁(10) = (10) \ ( (10)) U (10); |

The above sets of equations are based on specification in section 2.1. For the while statement with label 6, the entry and exit information is the same. The value of all the variables are same before and after the execution of the while statement and hence the reaching definitions also remain the same. For all other statements in the example program, the values of variables are getting modified and hence the analysis uses kill and gen functions to remove invalid definition and add latest definition information.

By substituting the actual values in the above set of equations, Table 2.4 gives the solution set. In the table, the solution for the entry information of label 6 is obtained by the union of information from label 5 and label 10. Label 6 has a looping construct and the information flows for the first time from label 5 and for the rest of the times from label 10 which is the last statement within the while loop. All other statements have their entry information same as the exit information of the previous node.

Table .4 Solution set

|  |  |  |
| --- | --- | --- |
| l | (l) | RD⦁(l) |
| 1 | {(n, ?), (f1, ?), (f2, ?), (x, ?), (ans, ?)} | {(n, 1), (f1, ?), (f2, ?), (x, ?), (ans, ?)} |
| 2 | {(n, 1), (f1, ?), (f2, ?), (x, ?), (ans, ?)} | {(n, 1), (f1, 2), (f2, ?), (x, ?), (ans, ?)} |
| 3 | {(n, 1), (f1, 2), (f2, ?), (x, ?), (ans, ?)} | {(n, 1), (f1, 2), (f2, 3), (x, ?), (ans, ?)} |
| 4 | {(n, 1), (f1, 2), (f2, 3), (x, ?), (ans, ?)} | {(n, 1), (f1, 2), (f2, 3), (x, 4), (ans, ?)} |
| 5 | {(n, 1), (f1, 2), (f2, 3), (x, 4), (ans, ?)} | {(n, 1), (f1, 2), (f2, 3), (x, 4), (ans, 5)} |
| 6 | {(n, 1), (f1, 2), (f2, 3), (x, 4), (ans, 5), (f1, 8), (f2, 9), (x, 10), (ans, 7)} | {(n, 1), (f1, 2), (f2, 3), (x, 4), (ans, 5), (f1, 8), (f2, 9), (x, 10), (ans, 7)} |
| 7 | {(n, 1), (f1, 2), (f2, 3), (x, 4), (ans, 5), (f1, 8), (f2, 9), (x, 10), (ans, 7)} | {(n, 1), (f1, 2), (f2, 3), (x, 4), (ans, 7), (f1, 8), (f2, 9), (x, 10)} |
| 8 | {(n, 1), (f1, 2), (f2, 3), (x, 4), (ans, 7), (f1, 8), (f2, 9), (x, 10)} | {(n, 1), (f1, 8), (f2, 3), (x, 4), (ans, 7), (f2, 9), (x, 10)} |
| 9 | {(n, 1), (f1, 8), (f2, 3), (x, 4), (ans, 7), (f2, 9), (x, 10)} | {(n, 1), (f1, 8), (f2, 9), (x, 4), (ans, 7), (x, 10)} |
| 10 | {(n, 1), (f1, 8), (f2, 9), (x, 4), (ans, 7), (x, 10)} | {(n, 1), (f1, 8), (f2, 9), (x, 10), (ans, 7)} |

## Algorithms for program slices

This subsection presents the algorithms for calculating a program slice with respect to an arbitrary point of interest. Five algorithms are involved in the calculation, one for each of the following: the algorithm for generation of program slices, the algorithm to compute free variables, the algorithm to compute reaching definition, the algorithm to compute and , and finally the algorithm to compute ud-chains.

### Algorithm for generation of program slices

This algorithm takes in a program and a point of interest in the program as input and returns all labels or line numbers in the program that affect the point of interest. The statements represented by the labels returned by the algorithm fall under two categories:

1. Statements that directly impact program slice – *assignment*, *read* statement
2. Statements that indirectly impact program slice – *while* loop, *if* statement

The algorithm makes use of flow graph, ud chains and the result of reaching definition analysis and a worklist *W* to compute the program slice with respect to the point of interest.

INPUT: program, point of interest in program

OUTPUT: List of labels that affect point of interest

ALGORITHM:

ReachingDefinitionAnalysis()

BlockAnalysis()

If pointOfInterest := null

then return null;

else

W := cons(pointOfInterest ,W );

While do

currentLineOfInterest := head(W);

W := tail(W);

programSlice := cons(currentLineOfInterest,programSlice);

for each free variable in the currentLineOfInterest with variablePosition not left

for each label l in udchain(freevariable, currentLineOfInterest)

if l not in programSlice

cons(l, W);

if ( currentLineOfInterest has a Boolean ancestor and labelOfbooleanAncestor not in programSlice)

cons(labelOfBooleanAncestor, W);

return programSlice;

Code 2.2 Algorithm for generation of program slices

This algorithm makes use of functions *cons(), head().* The *cons()* function takes in a variable and a list as argument and appends the value of the variable as the first element in the list. The *head()* function returns the first element in the list and the *tail*()functionreturns a smaller list without the first element.

The algorithm works as follows: The current point of interest is added to the program slice. For each free variable in the point of interest, the labels in which the free variables were most recently modified are obtained from ud chain and added to a worklist. If the current label has any Boolean ancestor and if this ancestor is not present in program slice, i.e. it has not already been processed, it is added to the worklist. If a statement is present inside a *while* loop or an *if* statement, then it is referred to as having a Boolean ancestor. During the construction of flow graph, ancestral details are modeled and each block will have a field that holds the label of the most recent ancestor. By checking for Boolean ancestor and including it in the program slice, all statements that indirectly impact the point of interest are included in the slice. As long as the worklist is not empty, the above process is repeated by choosing the first element in the worklist as current point of interest.

### Algorithm to compute free variables

BlockAnalysis()

Input: FlowGraph

Output: Variables in each line of the flow graph along with their position

For each block b in the flow graph

Get all variables used in the block

For each occurance of variable in the block

If block is an assignment statement

Get position of variable in statement as left or right

Else

Set variablePosition to null

cons((variableInstatement,linenumber,variablePosition), Variables);

return Variables;

Code 2.3 Algorithm to compute free variables

This algorithm finds the free variables in the program and their position with respect to assignment operator ie. left and right or none when the statement is not an assignment statement. The algorithm examines each block in the flow graph. The variables in the block are added to output list. If a variable is part of an assignment statement it records the variable and position of the variable with respect to the assignment operator i.e. left or right. The *right*() function in the above algorithm returns variables to the right of assignment operator ‘:=’. These are the free variables in an assignment statement.

### Algorithm to compute reaching definition

ReachingDefinitionAnalysis()

Input: A set of reaching definition equations

Output: The least solution to the equations

;

;

;

Code 2.4 Algorithm to compute reaching definition

The first step in constructing the program slices is to solve a set of equations using reaching definition analysis. This step is carried out as a worklist algorithm and has two phases an initialization phase and an iteration phase. In the initialization phase, a worklist and a set of equations are initialized. All flows of the flow graph of the form are added to a worklist. Here, information flows from label l to label . The equations are initialized by assigning extremal value which is or unknown for all free variables in the program if the label is an initial label.The equations are assigned the least element for all other labels.

In the second phase, the following steps are repeated as long there remains an element in the worklist. The first element of the worklist is considered for processing, represented by *head*(W) and of the form . The smaller worklist with the first element removed is placed in *tail*(W). The transition function takes the incoming value of reaching definition i.e pairs of variables and the labels at which the variables were modified most recently. If the variable is modified in current statement, all pairs containing that variable are killed. A new pair of modified variable’s name and label of current statement is added to the reaching definition equation. If the result of transition function is not a subset or equals to that of then we union the existing value of with the value from transfer function on . This is to ensure that any change in variable definition - label pairs at a particular label flow to label connected to it. This is because the output of is the input to Each label that has the label as a starting point is added to the worklist, so that the changes are reflected in all equations of labels where information flows into.

### Algorithm to compute killRD and genRD

Input: Block Analysis results, blocks in program

Output: and for each block in input program

For each block in input program

If is an assignment statement

then get variable whose position in is ‘left’

;

Else if is a read statement

then get free variable in

;

Else

Code 2.5 Algorithm to compute killRD and genRD

The algorithm generates equations for (what is newly modified in statement) and (what becomes invalid after this statement and gets removed) for each block in an input program. If the statement is read or assignment statement then value of variable is getting modified in the statement and equations are generated by removing all (variable name, label) pairs for modified variable and appending a (variable name, current label) pair to existing definition. For any other statement a low value is assigned.

### Algorithm to compute ud-chains

udchain(variable,label)

Input: Block Analysis results, currentLineOfInterest,RDentry

Output: udChain for currentLineOfInterest

For each variable in Variables

If( variable present in currentLineOfInterest and variablePosition is not ‘left’ )

For each (variable,lineNumber) in RDentry(currentLineOfInterest)

cons((lineNumber),udChain);

Code 2.6 Algorithm to compute ud-chains

Ud-chains or use definition chains return a list which contains the assignment statements related to use of a variable in a particular line. The algorithm to compute ud-chains takes in a variable and a label. If the free variable is not on the right hand side of assignment statement in a particular line, the label of last statement where the variable was modified is obtained from *RDentry*equation of the line and added to the output list.

### Data Structures

The algorithm to compute program slice presented above requires the creation of following classes:

The *VariableDefinition* is the smallest unit and is used to store a variable name and label pair.

Class VariableDefinition{

String variable;

Int label;

}

Code 2.7 Class *VariableDefinition*

The *Freevariable* class models the output of block analysis which analyses each block to determine the variable, position and label in which it is present for all free variables in the program.

Class FreeVariable {

String variableName;

String variablePosition;

Int label;

}

Code 2.8 Class *FreeVariable*

The *BlockAnlaysis* class is used to store information about free variables.

Class BlockAnalysis{

//Input

Vector<Block> statements;

//Output

Vector< FreeVariable > Variables;

}

Code 2.9 Class *BlockAnalysis*

The *ReachingDefinitionFactory* class is used to generate killRD and genRD equations for all blocks in a statement based on the type of the statement. This is designed to be a Factory class as the output differs based on type of statement.

Class ReachingDefinitionFactory {

//Input

Block b;

Vector< FreeVariable > Variables;

//Output

Vector <VariableDefinition> killRD;

Vector <VariableDefinition> genRD;

getKillRD(Block b);

getGenRD(Block b);

}

Code 2.10 Class *ReachingDefinitionFactory*

The solution to RDEntry and RDExit equations for a single statement is stored in ReachingDefinition class.

Class ReachingDefinition{

//Input

Vector <VariableDefinition> killRD;

Vector <VariableDefinition> genRD;

//Outputc

Vector <VariableDefinition> rdEntry;

Vector <VariableDefinition> rdExit;

}

Code 2.11 Class *ReachingDefinition*

The vector of labels which affect the current point of interest is stored in a *ProgramSlice* class. This class has input variables for flow graph, point of interest and a vector with solutions of reaching definitions equations for all statements in input program.

Class ProgramSlice{

//Input

int pointOfInterest;

FlowGraph flowGraph;

Vector < ReachingDefinition > rdResult;

//Output

Vector<int> sliceLabels;

Vector<int> getudChain(String,int);

}

Code 2.12 Class *ProgramSlice*

## Program slice using proposed algorithm

The algorithm to calculate the program slice begins with performing reaching definitions analysis and obtaining the reaching definitions entry and exit information which is same as the one mentioned in Section 2.2. Block analysis is performed as a next step where free variables in each line of graph and their position are identified. A detailed solution of how to compute the results of block analysis is shown in

Appendix **D**.

Finally, the code slice shown in is returned. The code slice is found to be the same as obtained from manual computation at the beginning of this chapter.

n := 20;

f1 := 0;

f2 := 1;

x := 2;

while x <= n do

ans := f1 + f2;

f1 := f2;

f2 := ans; (\* The point of interest \*)

x := x + 1

od

Code 2.13 The program slice for the point of interest in the example program

# Buﬀer overﬂow – detection of signs

Example program for buffer overflow:

program

int a[5];

int b[5];

int n;

n := -1;

while (n<7) do

a[n] := 1; (\* inadmissible array reference \*)

if(n >= 0 & n < 5)

then b[n] := 1; (\* admissible array reference \*)

else skip;

fi

n := n + 1;

od

end

The program presented above is the example program for detection of signs. In the while loop, during the first and the last iteration, the index of array *a* goes out of the bounds of the array, however the index of array *b* is always admissible.

## Detection of signs analysis

This subsection addresses the definition of the detection of signs analysis for the project language followed by a proof that the analysis is an instance of a monotone framework.

### Definition

The detection of sign analysis is defined as

where *L* is a complete lattice, is a set of transfer functions, *F* is a finite flow , *E* is a finite set of extremal labels, is an extremal value and is a mapping from labels to transfer functions.

#### Lattice L

The complete lattice *L* is defined as

where .

#### Mapping

The mapping of labels to transfer functions is constructed as

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

#### 

The which determines the signs of expressions is given by

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

where which could be and is specified by

and which could be the unary operand only is specified by

Finally, the which could be and /, and which could only be are defined in the Table 3.1 and Table 3.2.

Table .1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| + | - | 0 | + | - | - | 0 | + |
| - | {-} | {-} | {-,0,+} | - | {-,0,+} | {-} | {-} |
| 0 | {-} | {0} | {+} | 0 | {+} | {0} | {-} |
| + | {-,0,+} | {+} | {+} | + | {+} | {+} | {-,0,+} |
| \* | {-} | {0} | {+} | / | {-} | {0} | {+} |
| - | {+} | {0} | {-} | - | {+} |  | {-} |
| 0 | {0} | {0} | {0} | 0 | {0} |  | {0} |
| + | {-} | {0} | {+} | + | {-} |  | {+} |

Table .2

|  |  |  |  |
| --- | --- | --- | --- |
|  | - | 0 | + |
| - | {+} | {0} | {-} |

#### and

The is a set of “small” abstract states that makes up and it is given by

is given by

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

Finally, the and are defined with the tables below respectively.

Table .3

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **<** | - | 0 | + | **>** | - | 0 | + |
| - | {tt,ff} | {tt} | {tt} | - | {tt,ff} | {ff} | {ff} |
| 0 | {ff} | {ff} | {tt} | 0 | {tt} | {ff} | {ff} |
| + | {ff} | {ff} | {tt,ff} | + | {tt} | {tt} | {tt,ff} |
| **<=** | - | 0 | + | **>=** | - | 0 | + |
| - | {tt,ff} | {tt} | {tt} | - | {tt,ff} | {ff} | {ff} |
| 0 | {ff} | {tt} | {tt} | 0 | {tt} | {tt} | {ff} |
| + | {ff} | {ff} | {tt,ff} | + | {tt} | {tt} | {tt,ff} |
| **=** | - | 0 | + | **!=** | - | 0 | + |
| - | {tt,ff} | {ff} | {ff} | - | {tt,ff} | {tt} | {tt} |
| 0 | {ff} | {tt} | {ff} | 0 | {tt} | {ff} | {tt} |
| + | {ff} | {ff} | {tt,ff} | + | {tt} | {tt} | {tt,ff} |

Table .4

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| & | true | false | **|** | true | false |
| true | {tt} | {ff} | True | {tt} | { tt } |
| false | {ff} | {tt} | false | { tt } | {ff} |

### Discussion regarding correctness of the analysis

The lattice is defined as , since we are mapping all the variables together with arrays to . In our system, we do not distinguish the elements in an array but use the name of it as a symbol for the whole array representation.

For each statement in the language, special transfer function is designed, except for while statements and if statements where only Boolean expression is considered for constructing transfer function.

Furthermore, for each arithmetic expression we defined a way of deriving the sign. The arithmetic operations are defined in a consistent way to the conventional arithmetic. For example, the addition of negative numbers obviously results in negative number, while result of positive number and negative number addition cannot be stated unambiguously without the knowledge of the particular values. Hence, we consider that the addition of a positive number and a negative number may result in any of negative, positive or zero value.

The relational operations are defined using conventional rules as well. Similarly, in ambiguous cases where result cannot be specified without knowledge of particular values we assume that the result might be either true or false. For example, the result of comparison of two negative numbers is ambiguous and depends on the exact values of these numbers. On the other hand, for example, it is unambiguous that a negative number will always be less than any positive number. The Boolean operations are also defined using well known logic and both AND and OR operations are unambiguous.

Transfer function for *variable assignment* *statement* changes set of signs of the variable to the set of signs derived from the solution of the arithmetic expression on a right hand side of the assignment as described above.

Considering that an array represents sequence of variables and we do not distinguish between specific elements, *array assignment* *statement* transfer function designed in a way that helps to keep possible signs of all array elements. Thus, for this reason, the set of signs derived from the solution of a right hand side arithmetic expression is united with the set of signs mapped to the array before the execution of the transfer function.

*Read variable* *statement* and *read array statement* transfer functions assign the complete set of signs *{-,0,+}* to a variable or array, respectively. Since user’s input cannot be predicted during the static analysis.

*Skip* *statement* and *write* *statement* transfer functions do not influence sets of signs, and therefore, no signs are updated.

*Boolean statement* transfer function is designed to make a benefit from program graph in analysis and make the result more precise, we define to work with a single sign at a time for each variable in statement. More restrictive sets of signs are derived from solution of a Boolean statement, since only signs satisfying the condition of the statement are left in the sets of signs for each variable presented in the statement. As in the program graph, the Boolean condition of the false branch is defined as the logic not of the Boolean condition of the *then* branch, we only need to deal with the signs satisfying the Boolean condition in any case.

Considering that our design complies with conventional rules and all of the transfer functions are designed following over approximation but not under approximation, we argue that the presented detection of sign analysis is correct.

### Proof

To prove that our model for detection of sign analysis is an instance of monotone framework, we need to prove that the functions are monotone. However, the functions are monotone if the function determining the sign of expressions and the transfer function deciding the sign of arithmetic expressions in Boolean expressions are monotone.

1. First, we check that the functions are monotone.

For the case , suppose we have

It is easy to verify that for all the possible values of , there are

For the case , it is the same as the case .

Thus, we have proved that the functions are monotone.

2. Then we need to prove that is monotone.

Suppose we have , then for any variable x, we have and from there, it derives . By extracting all the elements satisfying in the set and the set respectively, we prove . Thus, we have proved that is monotone.

From the above proofs, it follows that the detection of sign analysis for the project language is an instance of monotone framework.

## Algorithm for array bound checking for the lower bounds

This algorithm takes results from the program graph , the free variables in program under consideration and the transition function table as defined in section 3.2 that specifies how the signs of variables changes for each construct of the while language. There are two main steps in array bound checking, the first is to construct a set of constraints for detection of sign analysis and solve them and the next is to use the solutions to determine the statements in which the violation of bounds occurs.

### Algorithm for solving detection of sign constraints

Input: Program Graph, Transition function table, Free variables

Output: Solutions to Detection of Sign

For all in do

For all q in do

If then DS(q) :={(x, )|x FV() }

else DS(q) :=

While W≠nil do

(q, ,q’):= head(W);W=tail(W);

If ( DS(q)) DS(q’), x FV()

then DS(q’) = DS(q’) ( DS(q)), x FV()

for all (q’, , q’’) in do

W:= cons((q’,, q’’),W);

Code 3.1 Algorithm for solving detection of sign constraints

The first step is construction of a set of constraints one for each node in the program graph and then solving them to get a set of possible signs for each free variable at that particular node. This step is carried out as a worklist algorithm and has two phases - an initialization phase and an iteration phase. In the initialization phase, the worklist and the set of constraints are initialized. All edges of the program graph of the form are added to a worklist. Here, *q* is the starting node, the terminal node and the block in the program that causes this state change. The constraints are initialized to extremal value which is for initial node and to the least element for all other nodes.

In the second phase, the following steps are repeated as long there remains an element in the worklist. The first element of the worklist is considered for processing, represented by *head*(*W*) and of the form *(q, , q’).* The smaller worklist with the first element removed is placed in *tail(W*). If the result of transition function on constraints of *q* namely *DS(q)* is not a subset or equals to that of *DS(q’)* then we union the existing value of *DS(q’)* with the value from transfer function on *DS(q).* This is to ensure that all values of variables in a node flow to all nodes connected to it. Every edge that has the modified node as a starting point is added to the worklist, so that the changes are reflected in to all nodes where the information flows.

### Algorithm for array bound checking for lower bounds

Input: Program Graph, Solutions of Detection of Sign analysis for the program

Output: VL – Set of blocks in which violation of lower bound occurs

For all (q,, q’) in do

If is a statement that includes an array as one of the free variables

If DS(q’) for index of the array contains {-}

then cons((),VL)

Code 3.2 Algorithm for array bound checking for lower bounds

The second stage of the algorithm for bound checking takes in all edges in the program graph and solutions of the detection of sign analysis as input. The following steps are repeated for each edge of the program graph. If the block has an array element as one of the free variable and if the solution of the target node contains {-} for the index variable of the array, then this means a negative number is used as the bound and this indicates a violation. Thus the lower bound violation is detected and the corresponding block is added to set VL which is the output and contains all the blocks where boundary violation occurs.

## Constraints and solutions for the example program

This subsection discusses the solution to the example program of buffer overflow by hand using the speciﬁcation of the analysis addressed in the previous subsections. The program graph is adopted for the detection of sign analysis as the program graph makes use of results of Boolean tests which gives rise to more precise analysis results in different branches. The program graph for the program presented at the beginning of this section is shown in Figure 3.1. The places where the indexes of the arrays are to be visited are *Node 3* and *Node 5*.



Figure .1 Program Graph of example program

Constraints for the aforementioned program are presented below:

*DS (1) [ {0}, {0}, {0}]*

*DS (2) fn:=-1 (DS (1))*

*DS (3) fn<7 (DS (2))*

*DS (4) fa[n]:=1 (DS (3))*

*DS (5) fn>=0&n<5 (DS (4))*

*DS (6) f˥(n>=0&n<5) (DS (4))*

*DS (7) fb[n]:=1 (DS (5))*

*DS (7) fskip (DS (6))*

*DS (2) fn:=n+1(DS (7))*

*DS (8) f˥(n<7) (DS (2))*

where *DS(1)* is an assignment constraint which assigns the default values of all the variables as 0.

The solutions for the constraints involve only one variable *n* since we are concerned about the buffer overflow analysis and *n* is the only variable in the program used as array index. Table 3.5 shows constraints solutions. Based on the results in the table, we can say that the array index for *a* might be inadmissible as the sign of *n* coming from *Node 3* could be negative. But the index for *b* will never be negative when the program goes from *Node 5*.

Table 3.5 Solutions to constraints for example program

|  |  |  |  |
| --- | --- | --- | --- |
| *DS(i)* | *n* | *A* | *B* |
| 1 | {0} | {0} | {0} |
| 2 | {-,0,+} | {0,+} | {0,+} |
| 3 | {-,0+} | {0,+} | {0,+} |
| 4 | {-,0,+} | {0,+} | {0,+} |
| 5 | {0,+} | {0,+} | {0,+} |
| 6 | {-,+} | {0,+} | {0,+} |
| 7 | {-,0,+} | {0,+} | {0,+} |
| 8 | {+} | {0,+} | {0,+} |

## Implementation

A single sign is represented with a Java enumerator *Sign* with 3 possible values corresponding to *minus, zero* and *plus*. Since each variable can have set of signs, the *Signs* class represents this set and implements all needed functionality to conveniently add and check which signs are set or not. Each variable or array represented in the program as a tuple of a string name and instance of the *Signs* class.

The *ArithDS* class implements all arithmetic operations of the While language. The *ArithDS* class constructor takes as input an arithmetic expression and current state of program variables, in this case, their signs. Then expression is analyzed and corresponding function is called to solve expression and detect signs. If there are nested expressions they are solved recursively. The *BoolDS* class implements all Boolean and relational operations of the While language. The structure of logic is the same but the goal is to reduce sets of signs of the variables or arrays in the Boolean expression. The *DSTranfFuncs* class is a class implementing all transfer functions as described in Section 3.1.1. Each input statement is analyzed and corresponding transfer function is called. Transfer functions make use of functionality of the *ArithDS* and *BoolDS* class if an arithmetic or boolean expression is needed to be solved.

The *DSWorklist* class contains implementation of the Maximal Fixed Point (MFP) [2] worklist algorithm as described in Section 3.2.1 and implementation of the algorithm for detection of array lower bound violation described in Section 3.2.2.

In current implementation if division by zero is detected, an exception is thrown and program is terminated without further analysis during this run. Otherwise, all array lower bound violations are reported together in the end of the analysis.

## Limitations and Discussion regarding improvements

A complete implementation of analyses presented in Section 3.1.1 requires more and more efforts with increasing complexity of expressions, thus the implementation is limited to certain extent which is explained in this section. Furthermore, improvements to increase precision of the detection of signs analysis are presented as well.

Considering arithmetic operations, if both operands are represented with the same variable, both of them should have the same sign at a time. For example, if *a{0,-,+},* then for *a\*a* only 3 cases are considered (*{0}\*{0}, {-}\*{-},{+}\*{+}*) instead of all possible combinations which result in *{0,+}*. However, if multiplication expression operands will be nested expressions, even though equal the program currently program cannot deduce it and consider limited number of cases. For example, the expression *((a-7)\*(a-7))* is calculated considering all 9 possible combinations and results in *{-,0,+}*. Thus, precision is lost.

An improvement would be to derive from nested arithmetic operations that they are identical as it is done for variables or arrays. Since this improvement considerably increases complexity of implementation, it is not provided.

Considering relational operations for conditional tests, sets of possible signs are reduced according to the Table 3.3. Only signs satisfying condition are left and others are reduced. In this case, the implementation is limited to simple cases. In particular, the signs sets can be reduced if at least one side of relational operation is represented as a variable or array, e.g. with the expression *(a < (b+1))* only signs of the variable *a* can be reduced. If both side expressions are arithmetic expressions and if there is a case then condition holds, then no signs are reduced.

Considering Boolean operations *&* and *|*, if any side expression contains relational operation which has an arithmetic expression inside, the program does not reduce signs. This restriction is applied since variables in arithmetic expression are not tracked and it can contain a variable for which signs are reduced in other part side of Boolean operation. For example, in case of *((a[2]<b) & (a[1]<0))* condition, if there is the case that this condition holds, the signs of array *a* and variable *b* can be modified to more restrictive ones. However, if any of expressions had arithmetic expression, it would not be possible.

An improvement, in these cases, would be to implement more complex logic to track the signs of the particular variables while calculating the arithmetic expression, and then, choose the ones satisfying relational operation condition.

Another limitation is applied for the algorithm detecting violation of the array low boundary. Violation is detected if an array index contains a number, an array or a variable, since the variable or array are looked up in the corresponding entry in the solutions table and sign of number is easily deducible. The analysis of the index expression would require complex changes, since at this point all information about types is lost and the program processes strings.

In this case, an improvement would be to solve arithmetic expressions in the index field of array.

Moreover, precision could be added by considering the values of numbers, at least, when arithmetic operations are performed with 1. For example, currently *{-} +1= {-,0,+}*, however, applying the proposed improvement the result would be *{-,0}*.

An improvement to efficiency of the analysis would be to use more advanced worklist algorithm discussed at the lecture [3], e.g. based on depth-first spanning forest and reversed postorder.

## Benchmarking

This section presents results of benchmarking.

### Benchmark 1

The example program below is selected to show that our program finds violation correctly and assignment statement *array[a] := a\*a* shows that *array* is assigned *{0, +}* even though *a* has *{-,0,+}.* This shows the functionality discussed in Section 3.5.

(\* Andrius Andrijauskas (s121060@student.dtu.dk), Lars Bonnicshen (lfbo@dtu.dk) \*)

program

int array[10];

int a;

skip; (\* label 0, a in {0}, array in {0} \*)

a := -1; (\* label 1, a in {-}, array in {0}\*)

while (a <= 10) do (\* label 2, a in {-,0,+}, array in {0, +} \*)

array[a]:= a \* a; (\* label 3, a in {-,0,+}, array in {0, +} a can be -1 and 10 \*)

a := a+1; (\* label 4, a in {-,0,+}, array in {0, +} \*)

od

skip; (\* label 5, a in {+}, array in {0, +} \*)

end

Code 3.3 Benchmark 1program

The output of our detection of signs analysis program presented below. The array low boundary violation is detected correctly.

Program graph:

(1,skip;,2), (2,a := -1;,3), (3,a<=10,4), (4,array[a] := a\*a;,5), (5,a := a+1;,3), (3,!a<=10,6), (6,skip;,7)

Detection of signs solutions table 17:

1: a={0} array={0}

2: a={0} array={0}

3: a={-,0,+} array={0,+}

4: a={-,0,+} array={0,+}

5: a={-,0,+} array={0,+}

6: a={+} array={0,+}

7: a={+} array={0,+}

Low boundary violations for array indexing:

(4,array[a] := a\*a;,5)

### Benchmark 2

The example program below is selected to show that our analysis of Boolean conditions works correctly and adds precision to the analysis.

(\*Kamran Manzoor s124569@student.dtu.dk

Noel Vang s082961@student.dtu.dk\*)

program

int buff[5];

int index;

int wlb;

int wub;

index := 4; (\*label 1\*)

wlb := -5; (\*label 2\*)

wub := 7; (\*label 3\*)

while index >= wlb do (\*label 4\*)

buff[index] := 10; (\*label 5 - low boundary violation \*)

index := index - 1; (\*label 6\*)

od

index := 0; (\*label 7\*)

while index <= wub do (\*label 8\*)

write buff[index]; (\*label 9\*)

index := index+1; (\*label 10\*)

od

end

Code 3.4 Benchmark 2 program

The detected violation of array lower bound repeats the result provided by the authors of the benchmark. However, authors initialized variables with *{0,-,+}* and it influenced their calculations, therefore their solutions table is not provided. The results of our program are presented below.

Program graph:

(1,index := 4;,2), (2,wlb := -5;,3), (3,wub := 7;,4), (4,index>=wlb,5), (5,buff[index] := 10;,6), (6,index := index-1;,4), (4,!index>=wlb,7), (7,index := 0;,8), (8,index<=wub,9), (9,write buff[index];,10), (10,index := index+1;,8), (8,!index<=wub,11)

Detection of signs solutions table 33:

1: index={0} wub={0} buff={0} wlb={0}

2: index={+} wub={0} buff={0} wlb={0}

3: index={+} wub={0} buff={0} wlb={-}

4: index={-,0,+} wub={+} buff={0,+} wlb={-}

5: index={-,0,+} wub={+} buff={0,+} wlb={-}

6: index={-,0,+} wub={+} buff={0,+} wlb={-}

7: index={-} wub={+} buff={0,+} wlb={-}

8: index={0,+} wub={+} buff={0,+} wlb={-}

9: index={0,+} wub={+} buff={0,+} wlb={-}

10: index={0,+} wub={+} buff={0,+} wlb={-}

11: index={+} wub={+} buff={0,+} wlb={-}

Low boundary violations for array indexing:

(5,buff[index] := 10;,6)

### Benchmark 3

The example program below is selected to demonstrate two essential limitations of our detection implementation analysis.

(\*Tomasz Cezary Maciazek s111954@student.dtu.dk\*)

(\*Arthur Hugo Maxime Desjardins s131187@student.dtu.dk\*)

program

int x;

int y;

int A[5];

x := 3; (\*label 1 x in {0}, y in {0}, A in {0}\*)

y := 2; (\*label 2 x in {+}, y in {0}, A in {0}\*)

while x > 0 do (\*label 3 x in {0,+}, y in {-,0,+}, A in {0,+}\*)

y := y - 1; (\*label 4 x in {+}, y in {-,0,+}, A in {0,+}\*)

x := x - 1; (\*label 5 x in {+}, y in {-,0,+}, A in {0,+}\*)

A[y] := x; (\*label 6 x in {0,+}, y in {-,0,+}, A in {0,+}\*)

A[x] := x + 2; (\*label 7 x in {0,+}, y in {-,0,+}, A in {0,+}\*)

A[5-x] := 2; (\*label 8 x in {0,+}, y in {-,0,+}, A in {0,+}\*)

od

skip; (\*label 9 x in {0}, y in {-,0,+}, A in {0,+}\*)

end

Code 3.5 Benchmark 3 program

The results documented by the authors of the benchmark program shows that violation is detected at label 6 *A[y] := x* and 8 *A[5-x] := 2*. Our implementation is not that precise, hence our system detects violations in edges *(6,A[y] := x;,7)* and *(7,A[x] := x+2;,8)*. Firstly, as discussed in Section 3.5, the implementation does not deal with values of numbers, that is why violation is detected in edge *(7,A[x] := x+2;,8)*. Secondly, as discussed in Section 3.5, arithmetic expressions are not handled at the point of search for array low boundary violations. The detailed output of our system is presented below.

Program graph:

(1,x := 3;,2), (2,y := 2;,3), (3,x>0,4), (4,y := y-1;,5), (5,x := x-1;,6), (6,A[y] := x;,7), (7,A[x] := x+2;,8), (8,A[5-x] := 2;,3), (3,!x>0,9), (9,skip;,10)

Detection of signs solutions table 24:

1: A={0} y={0} x={0}

2: A={0} y={0} x={+}

3: A={-,0,+} y={-,0,+} x={-,0,+}

4: A={-,0,+} y={-,0,+} x={+}

5: A={-,0,+} y={-,0,+} x={+}

6: A={-,0,+} y={-,0,+} x={-,0,+}

7: A={-,0,+} y={-,0,+} x={-,0,+}

8: A={-,0,+} y={-,0,+} x={-,0,+}

9: A={-,0,+} y={-,0,+} x={-,0}

10: A={-,0,+} y={-,0,+} x={-,0}

Low boundary violations for array indexing:

(6,A[y] := x;,7), (7,A[x] := x+2;,8)

# Buffer overflow – interval analysis

This section contains the buffer overflow detection using the interval analysis.

## Interval Analysis

This subsection presents the definition of the interval analysis for the project language followed by a proof that the definition is an instance of a monotone framework.

### Definition of Interval Analysis

The representation of interval analysis as a monotone framework is given by

where *L* is a complete lattice, is a set of transfer functions, *F* is a finite edges from a program graph, *E* is a finite set of extremal labels, is an extremal value [0, 0] and is a mapping from labels to transfer functions.

#### Lattice L

The complete lattice *L* is defined as

Where

and .

The ordering is given by () if and only if , where

and holds for all

For the mapping from to *Interval*, it is obvious that it is a complete lattice as 1) the possible values for and in the program is infinite and is also infinite by definition; 2) for () and (), the relation always holds as always contains all the elements in at least. Thus the lattice *L* is a complete lattice.

#### Mapping

The mapping of labels to transfer functions is constructed as

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

#### 

The which determines the interval for the value of an expression

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

where is an abstract operation on intervals which could be and . They are specified by

As well as

and which could be the unary operand only is specified by

Similarly, we have

Finally the function is defined by

#### and

The is a set of “small” abstract states that makes up and it is given by

is given by

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

Finally, the and are defined with the tables below respectively. One thing need to be point out is that and .

Table .1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| & | true | false | | | true | false |
| True | {tt} | {ff} | true | {tt} | {tt} |
| False | {ff} | {ff} | false | {tt} | {ff} |

### Discussion regarding correctness of the analysis

This subsection presents the correctness of the analysis presented above.

To achieve interval analysis, we map variables (and arrays) to intervals. Thus the lattice is defined as in which the elements in an array are treated as a whole. The interval is defined as a pair. The pair marks the possible values that a variable could be at a point during the execution of program. As the complete lattice definition requires that the lattice should not be infinite. We bound the values of a variable to be a value between *min* and *max*. The numbers that are greater than *max* are recorded as [] and the numbers that are less than *min* are coded as []. By using this definition, the following interval representations are illegal in our model: , .

For each statement in the language, a specific transfer function is designed, except for while statements and if statements in which only Boolean expression is considered. The transfer functions describe how intervals are mapped to variables.

*Variable assignment* For each assignment, the interval of the variable on the left side of the equation is derived by calculating the interval for the arithmetic expression on the right side. The interval for arithmetic expressions is computed using corresponding arithmetic expressions.

Take plus operation as an example. When we plus two intervals, say [, ] and [, ], the lower boundary and upper boundary are computed separately. When computing each boundary, we first consider the special cases where the plus operators contain infinity values (or ). If the plus operator comprises infinity values or the plus result of and goes beyond the valid interval range ([*min*, *max*]), then the result is set as the same infinity symbol as the original or computed one. Otherwise, it is the normal case, where the plus result lies in the valid interval range. For this case, the direct mathematical calculation result is returned.

However following the rules defined above, it might result in introducing illegal intervals and into our model. Suppose we have two intervals [-1,-1] and [-1, -1] and the *min* = -1. Then by applying the computation rules given for plus, we get an illegal interval []. To compensate this, we introduce the *chop* function.

The *chop* function takes an interval as input and chops it into one of the legal intervals in the model. The legal intervals are one of the intervals satisfying and .

*Array assignment* As an array is a series of variables, when mapping the intervals for the array assignment, we first get the existing interval of the array variable and then map the array interval as the union of the existing interval and the new interval computed by the arithmetical expression on the right of the assignment. By doing this, we loss some precise in our model but makes sure all the possible values for array variables are recorded.

*Skip and write statement* For the skip and write statements, no interval is updated due to the effect of these statement is doing nothing or showing results to users.

*Read statements* For the read statement for a variable or an array, as we have no pre-knowledge what value the user would provide, we simply assign the Interval of the variable or array involved as all the possible number, i.e. [].

*Boolean statement* As we want to benefit from using program graph in analyzing and make the result more precise, we define to split an interval for true and false Boolean branches. As on the program graph, the Boolean condition of the false branch is defined as the logic not of the Boolean condition of the true branch, we only need to deal with the intervals satisfying the Boolean condition.

The correctness of the relational operations mainly relies on intersection of both operand intervals. For example, if the interval 1 upper boundary is less than the interval 2 low boundary than obviously the interval 1 is less than the interval 2. Similarly, if the upper boundary of the interval 2 is less than the low boundary of the interval 1, then the interval 1 is greater than the interval 2. However, if these intervals have intersection they can be ether equal or one can be greater than other. Respectively, the intervals without intersection cannot be equal, while they are certainly equal if all the boundaries of both intervals are equal, for example the interval [1,1] equal to the interval [1,1]. If there is an intersection of two intervals, then the result could either be true or false.

Considering that our design comply with conventional rules we argue that the presented interval analysis is correct.

### Proof

To prove that the interval analysis given by us is an instance of monotone framework, we need to prove that the functions are monotone. Similar to the proof of sign detection, the functions are monotone if the function determining the interval of expressions and the transfer function deciding the interval of arithmetic expressions in Boolean expressions are monotone.

1) Firstly we check that the functions are monotone.

For the case , suppose we have

It is easy to verify that for , there are

Then for , suppose , where is the minimum value after multiplying the two *Interval* and , and is the maximum. Without losing any generality, let’s assume in which are two numbers in . Then for , we can always find an integer in or which is greater than (less than if is negative) or equal to . It is the same for . In other words, we could always find two integers in and whose absolute values are greater than or equal to the absolute values of and correspondingly. As the upper boundary of an interval after multiplication could only be a non-negative integer by definition, always holds. The same for the least bound . Thus we have proved . Finally, The same goes for .

For the case , it is very obvious that for , always holds.

Thus, we have proved that the functions are monotone.

2) Then we need to prove that is monotone. The proof is the same as the corresponding proof for sign detection analysis presented in the previous section.

Thus, it is proved that the interval analysis suggested by us for the project language is an instance of monotone framework.

### Implementation

In the implementation of the analysis model presented above, we’ve implemented most of the functions except for as it is too computation expensive. However we do implement for relation expressions that consist of one variable and one integer as the operands.

## Algorithm for array bound checking

This algorithm takes results from program graph , free variables in program under consideration and transition function table as defined in previous section that specifies the interval within which values of free variables occur for each construct of the while language. There are two main steps in array bound checking, the first is to construct a set of equations for interval analysis and solve them and the second is to use the solutions to determine statements in which violation of bounds occur. The algorithms for the two steps are presented below:

### Algorithm for solving Interval analysis equations

Input: Program Graph, Transition function table, Free variables

Output: Solutions to Interval analysis

For all in do

For all q in do

If then I(q) :={(x, )|x FV() }

else I(q) :=

While W≠nil do

(q, ,q’):= head(W);W=tail(W);

If ( I(q)), I(q’), x FV()

then I(q’) = I(q’) ( I(q)), x FV()

for all (q’, , q’’) in do

W:= cons((q’,, q’’),W);

Code 4.4.1 Algorithm for solving interval analysis equations

The first step is construction of a set of equations one for each node in the program graph and then solving them to get a set of intervals within which values for each free variable occur at that particular node. This step is carried out as a worklist algorithm and has two phases an initialization phase and an iteration phase. In initialization phase, a worklist and a set of equations are initialized. All edges of the program graph of the form are added to a worklist. Here, *q* is the starting node, the terminal node and the block in the program that causes this state change. The equations are initialized to extremal value which is for initial node and to least element for all other nodes.

In the second phase, following steps are repeated as long there is an element in the worklist. The first element of the worklist is considered for processing, represented by *head(W)* and of the form (*q, ,* ). The smaller worklist with the first element removed is placed in W. If the result of transition function on equations of q i.e. *I(q)* is not a subset or equals to that of *I(q’)* then we union existing value of *I()* with the value from transfer function on *I(q).* This is to ensure that all values of variables in a node flow to all nodes connected to it. The transition function makes use of a minimum and maximum bound for every variable and uses this to determine an interval for current value of variable. If value of a variable is greater than maximum value, upper bound of interval will be . If the value is less than minimum value lower bound of interval will be . Each edge that has the modified node as a starting point is added to the worklist, so that the changes are reflected in to all nodes where the information flows.

### Algorithm: Array Bound Checking

Input: Program Graph, Solutions of interval analysis for the program

Output: VL – Set of blocks in which violation of array bound occurs

For all (q,, q’) in do

If is a statement that includes an array as one of the free variables

If I(q’) for index of the array  ~~{[],[~~ or they don’t have intersection

then cons((),VL)

Code 4.2 Algorithm for array bound checking using interval analysis

The second stage of the algorithm for bound checking takes in all edges in the program graph and solutions of interval analysis as input. Following steps are repeated for each edge of the program graph. If block of an edge has an array element as one of the free variable and if the solution for that variable at the target node is a subset of {[], [ for index variable of the array, then the value of array index variable has violated the maximum or minimum array bound. If the solution for a variable at a target node is [], this means array index variable has a value that is less than minimum bound for the array when reaching that node. If solution is [ array index variable has a value that is greater than maximum bound for array when reaching that node. The worklist algorithm and transition functions perform the violation check and assign [ or [] based on type of violation. Thus array bound violation is detected in a block and if detected is added to output set *VL* containing all the blocks where boundary violation occurs.

## Constraints and solutions for the example program

For the detection of sign analysis we use program graph, as program graph makes it possible to use results of Boolean condition to pass different information to branches. The program graph for presented in Figure 4.1.



Figure .1. Program Graph of example program

Constraints for the aforementioned program are presented below.

*I (1) [ {0}, B{0}, {0}]*

*I (2) fn:=-1 (I (1))*

*I (3) fn<7 (I (2))*

*I (4) fa[n]:=1 (I (3))*

*I (5) fn>=0&n<5 (I (4))*

*I (6) f˥(n>=0&n<5) (I (4))*

*I (7) fb[n]:=1 (I (5))*

*I (7) fskip (I (6))*

*I (2) fn:=n+1(I (7))*

*I (8) f˥(n<7) (I (2))*

The solutions for the constraints include only a variable *n* since we are concerned of the buffer overflow analysis and *n* is the only variable in the program used as array index. Table 4.2 shows constraints solutions. Note, the table does not include all the arrays entries, since the point of the interests is the array boundaries check. However, the arrays entries would be represented as the variable *n.*

Table .2 Solutions to constraints for example program

|  |  |  |  |
| --- | --- | --- | --- |
| I() | n: min = 0, max = 4 | A | B |
| 1 | [0,0] | [0,0] | [0,0] |
| 2 | [-,] | [0,1] | [0,1] |
| 3 | [-,] | [0,1] | [0,1] |
| 4 | [-,] | [0,1] | [0,1] |
| 5 | [0,4] | [0,1] | [0,1] |
| 6 | [-,] | [0,1] | [0,1] |
| 7 | [-,] | [0,1] | [0,1] |
| 8 | [5,] | [0,1] | [0,1] |

## Discussion on precision of analysis

Our system is already of a high precision since the program graph is adopted for analysis in order to distinguish different conditions of Boolean expressions, while with flow graph this would not be possible. However our model could be improved if the array elements could be treated separately by interval. That results in a change in the complete lattice as well as the transfer functions. For example, one of the possible improvements could be change the complete lattice to be

We finally do not go for this implementation because the interval representing the array indices introduces too many set operations (union, intersection, subset) in the implementation. One compromise solution could be that the user inputs which array element he or she is interested and we treat the selected array elements as common variables.

# Security analysis

Example program for buffer overflow:

program

high int x;

low int y;

read x;

while x!=1 do

y := 1;

x := x – 1;

od

write y; (\* security violation \*)

end

The program presented above is the example program for security analysis. After the while loop, there is a violation of security since high level information is indirectly passed from *x* to *y.* Hence, *y* gets high security level and after the while loop statement *write y* will leak high level information about value of variable *x*.

## Security analysis

This subsection addresses the definition of the security analysis for the project language.

### Definition

The security analysis is defined as

where *L* is a complete lattice, is a set of transfer functions, *F* is a finite flow , *E* is a finite set of extremal labels, is an extremal value and is a mapping from labels to transfer functions.

#### Lattice L

The complete lattice *L* is defined as

where *ctx* security level of the context*.*

#### Mapping

The mapping of labels to transfer functions is constructed as

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

#### 

The which determines the signs of expressions is given by

|  |  |  |
| --- | --- | --- |
|  |  | *{low}* |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

where which could be and is specified by

Finally, the which could be and /, and which could only be are defined in the Table 3.1 and Table 3.2.

Table .1

|  |  |  |
| --- | --- | --- |
|  | low | high |
| low | {low} | {high} |
| high | {high} | {high} |

#### 

is given by

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

Finally, the and are defined with the tables below respectively.

Table .3

|  |  |  |
| --- | --- | --- |
|  | low | high |
| low | {low} | {high} |
| high | {high} | {high} |

Table .4

|  |  |  |
| --- | --- | --- |
| & | low | high |
| low | {low} | {high} |
| high | {high} | {high} |

# Appendix

**Appendix A The syntax of the while language**

Operators:

Notation:

**Appendix B An example for construcing flow graph**

This appendix gives a Java pseudo code example for *IfFlowGraph* to demonstrate how the flow graph for *IfStatement* is created (the fields of the flow graph are filled) recursively using the rules in Table 1.2 and Table 1.3.

Example:

IfFlowGraph {

//constructor

public IfFlowGraph(IfStatement st) {

// init variables

this.blocks = getBlocks();

labels = new Vector<int>();

flows = new Vector<Flow>();

final = new Vector<int>();

ancestorBoolLabel = -1;

// record new blocks and new labels

int l = blocks.getSize() + 1;

String boolBlock = st.getBoolExpr();

blocks.add(boolBlock);

lables.add(l);

// graph is created recursively

FlowGraph s1 = FlowGraphFactory.create(st.getThenStatement ());

FlowGraph s2 = FlowGraphFatrory.create(st.getElseStatement());

// set the ancestor labels in the blocks in the graph as l.

s1.setAncestorBoolLabel(l);

s2.setAncestorBoolLabel(l);

//follow the rules in and

labels.add(s1.getLabels());

labels.add(s2.getLabels());

init = l;

final.add(s1.getFinals());

final.add(s2.getFinals());

flows.add(s1.getFlow());

flows.add(s2.getFlow());

flows.add(new Flow(l, s1.getInit()));

flows.add(new Flow(l, s2.getInit()));

}

Besides, the factory class implementation is shown as below:

class FlowGraphFactory {

public static FlowGraph create(Statement st) {

if(st instanceof IfStatement)

return new IfFlowGraph(st);

else if…

}

}

**Appendix C The pseudo code of classes for constructing program graphs**

Class WhileProgramGraph extends ProgramGraph {

public WhileProgramGraph (WhileStatement st, int initialNode, int finalNode) { //constructor

String boolBlock = st.getBoolExpr().toString();

If (edges.empty()== false)

Edges.add(new edge(initialNode, boolblock, edges.last().qt +1);

Else edges.add( new edge(1, boolblock,2));

// graph is created recursively

ProgramGraphFactory.create(st.getStatement (), edges.last().qt, initialNode);

Edges.add(new edge(initialNode, boolblock= “!”+ boolblock, finalNode ? finalNode : edges.last().qs +1 );

// add exit from the loop: if this While is the last statement in the else branch than finalNode will be set and we use it, otherwise we use edges.last().qs +1 since edges.last().qt is the initial node of the enter condition for the loop.

}

}

Class SeqProgramGraph extends ProgramGraph {

public SeqProgramGraph (SeqStatement st, int initialNode, int finalNode) { //constructor

// Create factory

ProgramGraphFactory.create(st.getStatement(), initNode, 0);

ProgramGraphFactory.create(st.getStatement(),edges.last().qt, finalNode);

}

}

Class AssignProgramGraph extends ProgramGraph {

public AssignProgramGraph (AssignStatement st, int initialNode, int finalNode) { //constructor

String block = st.getArithExpr().toString();

If (edges.empty()== false)

Edges.add(new edge(initialNode, block, finalNode ? finalNode : edges.last().qt +1);

Else edges.add( new edge(1, boolblock,2));

}

}

The rest classes are very similar to *AssignProgramGraph* class.

**Appendix D Calculation of the program slice using the algorithms in Section 2.3**

The results of block analysis by using the algorithm in Section 2.3.2 are in below table.

Table D.1 The results of block analysis by using the algorithm in Section 2.3.2

|  |  |  |
| --- | --- | --- |
| Label | Variable | Variable Position |
| 1 | n | left |
| 2 | f1 | left |
| 3 | f2 | left |
| 4 | x | left |
| 5 | ans | left |
| 6 | x | none |
| 6 | n | none |
| 7 | ans | left |
| 7 | f1 | right |
| 7 | f2 | right |
| 8 | f1 | left |
| 8 | f2 | right |
| 9 | f2 | left |
| 9 | ans | right |
| 10 | x | left |
| 10 | x | right |

The application of the results of the other steps of the algorithm is as follows:

W:= nil;

pointOfInterest = 9;

W:=9;

*Iteration 1:*

W nil;

currentLineOfInterest = head(W)= 9;

W:=tail(W) = nil;

programSlice = 9;

Free variables in 9 i.e. f2:=ans and variablePosition not left is ans.

udchain(ans,9) = 7 from ;

7 is not in programslice;

W := 7;

Boolean ancestor of 9 is 6. 6 is not in program slice;

W := 6,7;

*Iteration 2:*

W nil;

currentLineOfInterest = head(W)= 6;

W:=tail(W) = 7;

programSlice = 6,9;

Free variables in currentLineOfInterest namely 6 i.e. x <= n and variablePosition not left are x and n;

udchain(x,6) = 4,10 from ;

udchain(n,6) = 1 from ;

1,4, and 10 are not in programslice;

W := 1,4,10,7;

Boolean ancestor of 6 is none;

*Iteration 3:*

W nil;

currentLineOfInterest = head(W)= 1;

W:=tail(W) = 4,10,7;

programSlice = 1,6,9;

No free variables in currentLineOfInterest namely 1 i.e. n:=20 and variablePosition not left .Hence, nothing to add to Worklist;

1 has no boolean ancestor.;

*Iteration 4:*

W nil;

currentLineOfInterest = head(W)= 4;

W:=tail(W) = 10,7;

programSlice = 1,4,6,9;

Free variables in currentLineOfInterest namely 4 i.e. x:=2 and variablePosition not left .Hence, nothing to add to Worklist;

4 has no boolean ancestor;

*Iteration 5:*

W nil

currentLineOfInterest = head(W)= 10;

W:=tail(W) = 7;

programSlice = 1,4,6,9,10;

Free variables in currentLineOfInterest namely 10 i.e. x:=x+1 and variablePosition not left is x;

udchain(x,10) = from ;

Both are in programSlice and hence, nothing to add to Worklist;

10 has 6 as boolean ancestor. But 6 is in programSlice hence, nothing to add to Worklist;

*Iteration 6:*

W nil;

currentLineOfInterest = head(W)= 7;

W:=tail(W) = ;

programSlice = 1,4,7,6,9,10;

Free variables in currentLineOfInterest namely 7 i.e. ans:=f1+f2; and variablePosition not left are f1 and f2;

udchain(f1,7) = from ;

udchain(f2,7) = from ;

2,3,8 are not in programSlice Hence, adding them to Worklist;

W:=2,3,8;

7 has 6 as boolean ancestor. But 6 is in programSlice hence, nothing to add to Worklist;

*Iteration 7:*

W nil;

currentLineOfInterest = head(W)= 2;

W:=tail(W) =

programSlice = 1,2,4,6,7,9,10;

No free variables in currentLineOfInterest namely 2 i.e. f1:=0 and with variablePosition not left. Hence, nothing is added to Worklist;

2 has no boolean ancestor. Hence, nothing is added to Worklist;

*Iteration 8:*

W nil;

currentLineOfInterest = head(W)= 3;

W:=tail(W) = ;

programSlice = 1,2,3,4,6,7,9,10;

No free variables in currentLineOfInterest namely 3 i.e. f2:=1 and with variablePosition not left. Hence, nothing to add to Worklist;

3 has no boolean ancestor. Hence, nothing is added to Worklist;

*Iteration 9:*

W nil;

currentLineOfInterest = head(W)= 8;

W:=tail(W) = ;

programSlice = 1,2,3,4,6,7,8,9,10;

Free variables in currentLineOfInterest namely 8 i.e. f1:=f2 and with variablePosition not left is f2;

udchain(f2,8) = from ;

3 and 9 are in programSlice. Hence, nothing to add to Worklist;

8 has 6 as boolean ancestor and 6 is present in programSlice. Hence, nothing is added to Worklist.;

*Iteration 10:*

W = nil;

The result shows that the program slice that is returned is programSlice = 1,2,3,4,6,7,8,9,10 which is in accordance with the target result.

# References

[1] Nielson, H. R. (2013, Sept 09). Lecture 2: Reaching definition, slides presented at DTU.

[2] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. Principles of program analysis. Springer-Verlag, 2005.

[3] Nielson, H. R. (2013, Oct 28). Lecture 7: Advanced algorithms, slides presented at DTU.