02242 Program Analysis

Anusha Sivakumar	s124571@student.dtu.dk
Nikita Martynov	s124570@student.dtu.dk
Zhen Li	s124568@student.dtu.dk

Table of Contents

1. Introduction

Software is often a vital component of mission critical systems. It is important that software performs as expected and does not crash because a software bug could prove to be costly. One major approach to ensure robustness of software is to use program analysis. Program analysis techniques analyze a program without actually executing the program. Program analysis helps in finding potential bugs and optimizing code. Program analysis is also useful when simulating the real environment in a test bed is very difficult and expensive. In this report, we have discussed about the design and implementation of four program analysis techniques – reaching definition analysis, detection of signs analysis, interval analysis and security analysis. We have implemented these program analysis modules for 'while' language, the syntax of which is available in the appendix A.

The remainder of this report is organized as follows: Section 2 describes the representation of program as an AST and explains the data structures and algorithm for flow graph and program graph. Section 3 presents the theory, design, implementation and test results of reaching definition analysis and program slicing. Section 4 explains the detection of signs as a monotone framework, proves the correctness of equations and discuss the design and implementation of the module. The representation of interval analysis as a monotone framework, proof of correctness, design, implementation and test results are presented in Section 5. The design of an algorithm for security analysis and its usefulness in finding security problems is described in section 6. Finally, our observations are presented as a conclusion in Section 7.

2. 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.

2.1. 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 (Appendix A), 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 2.1– Figure 2.5. 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.

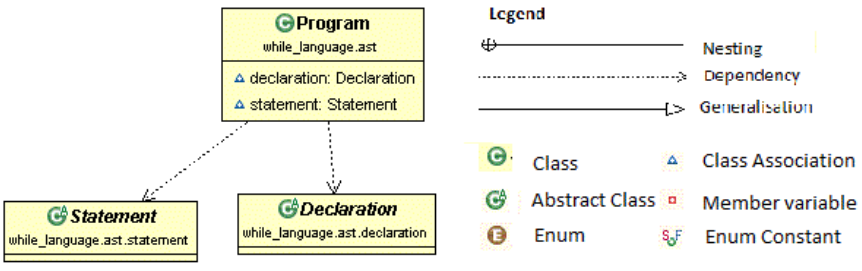

Figure 2.1 The UML diagram for *Program*, *Declaration* and *Statement*

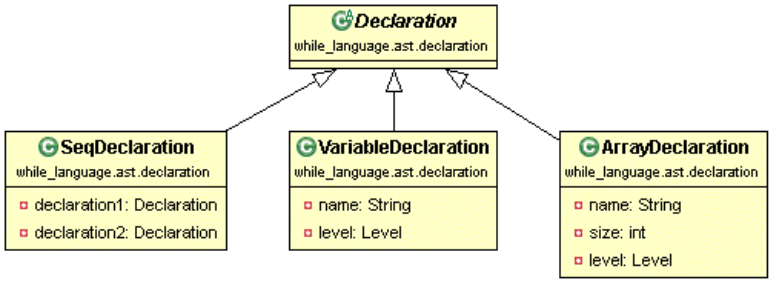

Figure 2.2 The UML diagram for *Declaration*

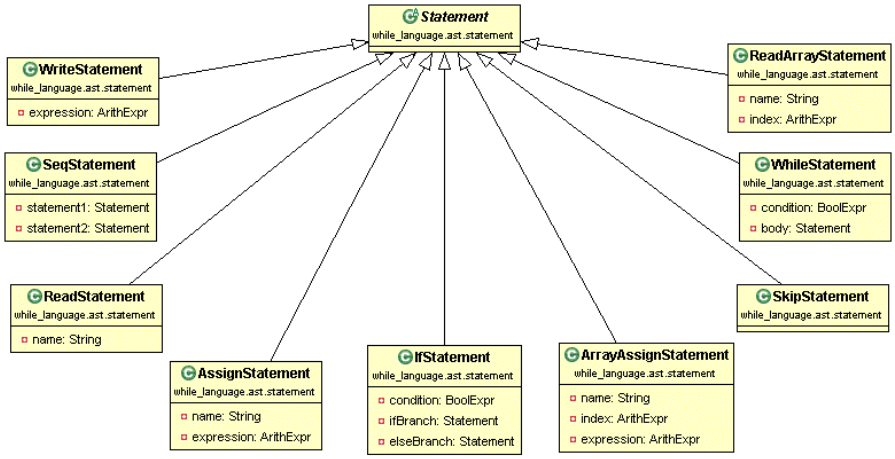

Figure 2.3 The UML diagram for *Statement*

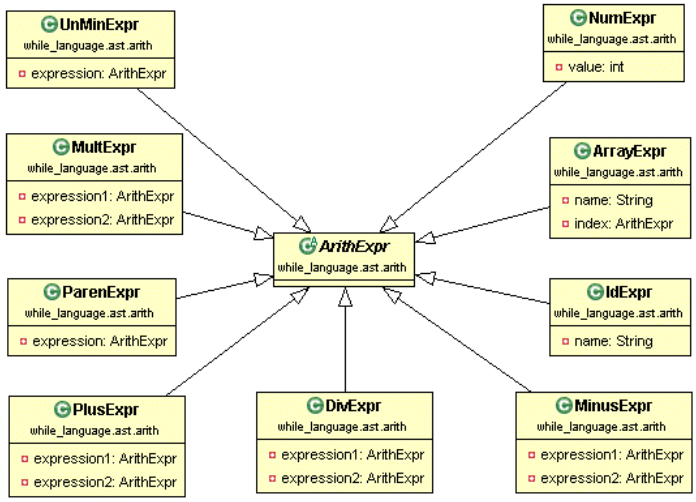

Figure 2.4 The UML diagram for *ArithExpr*

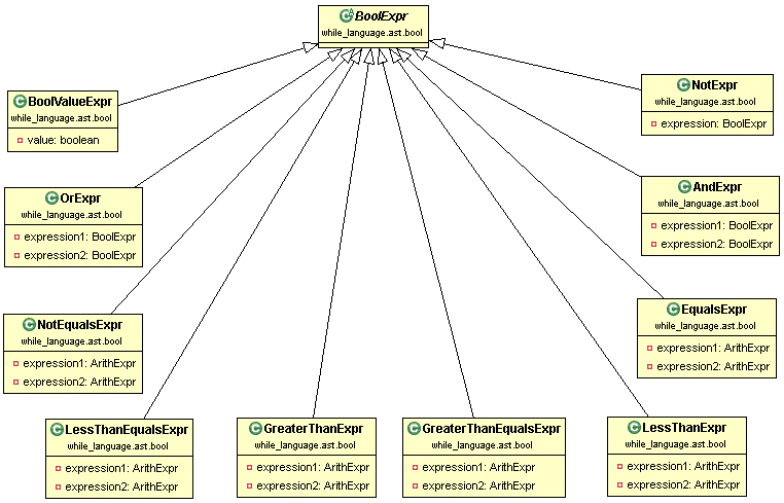

Figure 2.5 The UML diagram for *BoolExpr*

2.2. 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.

2.2.1. Data structure

The pseudo code presented in Code 2.1 shows the instance field of the data structure for flow graphs.

Figure 2.6 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.

```
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 2.1 The *FlowGraph* class for flow graphs

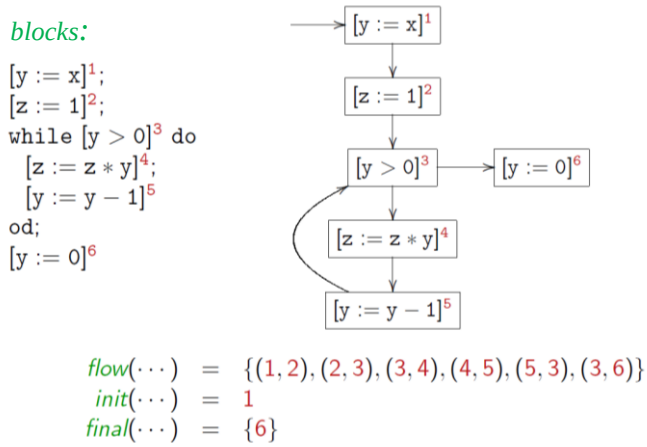

Figure 2.6 An example for the data structure of the flow graph

2.2.2. 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 2.1. 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 2.2 and Table 2.3 define rules to fill the instance field in each *FlowGraph* during the mapping from Statement classes to *FlowGraph* classes.

Table 2.1 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 2.2 Rules for constructing flow graph (part1)

S	***labels(S)***	***init(S)***	***final(S)***
$[x := a]^{\ell}$	$\{\ell\}$	ℓ	$\{\ell\}$
$[skip]^{\ell}$	$\{\ell\}$	ℓ	$\{\ell\}$
$[A[a_1] := a_2]^{\ell}$	$\{\ell\}$	ℓ	$\{\ell\}$
$[read\ x]^{\ell}$	$\{\ell\}$	ℓ	$\{\ell\}$
$[read\ A[a]]^{\ell}$	$\{\ell\}$	ℓ	$\{\ell\}$
$[write\ a]^{\ell}$	$\{\ell\}$	ℓ	$\{\ell\}$

$S_1; S_2$	$labels(S_1) \cup labels(S_2)$	$init(S_1)$	$final(S_2)$
$if\ [b]^\ell\ then\ S_1\ else\ S_2$	$\{\ell\} \cup labels(S_1) \cup labels(S_2)$	ℓ	$final(S_1) \cup final(S_2)$
$while\ [b]^\ell\ do\ S$	$\{\ell\} \cup labels(S)$	ℓ	$\{\ell\}$

Table 2.3 Rules for constructing flow graph (part2)

S	***flow(S)***	***blocks(S)***
$[x := a]^\ell$	$\emptyset$	$\{[x := a]^\ell\}$
$[skip]^\ell$	$\emptyset$	$\{[skip]^\ell\}$
$[A[a_1] := a_2]^\ell$	$\emptyset$	$\{[A[a_1] := a_2]^\ell\}$
$[read\ x]^\ell$	$\emptyset$	$\{[read\ x]^\ell\}$
$[read\ A[a]]^\ell$	$\emptyset$	$\{[read\ A[a]]^\ell\}$
$[write\ a]^\ell$	$\emptyset$	$\{[write\ a]^\ell\}$
$S_1; S_2$	$flow(S_1) \cup flow(S_2)$ $\cup \{(l, init(S_2)) \mid l \in final(S_1)\}$	$blocks(S_1) \cup blocks(S_2)$
$if\ [b]^\ell\ then\ S_1\ else\ S_2$	$flow(S_1) \cup flow(S_2)$ $\cup \{(l, init(S_1)), (l, init(S_2))\}$	$\{[b]^\ell\} \cup blocks(S_1)$ $\cup blocks(S_2)$
$while\ [b]^\ell\ do\ S$	$\{(l, init(S))\} \cup flow(S)$ $\cup \{(l', l) \mid l' \in final(S)\}$	$\{[b]^\ell\} \cup blocks(S)$

2.3. Program graphs

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

2.3.1. Data structure

The Java-like pseudo code of the program flow data structure is presented in Code 2.2.

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.

```
Class Edge{
    Int qs;
    String block;
    Int qt;
}
```

Code 2.2 The *Edge* class for program graphs

2.3.2. Rules for constructing program graphs

The rules for constructing the program graph from the AST are presented in the Table 2.4.

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 $pg_{q_S}^{q_t}(S)$ at the right hand side of the Table 2.4 denotes recursion in the algorithm.

Table 2.4 Rules for transforming the AST to the program graph where *S* is a statement and $pg_{q_S}^{q_t}(S)$ represents the program graph for statement *S* with initial node q_s and final node q_t.

S	$pg_{q_s}^{q_t}(s)$
$[x:=a]^l$	$\{(q_s, [x:=a]^l, q_t)\}$
$[skip]^l$	$\{(q_s, [skip]^l, q_t)\}$
S1;S2	$pg_{q_s}^{q}(S1) \cup pg_{q}^{q_t}(S2)$, *where q is a fresh node*
If $[b]^l$ *then S1 else S2*	$\{(q_s, [b]^l, q_1), (q_s, [\neg b]^l, q_2)\} \cup pg_{q_1}^{q_t}(S1) \cup pg_{q_2}^{q_t}(S2)$, *where* q_1 *and* q_2 *are fresh nodes*
While $[b]^l$ *do S*	$\{(q_s, [b]^l, q), (q_s, [\neg b]^l, q_t)\} \cup pg_{q}^{q_s}(S)$, *where q is a fresh node*
$[A[a_1]:=a_2]^l$	$\{(q_s, [A[a_1]:=a_2]^l, q_t)\}$
$[read\ x]^l$	$\{(q_s, [read\ x]^l, q_t)\}$
$[write\ a]^l$	$\{(q_s, [wrire\ a]^l, q_t)\}$
$[read\ A[a]]^l$	$\{(q_s, [read\ A[a]]^l, q_t)\}$

2.3.3. Implementation

For implementation the vector of edges is represented in the abstract class *ProgramGraph* and it is shown in Code 2.3. 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.

Each of the statements from the Table 2.4 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 2.4. 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 2.4 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 2.5.

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 2.5.

Code 2.6 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.

```
abstract class ProgramGraph{
      Static Vector < edge > edges = new Vector < edge >();
}
```

Code 2.3 The *ProgramGraph* class pseudo code

```
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 2.4 The *IfProgramGraph* class pseudo code

```
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 2.5 The *ProgramGraphFactory* class pseudo code

```
void Main(){
    ProgramGraphFactory.create(st.getStatement(), 1, 0);
}
```

Code 2.6 Main function

3. 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;
od
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 3.1. 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 3.1 The program slice for the point of interest in the example program

3.1. 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 3.1 is constructed with the assumption that input program $S_\star$ has isolated entries.

Table 3.1 Data Flow Equations for Reaching Definition Analysis

$kill_{RD}([x := a]^l)$	$\{(x,?)\} \cup \{(x,l') \mid B^{l'} \text{ is an assignment to } x \text{ in } S_\star\}$
$kill_{RD}([read\ x]^l)$	$\{(x,?)\} \cup \{(x,l') \mid B^{l'} \text{ is an assignment to } x \text{ in } S_\star\}$
$kill_{RD}([A[a1] := a2]^l)$	$\emptyset$
$kill_{RD}([read\ A[a]]^l)$	$\emptyset$
$kill_{RD}([b]^l)$	$\emptyset$
$kill_{RD}([write\ a]^l)$	$\emptyset$
$kill_{RD}([skip]^l)$	$\emptyset$
$gen_{RD}([x := a]^l)$	$\{(x,l)\}$
$gen_{RD}([read\ x]^l)$	$\{(x,l)\}$
$gen_{RD}([A[a1] := a2]^l)$	$\{(A,l)\}$
$gen_{RD}([read\ A[a]]^l)$	$\{(A,l)\}$
$gen_{RD}([skip]^l)$	$\emptyset$
$gen_{RD}([b]^l)$	$\emptyset$
$gen_{RD}([write\ x]^l)$	$\emptyset$

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 $\emptyset$ 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 RD_{entry} and RD_{exit} of label *l* are defined as below:

$$RD_{entry}(l) = \begin{cases} \{(x,?) \mid x \in FV(S_\star)\} & if\ l = init(S_\star) \\ \cup \{RD_{exit}(l') \mid (l,l') \in flow(S_\star)\} & otherwise \end{cases}$$

$$RD_{exit}(l) = \left(RD_{entry}(l) \backslash kill_{RD}(B^l)\right) \cup gen_{RD}(B^l) \text{ where } B^l \in blocks(S_\star)\}$$

3.2. Manual computation of solution

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

Table 3.2 *kill* and *gen* equations for example program

Label	$kill_{RD}$ (l)	gen_{RD} (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 3.1. 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 RD_{entry} and RD_{exit} information for the example program is presented in the table below.

Table 3.3 RD_{entry} and RD_{exit} information for the example program

$\boldsymbol{RD_{entry}}$ / $\boldsymbol{RD_{\circ}}$	$\boldsymbol{RD_{exit}}$ / **RD•**
$RD_{\circ}(1) = \{(n, ?), (f1, ?), (f2, ?), (x, ?), (ans, ?)\}$	RD•(1) = $RD_{\circ}$ (1) \ ($kill_{RD}$ (1)) U gen_{RD} (1);
$RD_{\circ}$ (2) = RD•(1)	RD•(2) = $RD_{\circ}$ (2) \ ($kill_{RD}$ (2)) U gen_{RD} (2);
$RD_{\circ}$ (3) = RD•(2)	RD•(3) =$RD_{\circ}$ (3) \ ($kill_{RD}$ (3)) U gen_{RD} (3);
$RD_{\circ}$ (4) = RD•(3)	RD•(4) = $RD_{\circ}$ (4) \ ($kill_{RD}$ (4)) U gen_{RD} (4);
$RD_{\circ}$ (5) = RD•(4)	RD•(5) = $RD_{\circ}$ (5) \ ($kill_{RD}$ (5)) U gen_{RD} (5);
$RD_{\circ}$ (6) = RD•(5) U RD•(10)	RD•(6) = $RD_{\circ}$ (6);
$RD_{\circ}$ (7) = RD•(6)	RD•(7) = $RD_{\circ}$ (7) \ ($kill_{RD}$ (7)) U gen_{RD} (7);
$RD_{\circ}$ (8) = RD•(7)	RD•(8) = $RD_{\circ}$ (8) \ ($kill_{RD}$ (8)) U gen_{RD} (8);
$RD_{\circ}$ (9) = RD•(8)	RD•(9) = $RD_{\circ}$ (9) \ ($kill_{RD}$ (9)) U gen_{RD} (9);
$RD_{\circ}$ (10) = RD•(9)	RD•(10) = $RD_{\circ}$ (10) \ ($kill_{RD}$ (10)) U gen_{RD} (10);

The above sets of equations are based on specification in section 3.1. 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 3.4 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 3.4 Solution set

Label	$RD_{\circ}$ (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),	{(n, 1), (f1, 2), (f2, 3), (x, 4), (ans, 5), (f1, 8), (f2,

	(f2, 9), (x, 10), (ans, 7)}	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)}

3.3. 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 $kill_{RD}$ and gen_{RD}, and finally the algorithm to compute ud-chains.

3.3.1. 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: blocks in the program, point of interest in program
OUTPUT: List of labels that affect point of interest
ALGORITHM:
GenerateFreeVariables()
ReachingDefinitionAnalysis()
W := nil;
 If pointOfInterest := null or pointOfInterest > blocks.size then
      return null;
else
     W := cons(pointOfInterest ,W );
     While W ≠nil  do
                currentLineOfInterest := head(W);
                W := tail(W);
                programSlice := cons(currentLineOfInterest,programSlice);
                 for each free variable in the currentLineOfInterest with variablePosition not left and not
                (variablePosition not write and write statement is the point of interest)
                          for each label l in udchain(freevariable,currentLineOfInterest)
                          if l not in programSlice and not in head of worklist
                                    cons(l, W);
                          if ( currentLineOfInterest has a Boolean ancestor and labelOfbooleanAncestor not in
                          programSlice and not in worklist)
                                    cons(labelOfBooleanAncestor, W);
  return programSlice;
```

Code 3.2 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*()function returns 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 *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.

3.3.2. Algorithm to compute free variables

```
GenerateFreeVariables()
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
            Get position of variable in statement from(left, right, write, index, none)
            Cons((variableInstatement,linenumber,variablePosition),Variables);
return Variables;
```

Code 3.3 Algorithm to compute free variables

This algorithm finds the free variables in the program and their position with respect to assignment operator i.e. 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.

3.3.3. Algorithm to compute reaching definition

```
ReachingDefinitionAnalysis()
Input: A set of reaching definition equations
Output: The least solution to the equations RD∘
W := nil;
for all (l, l′) in flow(S⋆) do W := cons((l, l′), W);
for all l in labels(S⋆)do
        if l = init(S⋆)then RD∘(l) := {(x, ?)| x ∈ FV(S⋆)} ;
        else RD∘(l) := ∅ ;
            while W ≠ nil do
                (l, l′) := head(W); W := tail(W);
                if (RD∘(l)  \kill_RD(l)) ∪ gen_RD(l) ⊈ RD∘(l′)
                then RD∘(l′) := (RD∘(l)  \kill_RD(l)) ∪ gen_RD(l) ;
                for all l" with (l′, l") in flow(S⋆)
                        W := cons((l′, l"), W) ;
 Compute (RD∘(l)  \kill_RD(l))∪gen_RD(l) where l is highest label in flow
```

Code 3.4 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 (l, l') are added to a worklist. Here, information flows from label l to label l'. 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 $\emptyset$ 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 (l, l') . The smaller worklist with the first element removed is placed in *tail*(W). The transition function $RD_{\circ}(l) \ \backslash kill_{RD}(l)) \cup gen_{RD}(l)$ 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 $RD_{\circ}(l')$ then we union the existing value of $RD_{\circ}(l')$ with the value from transfer function on $RD_{\circ}(l)$. 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 l is the input to l'. Each label that has the label l' as a starting point is added to the worklist, so that the changes are reflected in all equations of labels where information flows into.

3.3.4. Algorithm to compute $kill_{RD}$ and gen_{RD}

Input: Free Variable information, blocks in program
Output: $kill_{RD}$ and gen_{RD} for each block in input program
For each block B^l in input program
  If B^l is an assignment statement
  then get variable x whose position in B^l is 'left'
    $kill_{RD}(B^l) = \{(x, ?) \cup (x, l') | (B^{l'})$is an assignment to x$\}$
    $gen_{RD}(B^l) = \{(x, l)\}$;
  Else if B^l is a read statement
  then get free variable x in B^l
    $kill_{RD}(B^l) = \{(x, ?) \cup (x, l') | (B^{l'})$is an assignment to x$\}$
    $gen_{RD}(B^l) = \{(x, l)\}$;
  Else
    $kill_{RD}(B^l) = \emptyset$
    $gen_{RD}(B^l) = \emptyset$

Code 3.5 Algorithm to compute killRD and genRD

The algorithm generates equations for gen_{RD} (what is newly modified in statement) and $kill_{RD}$ (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 $\emptyset$ is assigned.

3.3.5. Algorithm to compute ud-chains

udchain(variable,label)
Input: Block Analysis results, currentLineOfInterest,RD_{entry}

```
Output: udChain for currentLineOfInterest
For each variable in Variables
If( variable present in currentLineOfInterest and      variablePosition is not 'left' )
      For each (variable,lineNumber) in RD_entry(currentLineOfInterest)
            cons((lineNumber),udChain);
```

Code 3.6 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 RD_{entry} equation of the line and added to the output list.

3.3.6. Data structures

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

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

```
Class ReachingDefinition{
      String variableName;
      Int label;
}
```

Code 3.7 Class *ReachingDefinition*

The *ReachingDefinitionColln* is a wrapper class that stores a set of ReachingDefinition objects.

```
Class ReachingDefinitionColln{
       Set<ReachingDefinition> rdSet;
}
```

Code 3.8 Class *ReachingDefinitionColln*

The *KillandGenAnalysis* class is used to generate $kill_{RD}$ and gen_{RD} equations for all blocks in a statement based on the type of the statement.

```
Class KillandGenAnalysis {
      ReachingDefinitionColln killRD;
      ReachingDefinitionColln genRD;
      getKillRD(Int label);
      getGenRD(Int label);
}
```

Code 3.9 Class *KillandGenAnalysis*

The solution to RD_{Entry} and RD_{Exit} equations for a single statement is stored in *ReachingDefinitionAnalysis* class.

```
Class ReachingDefinitionAnalysis{
      ReachingDefinitionColln   rdEntry;
      ReachingDefinitionColln rdExit;
}
```

Code 3.10 Class *ReachingDefinition*

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 3.11 Class *FreeVariable*

The *VariablePosition* enum defines the possible positions a variable can take in a statement.

```
enum VariablePosition {
    left, //left side of assignment operator
    right, //right side of assignment operator
    read, // variable present in a read statement
    write, //variable present in a write statement
    index,//variable present in an array index
    none;  // variable present in a statement other than assignment statement
}
```

Code 3.12 Enum VariablePosition

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

```
Class FreeVariableGenerator{
    //Input
    Vector<Block> blocks;
    //Output
    Vector< FreeVariable > Variables;
}
```

Code 3.13 Class *BlockAnalysis*

The *BooleanAncestorFinder* class is used to store information about the most recent ancestor for each label.

```
Class BooleanAncestorFinder {
    ArrayList<Integer> ancestors;
}
```

Code 3.14 Class *BooleanAncestorFinder*

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;
    //Output
    Vector<int> sliceLabels;
    Vector<int> getudChain(String,int);
}
```

Code 3.15 Class *ProgramSlice*

3.4. 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 3.2. 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 **B**. Finally, the code slice shown in Code 3.16 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 3.16 Program slice for the point of interest in the example program

3.5. Implementation

We have implemented all algorithms for reaching definition analysis and program slicing mentioned in the previous sections. The source code for both reaching definition analysis and program slicing is available in the package *program_slicing*. The main class for program slicing namely *ProgramSlice* exposes static methods to calculate program slice and print the computed slice. The *getProgramSlice* method in the *ProgramSlice* class requires reaching definition analysis results and the most recent Boolean ancestor values to calculate the program slice. The *getudchain* method exposed by *ProgramSlice* class computes the ud-chain for a given free variable and label. The *ReachingDefinitionAnalysis* class exposes a static *analyze* method that can be used to compute RDEntry and RDExit equations for each label in the program. This method makes use of static *analyze* method exposed by *KillandGenAnalyis* class to compute the KillRD and GenRD equations. The *ReachingDefinition* class defines the structure of a reaching definition – a variable and a label that is used by all the other classes in the package. The *ReachingDefinitionColln* class is a wrapper around a set of reaching definitions that are stored in the equations and exposes methods to union, complement other reaching definition collections.

BooleanAncestorFinder is a class that exposes *computeBoolAncestor* method to find the most recent or most immediate Boolean ancestor in the program hierarchy. It makes use of static *boolEndingEdges* variable exposed by *ProgramGraph* class. The most immediate Boolean ancestor is required to determine which of the conditional statements indirectly impact the execution of a particular statement in the program. All impacting Boolean statements also are included in the slice.

The package *free_variables* contains classes to generate free variables used by the *ProgramSlice* class. The *FreeVariableGenerator* is the main class in the package that exposes a static method *extractVariables* to compute variables in each line of the program. *VariablePosition* is an enum that represents the position of a variable within a statement. *FreeVariable* is a class that is used to store details about a variable. These two classes are used by the *FreeVariableGenerator* class.

3.6. Future improvements

In order to determine where a conditional statement begins and ends we have made use of program graphs. This choice was motivated by the fact that program graphs store the value of both the condition and its negated form in the blocks thus making it the easier to locate the end of the statement. On the other hand, for computing kill and gen analysis, reaching definitions and program slices, we have made use of flow graphs since our implementation exposed the blocks and labels which more naturally corresponds with the structure of slice. In slice we compute the labels impact a particular label. A more optimized solution would be to use only one of program or flow graphs.

Our implementation does not take into account the values assigned or stored in variables. It only considers the possibility that a variable's value could be modified in a statement or not. This affects the precision of our algorithm. Two of such cases are presented in Code 3.17

The program slice for the program in Code 3.17 when point of interest 7 is: 1,2,3,4,7. Even though only the third element of the array is written in statement 7, our implementation ignores the index part, considers the entire array as one variable and computes the slice. Hence, statements like label 4 get included in the output making the slice imprecise. The other case was found when we tested our implementation against a benchmark program by Maja Tønnesen (Code 3.18).

The slice output for the above program when point of interest is 6 is shown in Code 3.19. This program demonstrated an important limitation in our implementation. The computation of slice does not take into account the actual values of variables. For example, since x is assigned a value of 10 in label 1, the then branch of if statement will not get executed since x<5 will now be 10<5, which is false. Hence, ideally the slice for label 6 i.e. write y should not include label 4, as label 4 will not get executed and will not affect label 6. However, we currently do not take into account the actual value of variables and only consider whether a variable gets modified or not in a particular label. The output matches the expected output mentioned in the benchmark program. Thus taking into account the values for variables will make the program more precise. This is one of the areas of future improvement.

```
int A[10];
int x;
[A[1] := 2]^1;
[read x]^2;
if [A[1]>x]^3 then
[A[2] := 1]^4;
[write x]^5;
else
[write A[x]]^6;
fi
[write A[3]]^7;
```

Code 3.17 Improvement – Array Example Program

```
program
    int x;
    int y;
    x := 10; (* L1 *)
    y := 5;  (* L2 *)
    if(x<5)  (* L3 *)
            then y := x-y; (* L4 *)
            else y := 10;  (* L5 *)
    fi
     write y; (* real: y = {L5, L3, L1}, expected deduction: y = {L5, L4, L3, L2, L1} *)
```

```
end
```

Code 3.18 Benchmark Program

```
Input Program with labels added:
int x;
int y;
[x := 10]^1;
[y := 5]^2;
if [x<5]^3 then
[y := x-y]^4;
else
[y := 10]^5;
fi
[write y]^6;

The program slice when point of interest 6 is: 1,2,3,4,5,6
```

Code 3.19 Output of Benchmark Program

3.7. Benchmark testing

In this section we evaluate our implementation using our example program and some benchmarks given by our peers. We also discuss the limitations or design choices that are relevant to these benchmarks.

3.7.1. Example program

The output of program slicing and reaching definition analysis for the program at the beginning of this chapter is discussed in this section.

We compute the free variables in each line of the program, their position in the statement and the line number in which they are occurring. The position could be left or right of assignment statement, index of an array, read or write statement or none of these. We then use the blocks generated from flow graphs and the above free variable analysis to determine the equations for $kill_{RD}$ and gen_{RD}. $kill_{RD}$ and gen_{RD} are used to generate `RDEntry and RDExit` for all the labels in the program. The output for program slicing for the example program illustrated in Code 3.20.

We have chosen to include the conditional statements like if and while that contain the statements in the slice. This is because the conditionals influence whether a statement contained within is executed or not and hence, indirectly influence the values of variables in these statements. To determine the starting point and ending point of a conditional we have made use of program graphs.

```
Free Variables:
(n, left, 1) (f1, left, 2) (f2, left, 3) (x, left, 4) (ans, left, 5) (x, none, 6) (n, none, 6) (ans, left, 7) (f1, right, 7) (f2, right, 7) (f1, left, 8) (f2, right, 8) (f2, left, 9) (ans, right, 9) (x, left, 10) (x, right, 10) (x, write, 11)

killRD:
killRD(1) = {(n,?) (n,1) }
killRD(2) = {(f1,?) (f1,8) (f1,2) }
killRD(3) = {(f2,?) (f2,9) (f2,3) }
killRD(4) = {(x,?) (x,10) (x,4) }
killRD(5) = {(ans,?) (ans,7) (ans,5) }
killRD(6) = {}
killRD(7) = {(ans,?) (ans,7) (ans,5) }
killRD(8) = {(f1,?) (f1,8) (f1,2) }
killRD(9) = {(f2,?) (f2,9) (f2,3) }
killRD(10) = {(x,?) (x,10) (x,4) }
killRD(11) = {}
genRD:
```

```
genRD(1) = {(n,1) }
genRD(2) = {(f1,2) }
genRD(3) = {(f2,3) }
genRD(4) = {(x,4) }
genRD(5) = {(ans,5) }
genRD(6) = {}
genRD(7) = {(ans,7) }
genRD(8) = {(f1,8) }
genRD(9) = {(f2,9) }
genRD(10) = {(x,10) }
genRD(11) = {}

RDEntry:
RDEntry(1) = {(ans,?) (x,?) (f2,?) (f1,?) (n,?) }
RDEntry(2) = {(f1,?) (f2,?) (x,?) (ans,?) (n,1) }
RDEntry(3) = {(ans,?) (x,?) (f2,?) (f1,2) (n,1) }
RDEntry(4) = {(x,?) (ans,?) (f1,2) (f2,3) (n,1) }
RDEntry(5) = {(ans,?) (f1,2) (f2,3) (n,1) (x,4) }
RDEntry(6) = {(x,10) (f1,8) (f2,9) (f1,2) (f2,3) (n,1) (ans,7) (ans,5) (x,4) }
RDEntry(7) = {(x,10) (f1,8) (f2,9) (f1,2) (f2,3) (n,1) (ans,7) (ans,5) (x,4) }
RDEntry(8) = {(x,10) (f1,8) (f2,9) (f1,2) (f2,3) (n,1) (ans,7) (x,4) }
RDEntry(9) = {(x,10) (f1,8) (f2,9) (f2,3) (n,1) (ans,7) (x,4) }
RDEntry(10) = {(x,10) (f1,8) (f2,9) (n,1) (ans,7) (x,4) }
RDEntry(11) = {(x,10) (f1,8) (f1,2) (f2,9) (f2,3) (n,1) (ans,5) (ans,7) (x,4) }
RDExit:
RDExit(1) = {(ans,?) (x,?) (f2,?) (f1,?) (n,1) }
RDExit(2) = {(f2,?) (x,?) (ans,?) (f1,2) (n,1) }
RDExit(3) = {(ans,?) (x,?) (f1,2) (f2,3) (n,1) }
RDExit(4) = {(ans,?) (f1,2) (f2,3) (n,1) (x,4) }
RDExit(5) = {(f1,2) (f2,3) (n,1) (ans,5) (x,4) }
RDExit(6) = {(x,10) (f1,8) (f2,9) (f1,2) (f2,3) (n,1) (ans,7) (ans,5) (x,4) }
RDExit(7) = {(x,10) (f1,8) (f2,9) (f1,2) (f2,3) (n,1) (ans,7) (x,4) }
RDExit(8) = {(x,10) (f1,8) (f2,9) (f2,3) (n,1) (ans,7) (x,4) }
RDExit(9) = {(x,10) (f1,8) (f2,9) (n,1) (ans,7) (x,4) }
RDExit(10) = {(x,10) (f1,8) (f2,9) (n,1) (ans,7) }
RDExit(11) = {(x,10) (f1,8) (f1,2) (f2,9) (f2,3) (n,1) (ans,5) (ans,7) (x,4) }

Input Program with labels added:
int n;
int x;
int f1;
int f2;
int ans;
[n := 20]^1;
[f1 := 0]^2;
[f2 := 1]^3;
[x := 2]^4;
[ans := 0]^5;
while [x<=n]^6 do
[ans := f1+f2]^7;
[f1 := f2]^8;
[f2 := ans]^9;
[x := x+1]^10;
od
[write x]^11;

The program slice when point of interest 9 is: 1,2,3,4,6,7,8,9,10
```

Code 3.20 Output of Example Program

3.7.2. Benchmarks from peers

In this section, outputs of three benchmark programs are discussed. Each of benchmark highlights a design decision or a limitation of the implementation.

Benchmark 1

The first benchmark by Ibrahim Nemli, Kim Rostgaard Christensen and Peter Gammelgaard Poulsen is shown in Code 3.21. The slice output for the above program when point of interest is 5 is shown in Code 3.22. We have chosen to include the conditional statements like *if* and *while* that contain the statements in the slice. We have also chosen not to include the declaration statements in the slice. Hence, the output is different from the output mentioned in this benchmark program - which is int x and 1,5.

```
program
    int x;
    int y;
    int z;
    y := x;
    z := 1;
    while y>0 do
            z:= z*y;
            y:= y−1; (* point of interest *)
    od
    y:=0;
end
```

Code 3.21 Benchmark Program 1

```
Input Program with labels added:
int x;
int y;
int z;
[y := x]^1;
[z := 1]^2;
while [y>0]^3 do
[z := z*y]^4;
[y := y-1]^5;
od
[y := 0]^6;

The program slice when point of interest 5 is: 1,3,5
```

Code 3.22 Output of Benchmark Program 1

Benchmark 2

The slice output for the above program when point of interest is 7 is in Code 3.24. Our implementation does not parse brackets i.e () in read and write statement. Also, the parser expects every non boolean statement to end with a ‘;’ .Our parser is case sensitive 'While' is not accepted but 'while' is accepted. Hence, after making these changes in the given input program we get the output as 1,2,3,4,5,6,7,8,9. An important design consideration here is that if a variable occurs in the index expression of an array its value is not considered to be modified irrespective of whether it occurs on the right or left side of the assignment operator. Hence, the statements that affect those index variables will also be included in the program slice.

```
int x;
int y;
int A[20];
read y;
```

```
if y<10 then
x := 30;
else
x := 10;
 fi
y := 3;
while x<20 do
A[x] := x+y;
y := y*x;
x := x+1;
od
write y;
```

Code 3.23 Benchmark Program 2

```
Input Program with labels added:
int x;
int y;
int A[20];
[read y]^1;
if [y<10]^2 then
[x := 30]^3;
else
[x := 10]^4;
fi
[y := 3]^5;
while [x<20]^6 do
[A[x] := x+y]^7;
[y := y*x]^8;
[x := x+1]^9;
od
[write y]^10;

The program slice when point of interest 7 is: 1,2,3,4,5,6,7,8,9
```

Code 3.24 Output of Benchmark Program 2

4. 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.

4.1. 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.

4.1.1. Definition

The detection of sign analysis is defined as

$$(L, \mathcal{F}, F, E, \iota, f.)$$

where L is a complete lattice, $\mathcal{F}$ is a set of transfer functions, F is a finite flow $pg_{q_s}^{q_t}(S_\star)$, E is a finite set of extremal labels $\{init(S_\star)\}$, ι is an extremal value $\{0\}$ and $f.$ is a mapping from labels to transfer functions.

Lattice L

The complete lattice L is defined as

$$\widehat{State_S} = (Var_\star \cup Arr_\star \to Sign, \sqsubseteq)$$

where $Sign = (\mathcal{P}(\{-,0,+\}, \subseteq)$.

Mapping $f.^S$

The mapping $f.^S$ of labels to transfer functions is constructed as

$$\begin{aligned}
[x := a]^l: &\quad f_l^S(\hat{\sigma}) = \hat{\sigma}[x \mapsto \mathcal{A}_s[\![a]\!]\hat{\sigma}] \\
[skip]^l: &\quad f_l^S(\hat{\sigma}) = \hat{\sigma} \\
[A[a_1] := a_2]^l: &\quad f_l^S(\hat{\sigma}) = \hat{\sigma}[A \mapsto A \sqcup \mathcal{A}_s[\![a_2]\!]\hat{\sigma}] \\
[read\ x]^l: &\quad f_l^S(\hat{\sigma}) = \hat{\sigma}[x \mapsto \{0,-,+\}]
\end{aligned}$$

$$
\begin{aligned}
[read\ A[a]]^l: &\quad f_l^S(\hat{\sigma}) = \hat{\sigma}[A \mapsto \{0,-,+\}] \\
[write\ a]^l: &\quad f_l^S(\hat{\sigma}) = \hat{\sigma} \\
[b]^l: &\quad f_l^S(\hat{\sigma}) = \sqcup \{\hat{\sigma}' \in Atom(\hat{\sigma}) | tt \in \mathcal{B}_S[\![b]\!]\hat{\sigma}'\}
\end{aligned}
$$

$\mathcal{A}_s[\![a]\!]$

The $\mathcal{A}_s[\![a]\!]: \widehat{State_S} \to L$ which determines the signs of expressions is given by

$$
\begin{aligned}
\mathcal{A}_s[\![n]\!]\hat{\sigma} &= \begin{cases} \{-\} & if\ n < 0 \\ \{0\} & if\ n = 0 \\ \{+\} & if\ n > 0 \end{cases} \\
\mathcal{A}_s[\![x]\!]\hat{\sigma} &= \hat{\sigma}(x) \\
\mathcal{A}_s[\![a_1\ op_a\ a_2]\!]\hat{\sigma} &= \mathcal{A}_s[\![a_1]\!]\hat{\sigma}\ \widehat{op_a}\ \mathcal{A}_s[\![a_2]\!]\hat{\sigma} \\
\mathcal{A}_s[\![-x]\!]\hat{\sigma} &= \widehat{op_{ua}}\widehat{\mathcal{A}_s[\![x]\!]}\hat{\sigma} \\
\mathcal{A}_s[\![(x)]\!]\hat{\sigma} &= \hat{\sigma}(x)
\end{aligned}
$$

where $\widehat{op_a}: L \times L \to L$ which could be $\hat{+}, \hat{-}, \hat{\times}$ and $\hat{/}$ is specified by

$$\widehat{op_a}(S_1, S_2) = \{s_1\ op_a\ s_2 |\ s_1 \in S_1, s_2 \in S_2\}$$

and $\widehat{op_{ua}}: L \to L$ which could be the unary operand $\hat{-}$ only is specified by

$$\widehat{op_{ua}}(S_1) = \{op_{ua}\ s_1 |\ s_1 \in S_1\}$$

Finally, the op_a which could be $+, -, \times$ and /, and op_{ua} which could only be $-$ are defined in the Table 4.1 and Table 4.2.

Table 4.1 op_a

+	-	0	+	-	-	0	+
-	{-}	{-}	{-,0,+}	-	{-,0,+}	{-}	{-}
0	{-}	{0}	{+}	0	{+}	{0}	{-}
+	{-,0,+}	{+}	{+}	+	{+}	{+}	{-,0,+}
*	{-}	{0}	{+}	/	{-}	{0}	{+}
-	{+}	{0}	{-}	-	{+}	∅	{-}
0	{0}	{0}	{0}	0	{0}	∅	{0}
+	{-}	{0}	{+}	+	{-}	∅	{+}

Table 4.2 op_{ua}

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

$Atom(\hat{\sigma})$ and $\mathcal{B}_s[\![b]\!]$

The $Atom(\hat{\sigma})$ is a set of "small" abstract states that makes up $\hat{\sigma}$ and it is given by

$$Atom(\hat{\sigma}) = \{\hat{\sigma}' \sqsubseteq \hat{\sigma}\ | \forall x: |\hat{\sigma}'(x)| = 1\}$$

$\mathcal{B}_S: BExp \to (\widehat{State} \to \mathcal{P}(\{tt, ff\})$ is given by

$$\begin{aligned} \mathcal{B}_s[\![a_1\ op_r\ a_2]\!]\hat{\sigma}' &= \mathcal{A}_s[\![a_1]\!]\hat{\sigma}'\ \widehat{op_r}\ \mathcal{A}_s[\![a_2]\!]\hat{\sigma}' \\ \mathcal{B}_s[\![b_1\ op_b\ b_2]\!]\hat{\sigma}' &= \mathcal{B}_s[\![b_1]\!]\hat{\sigma}'\ \widehat{op_b}\ \mathcal{B}_s[\![b_2]\!]\hat{\sigma}' \\ \mathcal{B}_s[\![\neg b]\!]\hat{\sigma}' &= \{\neg t | t \in \mathcal{B}_s[\![b]\!]\hat{\sigma}'\} \end{aligned}$$

Finally, the $\widehat{op_r}$ and $\widehat{op_b}$ are defined with the tables below respectively.

Table 4.3 op_r

<	-	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 4.4 op_b

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

4.1.2. Discussion regarding correctness of the analysis

The lattice is defined as $\widehat{State_S} = (Var_\star \cup Arr_\star \to Sign, \sqsubseteq)$, since we are mapping all the variables together with arrays to $\mathcal{P}(\{-,0,+\})$. 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 $Atom(\hat{\sigma}) = \{\hat{\sigma}' \sqsubseteq \hat{\sigma} \mid \forall x: |\hat{\sigma}'(x)| = 1\}$ 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.

4.1.3. Proof

To prove that our model for detection of sign analysis is an instance of monotone framework, we need to prove that the functions $f_{\cdot}^{S}: \widehat{State_S} \rightarrow \widehat{State_S}$ are monotone. However, the functions $f_{\cdot}^{S}$ are monotone if the function determining the sign of expressions $\mathcal{A}_s[\![a]\!]: \widehat{State_S} \rightarrow Sign$ and the transfer function deciding the sign of arithmetic expressions in Boolean expressions $f_b(\hat{\sigma}) = \sqcup \{\hat{\sigma}' \in Atom(\hat{\sigma}) | tt \in \mathcal{B}_S[\![b]\!]\hat{\sigma}'\}$ are monotone.

1. First, we check that the functions $\mathcal{A}_s[\![a]\!]: \widehat{State_S} \rightarrow Sign$ are monotone.

For the case $\widehat{op_a}(S_1, S_2)$, suppose we have

$$\mathrm{A} \sqsubseteq B, \mathrm{A}, B \in \mathcal{P}(\{-,0,+\})$$

It is easy to verify that for all the possible values of $\widehat{op_a}$, there are

$$\widehat{op_a}(A, A) \sqsubseteq \widehat{op_a}(B, B)$$

For the case $\widehat{op_{ua}}(S_1)$, it is the same as the case $\widehat{op_a}(S_1, S_2)$.

Thus, we have proved that the functions $\mathcal{A}_s[\![a]\!]: \widehat{State_S} \rightarrow Sign$ are monotone.

2. Then we need to prove that $f_b(\hat{\sigma}) = \sqcup \{\hat{\sigma}' \in Atom(\hat{\sigma}) | tt \in \mathcal{B}_S[\![b]\!]\hat{\sigma}'\}$ is monotone.

Suppose we have $\widehat{\sigma_1} \sqsubseteq \widehat{\sigma_2}$, then for any variable x, we have $\widehat{\sigma_1}(x) \subseteq \widehat{\sigma_2}(x)$ and from there, it derives $Atom(\widehat{\sigma_1}) \subseteq Atom(\widehat{\sigma_2})$. By extracting all the elements $\hat{\sigma}'$ satisfying $tt \in \mathcal{B}_S[\![b]\!]\hat{\sigma}'$ in the set $Atom(\widehat{\sigma_1})$ and the set $Atom(\widehat{\sigma_2})$ respectively, we prove $f_b(\widehat{\sigma_1}) \subseteq f_b(\widehat{\sigma_2})$. Thus, we have proved that $f_b(\hat{\sigma})$ is monotone.

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

4.2. Algorithm for array bound checking for the lower bounds

This algorithm takes results from the program graph $pg_{q_s}^{q_t}(S_\star)$, the free variables in program under consideration and the transition function table as defined in section 4.2 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.

4.2.1. 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. The algorithm for doing this is shown in Code 4.1. 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 (q, B^l, q') are added to a worklist. Here, q is the starting node, q' the terminal node and B^l the block in the program that causes this state change. The constraints are initialized to extremal value which is $\emptyset$ for initial node and to the least element $\emptyset$ 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, B^l, 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.

4.2.2. 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 algorithm for doing this is shown in Code 4.2. The following steps are repeated for each edge (q, B^l, q') of the program graph. If the block B^l 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 B^l is added to set VL which is the output and contains all the blocks where boundary violation occurs.

Input: Program Graph, Transition function table, Free variables
Output: Solutions to Detection of Sign
W := nil;
For all (q, B^l, q') in $pg_{q_s}^{q_t}(S_\star)$ do
    W := cons$\left((q, B^l, q'), W\right)$;
For all q in Node($S_\star$) do
    If q ∈ init($S_\star$) then DS(q) :={(x, ∅)|x∈ FV($S_\star$) }
    else DS(q) := ∅
While W≠nil do
    (q, B^l,q'):= head(W);W=tail(W);
    If f_{B^l}(DS(q)) ⊈ DS(q'), x∈ FV($S_\star$)
        then DS(q') = DS(q')∪ f_{B^l}(DS(q)), x∈ FV($S_\star$)
        for all (q', B^l, q'') in $pg_{q_s}^{q_t}(S_\star)$ do
        W:= cons((q', B^l, q''),W);

Code 4.1 Algorithm for solving detection of sign constraints

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, B^l, q') in $pg_{q_s}^{q_t}(S_*)$ do
If B^l is a statement that includes an array as one of the free variables
    If DS(q') for index of the array contains {-}
        then cons((B^l),VL)

Code 4.2 Algorithm for array bound checking for lower bounds

4.3. 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 4.1. The places where the indexes of the arrays are to be visited are *Node 3* and *Node 5*.

Constraints for the aforementioned program are presented below:

DS (1) ⊒ [A ↦ {0}, B ↦ {0}, n ↦ {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 4.5 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*.

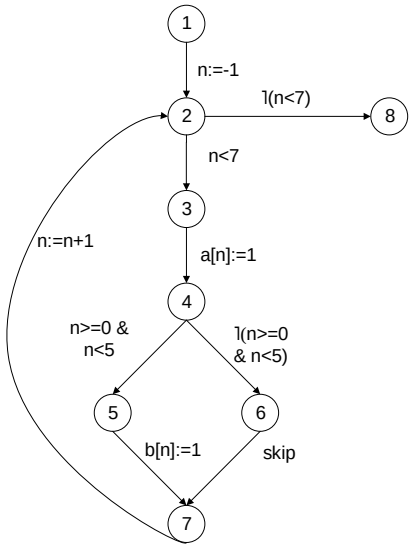

Figure 4.1 Program graph of the example program

Table 4.5 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,+}

4.4. 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 4.1.1. 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 4.2.1 and implementation of the algorithm for detection of array lower bound violation described in Section 4.2.2.

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.

4.5. Limitations and discussion regarding improvements

A complete implementation of analyses presented in Section 4.1.1 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 4.3. 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.

4.6. Benchmarking

This section presents results of benchmarking.

4.6.1. Benchmark 1

The example program illustrated in Code 4.3 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 4.5. The benchmark is provided by Andrius Andrijauskas and Lars Bonnicshen.

The output of our detection of signs analysis program is presented in Code 4.4. The array low boundary violation is detected correctly.

```
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 4.3 Benchmark 1program

```
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)
```

Code 4.4 Analysis result of benchmark 1

4.6.2. Benchmark 2

The example program presented in Code 4.5 is selected to show that our analysis of Boolean conditions works correctly and adds precision to the analysis. This benchmark is provided by Kamran Manzoor and Noel Vang.

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 in Code 4.6.

```
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 4.5 Benchmark 2 program

```
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)
```

Code 4.6 Analysis result of benchmark 2

4.6.3. Benchmark 3

The example program presented in Code 4.7 is selected to demonstrate two essential limitations of our detection implementation analysis. This benchmark is obtained from Tomasz Cezary Maciazek and Hugo Maxime Desjardins.

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 4.5, 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 4.5, 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
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 4.7 Benchmark 3 program

```
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)
```

Code 4.8 Analysis result of benchmark 3

5. Buffer overflow – interval analysis

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

5.1. Interval analysis

This subsection presents the definition of the interval analysis as an instance of the monotone framework for the project language followed by a comprehensive explanation of the correctness of the definition. Finally a proof is offered showing that the definition given is indeed an instance of the monotone framework.

5.1.1. Definition of interval analysis

The representation of interval analysis as an instance of the monotone framework is given by

$$(L, \mathcal{F}, F, E, \iota, f.)$$

where L is a complete lattice, $\mathcal{F}$ is a set of transfer functions, F is a finite edges $pg_{q_s}^{q_t}(S_\star)$ from a program graph, E is a finite set of extremal labels $\{init(S_\star)\}$, ι is an extremal value [0, 0] and $f.$ is a mapping from labels to transfer functions.

Lattice L

The complete lattice L is defined as

$$\widehat{State_I} = (Var_\star \cup Arr_\star) \to Interval, \sqsubseteq)$$

Where $Interval = \{\bot\} \cup \{\{z1, z2\} | z1 \leq z2, z_1 \in \boldsymbol{Z}' \cup \{-\infty\}\ and\ z_2 \in \boldsymbol{Z}' \cup \{\infty\}\}$
and $\{\boldsymbol{Z}' = z \mid min - 1 \leq z \leq max + 1\}$ [1] $and - \infty \leq \infty$.

The ordering $\sqsubseteq$ is given by $int_1 \sqsubseteq int_2 (int_1, int_2 \in Interval)$ if and only if $\beta(int_1) \subseteq \beta(int_2)$, where

$\beta(\bot) = \emptyset$
$\beta([z_1, z_2]) = \{z \in Z \mid z_1 \leq z_2\}\ if\ z_1 \in \boldsymbol{Z}' \cup \{-\infty\}\ and\ z_2 \in \boldsymbol{Z}' \cup \{\infty\}$
$\beta([-\infty, min - 1]) = \{z \mid z < min\}$
$\beta([max + 1, \infty]) = \{z \mid max < z\}$
and $-\infty \leq z\ and\ z \leq \infty$ holds for all $z \in \boldsymbol{Z}$

For the mapping from $Var_\star \cup Arr_\star$ to *Interval*, it is obvious that it is a complete lattice as 1) the possible values for $Var_\star$ and $Arr_\star$ in the program is infinite and $Interval$ is also infinite by definition; 2) for $A_1 \sqsubseteq A_2$ $(A_1, A_2 \in Var_\star \cup Arr_\star)$ and $int_1 \sqsubseteq int_2$ $(int_1, int_2 \in Interval)$, the relation $A_2[int_2] \sqsubseteq A_2[int_2]$ always holds as $A_2[int_2]$ always contains all the elements in$A_1[int_1]$ at least. Thus the lattice L is a complete lattice.

[1] *min* and *max* are two constant integers that could be specified. For buffer overflow analysis, the values of *min* and *max* could be assigned as 0 and the size of an array respectively.

Mapping $f_\cdot^S$

The mapping $f_\cdot^S$ of labels to transfer functions is constructed as

$$\begin{aligned}
[x := a]^l: &\quad f_l^I(\hat{\sigma}) = \hat{\sigma}[x \mapsto \mathcal{A}_I[\![a]\!]\hat{\sigma}] \\
[skip]^l: &\quad f_l^I(\hat{\sigma}) = \hat{\sigma} \\
[A[a_1] := a_2]^l: &\quad f_l^I(\hat{\sigma}) = \hat{\sigma}[A \mapsto A \sqcup \mathcal{A}_I[\![a_2]\!]\hat{\sigma}] \\
[read\ x]^l: &\quad f_l^I(\hat{\sigma}) = \hat{\sigma}[x \mapsto [-\infty, \infty]] \\
[read\ A[a]]^l: &\quad f_l^I(\hat{\sigma}) = \hat{\sigma}[A \mapsto [-\infty, \infty]] \\
[write\ a]^l: &\quad f_l^I(\hat{\sigma}) = \hat{\sigma} \\
[b]^l: &\quad f_l^I(\hat{\sigma}) = \sqcup \{\hat{\sigma}' \in Atom(\hat{\sigma}) | tt \in \mathcal{B}_I[\![b]\!]\hat{\sigma}'\}
\end{aligned}$$

$\mathcal{A}_I[\![a]\!]$

The $\mathcal{A}_I[\![a]\!]: \widehat{State}_I \to Interval$ which determines the interval for the value of an expression

$$\begin{aligned}
\mathcal{A}_I[\![n]\!]\hat{\sigma} &= \begin{cases} [n,n] & if\ min \le n \le max \\ [-\infty, min-1] & if\ n < min \\ [max+1, \infty] & if\ n > max \end{cases} \\
\mathcal{A}_I[\![x]\!]\hat{\sigma} &= \hat{\sigma}(x) \\
\mathcal{A}_I[\![a_1\ op_a\ a_2]\!]\hat{\sigma} &= \mathcal{A}_I[\![a_1]\!]\hat{\sigma}\ \widehat{op_a}\ \mathcal{A}_I[\![a_2]\!]\hat{\sigma} \\
\mathcal{A}_I[\![-x]\!]\hat{\sigma} &= \widehat{op_{ua}}\mathcal{A}_I[\![x]\!]\hat{\sigma} \\
\mathcal{A}_I[\![(x)]\!]\hat{\sigma} &= \hat{\sigma}(x)
\end{aligned}$$

where $\widehat{op_a}: Interval \times Interval \to Interval$ is an abstract operation on intervals which could be $\hat{+}, \hat{-}, \hat{\times}$ and $\hat{/}$. They are specified by

$$[z_{11}, z_{12}]\hat{+}[z_{21}, z_{22}] = chop([z_1, z_2]),$$

$$where\ z_i = \begin{cases} -\infty & if\ z_{1i} = -\infty\ or\ z_{2i} = -\infty \\ \infty & if\ z_{1i} = \infty\ or\ z_{2i} = \infty \\ -\infty & if\ z_{1i} + z_{2i} < min \\ \infty & if\ z_{1i} + z_{2i} > max \\ z_{1i} + z_{2i} & otherwise \end{cases}$$

$$[z_{11}, z_{12}]\hat{-}[z_{21}, z_{22}] = [z_{11}, z_{12}]\hat{+}[-z_{22}, -z_{21}]$$

$$\begin{aligned}
[z_{11}, z_{12}]\ \hat{\times}\ [z_{21}, z_{22}] = chop([&min(z_{11} \times z_{21},\ z_{11} \times z_{22},\ z_{12} \times z_{21},\ z_{12} \times z_{22}), \\
&max(z_{11} \times z_{21},\ z_{11} \times z_{22},\ z_{12} \times z_{21},\ z_{12} \times z_{22})]),
\end{aligned}$$

$$where\ \ z_{ij} \times z_{kl} = \begin{cases} 0 & if\ z_{ij} = 0\ or\ z_{kl} = 0 \\ \infty & if \begin{pmatrix} signs\ of\ z_{ij}\ and\ z_{kl}\ are\ same\ and\ at \\ least\ one\ of\ them\ is\ either - \infty\ or\ \infty \end{pmatrix} \\ -\infty & if \begin{pmatrix} signs\ of\ z_{ij}\ and\ z_{kl}\ are\ distinct\ and\ at \\ least\ one\ of\ them\ is\ either - \infty\ or\ \infty \end{pmatrix} \\ \infty & if\ z_{ij} \times z_{kl} > max \\ -\infty & if\ z_{ij} \times z_{kl} < min \\ z_{ij} \times z_{kl} & otherwise \end{cases}$$

$$\begin{aligned}
[z_{11}, z_{12}]\hat{/}[z_{21}, z_{22}] = chop([&min(z_{11}/z_{21},\ z_{11}/z_{22},\ z_{12}/z_{21},\ z_{12}/z_{22}), \\
&\max(z_{11}/z_{21},\ z_{11}/z_{22},\ z_{12}/z_{21},\ z_{12}/z_{22})]),
\end{aligned}$$

$$where\ z_{ij}/z_{kl} = \begin{cases} \bot & if\ 0 \in [z_{21}, z_{22}] \\ 1 & if\ z_{ij} = z_{kl} = -\infty\ or\ z_{ij} = z_{kl} = \infty \\ -1 & if\ (z_{ij} = \infty, z_{kl} = -\infty) or\ (z_{ij} = -\infty, z_{kl} = \infty) \\ 0 & if\ z_{ij} = 0\ or\ z_{kl}\ is - \infty\ or\ \infty \\ z_{ij} & if\ z_{ij}\ is\ -\infty\ or\ \infty \\ \infty & if\ z_{ij}/z_{kl} > max \\ -\infty & if\ z_{ij}/z_{kl} < min \\ z_{ij}/z_{kl} & otherwise \end{cases}$$

as well as

$$\bot\ \widehat{op_a}\ int = int\ \hat{+}\bot = \bot\ for\ all\ int \in Interval$$

and $\widehat{op_{ua}}: Interval \to Interval$ which could be the unary operand $\hat{-}$ only is specified by

$$\hat{-}[z_1, z_2] = chop([-z_2, -z_1])$$

Similarly, we have

$$\widehat{op_{ua}}\ \bot = \bot$$

Finally the $chop$ function is defined by

$$chop([a,b]) = \begin{cases} [a,b] & if\ min \le a, b \le max \\ [-\infty, min-1] & if\ b < min \\ [max+1, \infty] & if\ a > max \\ [-\infty, b] & if\ b \le max, a < min \\ [a, \infty] & if\ a \ge min, b > max \\ [-\infty, \infty] & if\ a < min, b > max \end{cases}$$

$Atom(\hat{\sigma})$ and $\mathcal{B}_I[\![b]\!]$

The $Atom(\hat{\sigma})$ is a set of "small" abstract states that makes up $\hat{\sigma}$ and it is given by

$$Atom(\hat{\sigma}) = \{\hat{\sigma}' \sqsubseteq \hat{\sigma}\ |\forall x: |\hat{\sigma}'(x)| = 1\}$$

$\mathcal{B}_I: BExp \to (\widehat{State} \to \mathcal{P}(\{tt, ff\})$ is given by

$$\begin{array}{rcl} \mathcal{B}_I[\![a_1\ op_r\ a_2]\!]\hat{\sigma}' & = & \mathcal{A}_I[\![a_1]\!]\hat{\sigma}'\ \widehat{op_r}\ \mathcal{A}_I[\![a_2]\!]\hat{\sigma}' \\ \mathcal{B}_I[\![b_1\ op_b\ b_2]\!]\hat{\sigma}' & = & \mathcal{B}_I[\![b_1]\!]\hat{\sigma}'\ \widehat{op_b}\ \mathcal{B}_I[\![b_2]\!]\hat{\sigma}' \\ \mathcal{B}_I[\![\neg b]\!]\hat{\sigma}' & = & \{\neg t | t \in \mathcal{B}_I[\![b]\!]\hat{\sigma}'\} \end{array}$$

Finally, the $\widehat{op_r}$ and $\widehat{op_b}$ are defined with the tables below respectively. One thing need to be point out is that $\infty! = \infty$ and $-\infty! = -\infty$.

$$[z_{11}, z_{12}] \hat{<} [z_{21}, z_{22}] = \begin{cases} \{tt\} & if\ z_{12} < z_{21} \\ \{ff\} & if\ z_{22} \le z_{11} \\ \{tt, ff\} & otherwise \end{cases}$$

$$[z_{11}, z_{12}] \hat{>} [z_{21}, z_{22}] = \begin{cases} \{tt\} & if\ z_{11} > z_{22} \\ \{ff\} & if\ z_{21} \geq z_{12} \\ \{tt, ff\} & otherwise \end{cases}$$

$$[z_{11}, z_{12}] \widehat{<=} [z_{21}, z_{22}] = \begin{cases} \{tt\} & if\ z_{12} \leq z_{21} \\ \{ff\} & if\ z_{22} < z_{11} \\ \{tt, ff\} & otherwise \end{cases}$$

$$[z_{11}, z_{12}] \widehat{>=} [z_{21}, z_{22}] = \begin{cases} \{tt\} & if\ z_{11} \geq z_{22} \\ \{ff\} & if\ z_{21} > z_{12} \\ \{tt, ff\} & otherwise \end{cases}$$

$$[z_{11}, z_{12}] \hat{=} [z_{21}, z_{22}] = \begin{cases} \{tt\} & if\ z_{11} = z_{12} = z_{21} = z_{22} \\ \{ff\} & if\ \{x | z_{11} \leq x \leq z_{12}\} \cap \{x | z_{21} \leq x \leq z_{22}\} = \emptyset \\ \{tt, ff\} & otherwise \end{cases}$$

$$[z_{11}, z_{12}] \widehat{!=} [z_{21}, z_{22}] = \begin{cases} \{ff\} & if\ z_{11} = z_{12} = z_{21} = z_{22} \\ \{tt\} & if\ \{x | z_{11} \leq x \leq z_{12}\} \cap \{x | z_{21} \leq x \leq z_{22}\} = \emptyset \\ \{tt, ff\} & otherwise \end{cases}$$

Table 5.1 op_b

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

5.1.2. Discussion regarding correctness of the analysis

This subsection presents the correctness of the analysis presented above.

To achieve interval analysis, variables (and arrays) are mapped to intervals for every label in program. Thus the lattice is defined as $\widehat{State_l} = (Var_\star \cup Arr_\star) \rightarrow Interval, \sqsubseteq$ in which the elements in an array are treated as a whole. The interval is defined as a pair. The pair embraces 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. The value of a variable is bounded to be a number between *min* and *max*. The numbers that are greater than *max* are recorded as $[max + 1, \infty]$ and the numbers that are less than *min* are coded as $[-\infty, min - 1]$. By using this definition, the following interval representations are illegal in our model: $[-\infty, -\infty]$, $[\infty, \infty]$.

For each label in the program, a specific transfer function is designed. The mapping from labels to statements is one-to-one except for *while* statements and *if* statements in which only Boolean expression is considered. In the rest of this subsection, we will go through all the mappings from labels to transfer functions and discuss the correctness of each transfer function of them one by one.

Variable assignment For each assignment, the interval of the variable on the left side of the equation is derived by calculating the interval of 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 computing the sum of two intervals, say $[z_{11}, z_{12}]$ and $[z_{21}, z_{22}]$, the lower boundary and upper boundary are computed separately. When computing each boundary, first the special cases where the plus operators contain infinity values ($-\infty$ or ∞) are considered. If the plus operands comprises infinity values or the plus result of z_{1i} and z_{2i} 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 $[-\infty, -\infty]$ and $[\infty, \infty]$ 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 addition, we get an illegal interval $[-\infty, -\infty]$. To compensate this, the *chop* function is introduced.

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 $Interval = \{\perp\} \cup \{\{z1, z2\} | z1 \leq z2, z_1 \in \boldsymbol{Z'} \cup \{-\infty\} \; and \; z_2 \in \boldsymbol{Z'} \cup \{\infty\}\}$ and $\{\boldsymbol{Z'} = z \mid min - 1 \leq z \leq max + 1\}$.

Array assignment As an array is a series of variables, when mapping an new interval to an array, the existing interval assigned to the array should also be included in the new interval. Thus the union of the existing interval and the new interval (which is computed by the arithmetical expression on the right of the array assignment) is set as the new interval for the array. The union of two intervals, say $[z_{11}, z_{12}]$ and $[z_{21}, z_{22}]$ is computed as $[\min(z_{11}, z_{21}), \max(z_{12}, z_{22})]$. By doing this, we lose some preciseness in our model but it makes sure that 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 fact that the statement is just either doing nothing or showing results to users.

Read statements For the read statement of a variable or an array, as no pre-knowledge of what value the user would provide is available during static analysis, the interval of the variable or array involved is simply assigned as all the possible numbers, i.e. $[-\infty, \infty]$.

Boolean statement As we want to benefit from using program graph in analyzing to make the result more precise, we define $Atom(\hat{\sigma}) = \{\hat{\sigma}' \sqsubseteq \hat{\sigma} \mid \forall x: |\hat{\sigma}'(x)| = 1\}$ 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 $Atom(\hat{\sigma})$ splits an interval into "small" intervals. For example, suppose we have the Boolean condition *a>1* and the interval mapped to *a* is [1, 2]. When comparing the two operands in the Boolean condition, the interval of the left operand *a* is split into [1, 1] and [2, 2]. The same goes for the right operand. However as the interval of an integer only contains itself, the interval

of the integer is [1, 1]. Then the result of the Boolean expression [1, 1] > [1, 1] and [2, 2] > [1, 1] is computed following the rules defined for $\widehat{op_r}$. That is the intervals resulting in {*tt*} or {*tt, ff*}, i.e. [2, 2] for *a*. If the result is made up with more than one interval, the union of them is set as the new Interval of the variable after the Boolean condition. However, again, by doing union, we might lose some precision in the result.

Based on the discussion given above, considering that our design complies with conventional rules, we argue that the presented interval analysis is correct.

5.1.3. Proof

To prove that the interval analysis given by us is an instance of monotone framework, we need to prove that the functions $f_\cdot^I: \widehat{State}_I \to \widehat{State}_I$ are monotone. Similar to the proof of sign detection, the functions $f_\cdot^I$ are monotone if the function determining the interval of expressions $\mathcal{A}_I[\![a]\!]: \widehat{State}_I \to Interval$ and the transfer function deciding the interval of arithmetic expressions in Boolean expressions $f_b(\hat{\sigma}) = \sqcup \{\hat{\sigma}' \in Atom(\hat{\sigma}) | tt \in \mathcal{B}_I[\![b]\!]\hat{\sigma}'\}$ are monotone.

1) Firstly we check that the functions $\mathcal{A}_I[\![a]\!]: \widehat{State}_I \to Interval$ are monotone.

For the case $\widehat{op_a}$, suppose we have

$$I_1 \sqsubseteq I_2, I_1' \sqsubseteq I_2', I_1, I_2, I_1', I_2' \in Interval$$

It is easy to verify that for $\hat{+}$ ($or\ \hat{-}$), there are

$$I_1 \hat{+} I_1' \sqsubseteq I_2 \hat{+} I_2' \ (or\ I_1 \hat{-} I_1' \sqsubseteq I_2 \hat{-} I_2')$$

Then for $\hat{\times}$ ($or\ \hat{/}$), suppose $I_i \hat{\times} I_i' = [x_i, y_i], i = 1,2$, where x_i is the minimum value after multiplying the two *Interval* I_i and I_i', and y_i is the maximum. Without losing any generality, let's assume $y_1 = z_{11} \times z_{12}$ in which $z_{11}, z_{12}$are two numbers in $I_1 \cup I_1'$. Then for z_{11}, we can always find an integer z_{21} in I_2 or I_2' which is greater than (less than if z_{11} is negative) or equal to z_{11}. It is the same for z_{12}. In other words, we could always find two integers in I_2 and I_2' whose absolute values are greater than or equal to the absolute values of z_{11} and z_{12} correspondingly. As the upper boundary of an interval after multiplication could only be a non-negative integer by definition, $y_1 \leq y_2$ always holds. The same for the least bound x_i. Thus we have proved $I_1 \hat{\times} I_1' \sqsubseteq I_2 \hat{\times} I_2'$. Finally, The same goes for $\hat{/}$.

For the case $\widehat{op_{ua}}$, it is very obvious that for $I_1 \sqsubseteq I_2, I_1, I_2 \in Interval$, $\hat{-} I_1 \sqsubseteq \hat{-} I_2$ always holds.

Thus, we have proved that the functions $\mathcal{A}_I[\![a]\!]: \widehat{State}_I \to Interval$ are monotone.

2) Then we need to prove that $f_b(\hat{\sigma}) = \sqcup \{\hat{\sigma}' \in Atom(\hat{\sigma}) | tt \in \mathcal{B}_I[\![b]\!]\hat{\sigma}'\}$ 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.

5.2. Algorithms for array boundary checking

This algorithm takes results from program graph $pg_{q_s}^{q_t}(S_\star)$, free variables in program and transition function table 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:

5.2.1. Algorithm for solving interval analysis equations

The algorithm for solving interval analysis equations is shown in Code 5.1. The first step in the algorithm is the 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 (q, B^l, q') are added to a worklist. Here, q is the starting node, q' the terminal node and B^l the block in the program that causes this state change. The equations are initialized to extremal value which is [0, 0] for initial node and to least element $\perp$ 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, B^l, 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 $I(q')$ 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 the definition presented in previous subsection. 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.

5.2.2. Algorithm for array bound checking

The second stage of the algorithm for bound checking (Code 5.2) takes in all edges in the program graph and solutions of interval analysis as input. Following steps are repeated for each edge (q, B^l, q') of the program graph. If block B^l of an edge has an array element as one of the free variable and if the solution for that variable at the target node q' intersect $)[-\infty, min-1]$ or $[max+1, \infty]$ not equals to empty for index variable of the array, then the value of array index variable has violated the maximum or minimum array bound. Thus array bound violation is detected in a block B^l and if detected B^l is added to output set *VL* containing all the blocks where boundary violation occurs.

Input: Program Graph, Transition function table, Free variables
Output: Solutions to Interval analysis
W := nil;
For all (q, B^l, q') in $pg_{q_s}^{q_t}(S_\star)$ do
    W := cons$\left((q, B^l, q'), W\right)$;
For all q in Node($S_\star$) do
    If q ∈ init($S_\star$) then I(q) :={(x, ∅)|x∈ FV($S_\star$) }

```
        else I(q) := ∅
While W≠nil do
        (q, B^l,q'):= head(W);W=tail(W);
        If f_{B^l}( I(q)), ⊈ I(q'), x∈ FV(S_*)
                then I(q') = I(q')∪ f_{B^l}( I(q)), x∈ FV(S_*)
                for all (q', B^l, q'') in pg^{q_t}_{q_s}(S_*) do
                W:= cons((q', B^l, q''),W);
```

Code 5.1 Algorithm for solving interval analysis equations

```
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, B^l, q') in pg^{q_t}_{q_s}(S_*) do
If B^l is a statement that includes an array as one of the free variables
    If I(q') for index of the array I(q')∩ [−∞, min − 1] ≠ ∅or I(q') ∩ [max + 1, ∞] ≠ ∅
        then cons((B^l),VL)
```

Code 5.2 Algorithm for array bound checking using interval analysis

5.3. Implementation

In the implementation, we've implemented all mappings and transfer functions except for $Atom(\hat{\sigma}) = \{\hat{\sigma}' \sqsubseteq \hat{\sigma} \mid \forall x: |\hat{\sigma}'(x)| = 1\}$ as it is too computation expensive. However we do implement $Atom(\hat{\sigma})$ for relation expressions that consist of one variable and one integer as the operands. For other cases, the whole interval is used directly in relational operations.

The source code for the interval analysis is managed in the package *interval_analysis*. The main class for interval analysis is *IntervalAnalysis*. It provides several static methods to perform analysis, printing solution table and printing violation edges. The static *analyze* method in the *IntervalAnalysis* class requires the interested values for *min* and *max*, the program graph and free variables as method invocation inputs. This *analyze* method then makes use of *IAWorkList* class to compute the solution table for each variable and each label on the program graph. Finally, the violation edges is detected based on the interval value in the solution table.

The main structure of the *IAWorkList* class is the same as the *DSWorkList* class for detection of signs. The only difference is how the transfer functions are provided. In interval analysis, the transfer functions are implemented as a factory class *TransferFunctionFactory* where the mapping between labels to transfer functions and the computation of intervals for variables is implemented. The *create* method in the *TransferFunctionFactory* takes an edge from the program graph as input and returns the updated solution table for the label on the end node of the edge. Finally, the *Interval*, *ArithInterval* and *BoolInterval* classes implements the basic arithmetical and relational operations between intervals.

5.4. Solutions of benchmark tests

In this part of the report, we discuss the solutions by hand and by program for the example program presented in the beginning of detection of signs analysis. In addition, the evaluation results of some selected benchmark tests provided by peers are also presented.

5.4.1. Constraints and solutions for the example program

For the interval 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 of the example program is presented in Figure 5.1.

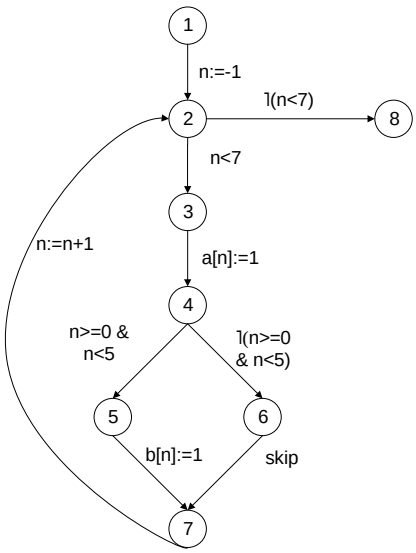

Figure 5.1. Program graph of the example program

Constraints for the aforementioned program are presented below.

I (1) ⊒ [a ↦ {0}, b↦{0}, n ↦ {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))

In the solutions for the constraints, only the variable *n* is important since we are concerned of the buffer overflow analysis and *n* is the only variable in the program used as array index. Table 5.2 shows constraints solutions.

The program computation result complies with our expected results and a violation is detected for the assignment of array *a*. (Code 5.3)

Table 5.2 Solutions to constraints for example program by hand

I(label)	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]

Interval analysis solution table:
1: b[0, 0] a[0, 0] n[0, 0]

```
2: b[0, 1] a[0, 1] n[-INF, +INF]
3: b[0, 1] a[0, 1] n[-INF, +INF]
4: b[0, 1] a[0, 1] n[-INF, +INF]
5: b[0, 1] a[0, 1] n[0, 4]
6: b[0, 1] a[0, 1] n[-INF, +INF]
7: b[0, 1] a[0, 1] n[-INF, +INF]
8: b[0, 1] a[0, 1] n[5, +INF]
Array indexing violation found:
(3, a[n] := 1;, 4)
```

Code 5.3 Solutions to constraints for example program by program

5.4.2. Selected benchmark tests

We also run our program with some of the benchmark tested given by peers. Some results and analysis are provided here.

Benchmark 1 The benchmark shown in Code 5.4 is provided by Jens Schønberg with labeling information added as comments in his source code. This benchmark test is used to show that our result analyzes the array boundary indexing correctly.

We perform the interval analysis by setting the program inputs *min* = 0 and *max* = 7 as the size of the array interested is 8. Our program successfully detected the violation for the array indexing and the result is illustrated in Code 5.5. The solution table also shows that before entering the *while* loop, the value of *j* could be 2 to plus infinity, however within the loop, the value of *j* becomes more precise as the value of it is re-assigned using *j=j*2;* After exiting from the loop, the value of *j* includes 2 and 3 again as the program takes into account the possibility that the *while* loop is never executed.

```
program
int a;
int i;
int j;
int B[8];
read(a);              (*label 1*)
i:=0;                 (*label 2*)
j:=2;                 (*label 3*)
while(i <= 8) do      (*label 4*)
    j:=j*2;           (*label 5*)
    B[i]:=a/j;        (*label 6*)
    i:=i+1;           (*label 7*)
od
write(B);             (*label 8*)
end

(* Code goes out-of-bounds at the 8th iteration of the while-loop *)
```

Code 5.4 Benchmark 1

```
Program graph:
(1,read a;,2), (2,i := 0;,3), (3,j := 2;,4), (4,i<=8,5), (5,j := j*2;,6), (6,B[i] := a/j;,7), (7,i := i+1;,4),
(4,!i<=8,8), (8,write B;,9)

Interval analysis solution table:
1: B[0, 0] a[0, 0] j[0, 0] i[0, 0]
2: B[0, 0] a[-INF, +INF] j[0, 0] i[0, 0]
3: B[0, 0] a[-INF, +INF] j[0, 0] i[0, 0]
4: B[-INF, +INF] a[-INF, +INF] j[2, +INF] i[0, +INF]
```

```
5: B[-INF, +INF] a[-INF, +INF] j[2, +INF] i[0, +INF]
6: B[-INF, +INF] a[-INF, +INF] j[4, +INF] i[0, +INF]
7: B[-INF, +INF] a[-INF, +INF] j[4, +INF] i[0, +INF]
8: B[-INF, +INF] a[-INF, +INF] j[2, +INF] i[8, +INF]
9: B[-INF, +INF] a[-INF, +INF] j[2, +INF] i[8, +INF]
Array indexing violation found:
(6, B[i] := a/j;, 7)
```

Code 5.5 Solution to benchmark 1

Benchmark 2 The Benchmark presented in Code 5.6 is provided by Tomasz Cezary Maciazek and Hugo Maxime Desjardins. This example demonstrates the high preciseness and correctness of our model as well as some limitations in our program.

The analysis result generated by the program is shown in Code 5.7. By comparing our computation result with the result provided by the benchmark authors, we could conclude that our result is more precise than the result given. This is probably due to the fact that we take the Boolean condition results in to consideration for different program graph branches. Based on the authors' suggestion, the violation should be detected at label 8, that is *A[5-x] := 2*. This is true as the interval of the arithmetical expression *5-x* is *[3, +INF]*. However due to the limitation of the implemented boundary detection algorithm, our program only detects the buffer overflow/underflow in array indices which are made of a variable or an integer. But we found another violation that is neglected by the author in the array assignment *A[y] := x*. This violation really exists as the lower boundary of variable *y* is always less than *x* by 1. During the execution where the *x* becomes 0 at label 6, *y* becomes -1 in the previous block.

```
program
int x;
int y;
int A[5];
x := 3;                          (*label 1 x [3,3], y [0,0], A [0,0]*)
y := 2;                          (*label 2 x [3,3], y [2,2], A [0,0]*)
while x > 0 do                   (*label 3 x [3,3], y [2,2], A [0,0]*)
    y := y - 1;                  (*label 4 x [-inf,3], y [-inf,2], A [-inf,4]*)
    x := x - 1;                  (*label 5 x [-inf,2], y [-inf,1], A [-inf,4]*)
    A[y] := x;                   (*label 6 x [-inf,2], y [-inf,1], A [-inf,4]*)
    A[x] := x + 2;               (*label 7 x [-inf,2], y [-inf,1], A [-inf,4]*)
    A[5-x] := 2;                 (*label 8 x [-inf,2], y [-inf,1], A [-inf,4]*)
od
skip;                            (*label 9 x [-inf,2], y [-inf,1], A [-inf,4]*)
end
(*The Interval Analysis program should return label 8.*)
```

Code 5.6 Benchmark 2

```
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)
Interval analysis solution table:
1: A[0, 0] y[0, 0] x[0, 0]
2: A[0, 0] y[0, 0] x[3, 3]
3: A[0, 4] y[-INF, 2] x[0, 3]
4: A[0, 4] y[-INF, 2] x[1, 3]
5: A[0, 4] y[-INF, 1] x[1, 3]
6: A[0, 4] y[-INF, 1] x[0, 2]
7: A[0, 4] y[-INF, 1] x[0, 2]
8: A[0, 4] y[-INF, 1] x[0, 2]
```

```
9: A[0, 4] y[-INF, 2] x[0, 0]
10: A[0, 4] y[-INF, 2] x[0, 0]
Array indexing violation found:
(6, A[y] := x;, 7)
```

Code 5.7 Solution to benchmark 2

5.5. Discussion on precision of analysis and improvement

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

$$\widehat{State}_I = (Var_\star \cup (Arr_\star \times Interval)) \to Interval, \sqsubseteq)$$

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. Another solution could be treating each element in the array separately, however this requires that the array size is known beforehand.

Besides the loss of precision introduced by array variables, it could also result from the union operation, arithmetical computations that go beyond boundaries, etc.

6. Security analysis

Example program for buffer overflow:

```
program
high int x;
low int y;
low int z;
low int k;
read x;
while x!=1 do
      y := 1;
      x := x - 1;
od
z := z*k;
write z;
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*. However, statement *write z* does not leak information.

6.1. Definition

This subsection addresses the definition of the security analysis for the project language. The security analysis is defined as

$$(L, \mathcal{F}, F, E, \iota, f_\cdot)$$

where L is a complete lattice, $\mathcal{F}$ is a set of transfer functions, F is a finite flow $pg_{q_s}^{q_t}(S_\star)$, E is a finite set of extremal labels $\{init(S_\star)\}$, ι is an extremal value $\{low\}$ and $f_\cdot$ is a mapping from labels to transfer functions.

Lattice L

The complete lattice L is defined as

$$\widehat{State_{SL}} = Var_\star \cup Arr_\star \cup \{ctx\} \to (\mathcal{P}(\{high, low\})$$

where *ctx* security level of the context.

Mapping $f_\cdot^{SL}$

The mapping $f_\cdot^{SL}$ of labels to transfer functions is constructed as

$$\begin{aligned}
[x := a]^l: &\quad f_l^{SL}(\hat{\sigma}) = \begin{cases} \hat{\sigma} \ if\ ctx\ and\ A_{SL}[\![a]\!] = \{low\} \\ \hat{\sigma}[x \mapsto \{high\}] if\ ctx\ or\ A_{SL}[\![a]\!] = \{high\} \end{cases} \\
[skip]^l: &\quad f_l^{SL}(\hat{\sigma}) = \hat{\sigma} \\
[A[a_1] := a_2]^l: &\quad f_l^{SL}(\hat{\sigma}) = \begin{cases} \hat{\sigma} \ if\ ctx\ and\ A_{SL}[\![a_2]\!] = \{low\} \\ \hat{\sigma}[A \mapsto \{high\}] if\ ctx\ or\ A_{SL}[\![a_2]\!] = \{high\} \end{cases} \\
[read\ x]^l: &\quad f_l^{SL}(\hat{\sigma}) = \hat{\sigma} \\
[read\ A[a]]^l: &\quad f_l^{SL}(\hat{\sigma}) = \hat{\sigma}
\end{aligned}$$

$$[write\ a]^l: \quad f_l^{SL}(\hat{\sigma}) = \hat{\sigma}$$
$$[b]^l: \quad f_l^{SL}(\hat{\sigma}) = \hat{\sigma}[ctx \mapsto \{high\}]\ if\ \ B[\![b]\!] = \{high\}$$

$\mathcal{A}_{SL}[\![a]\!]$

The $\mathcal{A}_{SL}[\![a]\!]: \widehat{State}_{SL} \to L$ which determines the signs of expressions is given by

$$\begin{aligned}
\mathcal{A}_{SL}[\![n]\!]\hat{\sigma} &= \{low\} \\
\mathcal{A}_{SL}[\![x]\!]\hat{\sigma} &= \hat{\sigma}(x) \\
\mathcal{A}_{SL}[\![a_1\ op_a\ a_2]\!]\hat{\sigma} &= \mathcal{A}_{SL}[\![a_1]\!]\hat{\sigma}\ \widehat{op_a}\ \mathcal{A}_{SL}[\![a_2]\!]\hat{\sigma} \\
\mathcal{A}_{SL}[\![-x]\!]\hat{\sigma} &= \hat{\sigma}(x) \\
\mathcal{A}_{SL}[\![(x)]\!]\hat{\sigma} &= \hat{\sigma}(x)
\end{aligned}$$

where $\widehat{op_a}: L \times L \to L$ which could be $\hat{+}, \hat{-}, \hat{\times}$ and $\hat{/}$ is specified by

$$\widehat{op_a}(S_1, S_2) = \{s_1\ op_a\ s_2 |\ s_1 \in S_1, s_2 \in S_2\}$$

Finally, the op_a which could be $+, -, \times$ and /, and op_{ua} which could only be $-$ are defined in the Table 4.1.

Table 6.1 op_a

op_a	low	high
low	{low}	{high}
high	{high}	{high}

$\mathcal{B}_{SL}[\![b]\!]$

$\mathcal{B}_{\mathrm{SL}}: BExp \to (\widehat{State} \to \mathcal{P}(\{low, high\})$ is given by

$$\begin{aligned}
\mathcal{B}_{SL}[\![a_1\ op_r\ a_2]\!]\hat{\sigma}' &= \mathcal{A}_{SL}[\![a_1]\!]\hat{\sigma}'\ \widehat{op_r}\ \mathcal{A}_s[\![a_2]\!]\hat{\sigma}' \\
\mathcal{B}_{SL}[\![b_1\ op_b\ b_2]\!]\hat{\sigma}' &= \mathcal{B}_{SL}[\![b_1]\!]\hat{\sigma}'\ \widehat{op_b}\ \mathcal{B}_{SL}[\![b_2]\!]\hat{\sigma}' \\
\mathcal{B}_{SL}[\![\neg b]\!]\hat{\sigma}' &= \mathcal{B}_{SL}[\![b]\!]\hat{\sigma}'
\end{aligned}$$

Finally, the $\widehat{op_r}$ and $\widehat{op_b}$ are defined with the tables below respectively.

Table 6.2 $\mathrm{op_r}$

$\mathrm{op_r}$	low	high
low	{low}	{high}
high	{high}	{high}

Table 6.3 op_b

&	low	high
low	{low}	{high}
high	{high}	{high}

The algorithms and implementation part are not presented in the report since the security analysis has the same structure as detection of signs analysis with the difference that that security level is always single for any variable. Therefore, these sections are omitted.

6.2. Limitations and discussion regarding improvements

The program for security analysis is implemented in a way that the context security level is remembered before it can be modified in the boolean condition and after the end of the statement containing the conditional test the security context is returned to the level which was before the this conditional statement. This feature provides improvements to precision which are shown in the next section by the execution of our test program. However, we did not come up with theoretical explanation of this feature. That is why it is not expressed in the definition.

Moreover, the security analysis is limited due to implemented advanced feature, thus, nested conditional statements are not handled.

The improvement would be to implement handling of nested conditional statements.

6.3. Benchmarking

6.3.1. Benchmark 1

This example program is our test program which shows the advantages of our implementation. The program is presented in Code 6.1. Since the security context is reduced to the low level after the exit from the while loop, the security level of the z variable is kept low. That is why the only possible information leakage is in write y statement, because the security level of y is raised to high due to high context.

The detailed output of our analysis is presented in Code 6.2.

```
program
high int x;
low int y;
low int z;
low int k;
read x;
while x!=1 do
y := 1;
x := x - 1;
od
z := z*k;
write z;
write y; (* security violation *)
end
```

Code 6.1 Benchmark 1program

```
Program graph:
(1,read x;,2), (2,x!=1,3), (3,y := 1;,4), (4,x := x-1;,2), (2,!x!=1,5), (5,z := z*k;,6), (6,write z;,7), (7,write y;,8)

Security Level Analysis solutions table 19:
1: ctx={low}    z={low}    k={low}    y={low}    x={high}
2: ctx={high}   z={low}    k={low}    y={high}   x={high}
3: ctx={high}   z={low}    k={low}    y={high}   x={high}
4: ctx={high}   z={low}    k={low}    y={high}   x={high}
5: ctx={low}    z={low}    k={low}    y={high}   x={high}
6: ctx={low}    z={low}    k={low}    y={high}   x={high}
7: ctx={low}    z={low}    k={low}    y={high}   x={high}
8: ctx={low}    z={low}    k={low}    y={high}   x={high}
```

```
Security level violations:
(7,write y;,8)
```

Code 6.2 Analysis result of benchmark 1

6.3.2. Benchmark 2

The example program below is selected to show limitations of our implementation. Our system does not detect possible security violation in the while loop, where a number is output by write(0) statement. However the rest 3 security violations are detected. Moreover, the last statement write(h3) does not violates the security policy since the security level of the variable h3 is downgraded by the assignment of the low security level data.

The benchmark is provided by Andrius Andrijauskas and Lars Bonnicshen and presented in Code 6.3. The detailed output of our program is presented in Code 6.4.

```
program
        high int h1;
         high int h2;
        high int h3;
        low int l1;
         low int l2;
         low int l3;
        l3 := l1 + h2; (* l3 is now HIGH *)
        h3 := 2 + l1; (* h3 is now LOW *)
        if h2 = 0 then
                l2 := 5; (* l2 is now HIGH *)
        else
                l2 := 9; (* l2 is now HIGH *)
        fi
        while h1 != l1 do
                write(0); (* Warn execution depends on HIGH expression *)
                l1 := l1 + 1; (* l1 is now HIGH *)
        od
        write(l2); (* Warn l2 is HIGH*)
        write(l3); (* Warn l3 is HIGH*)
        write(h2);  (* Warn h2 is HIGH *)
        write(h3);
end
```

Code 6.3 Benchmark 1program

```
Program graph:
(1,l3 := l1+h2;,2), (2,h3 := 2+l1;,3), (3,h2=0,4), (3,!h2=0,5), (4,l2 := 5;,6), (5,l2 := 9;,6), (6,h1!=l1,7), (7,l1 := l1+1;,6), (6,!h1!=l1,8), (8,write l2;,9), (9,write l3;,10), (10,write h2;,11), (11,write h3;,12)

Security level violations:
(9,write l2;,10), (10,write l3;,11), (11,write h2;,12)
```

Code 6.4 Analysis result of benchmark 1

7. Conclusion

At this project dedicated to static program analysis 4 different analysis of monotone framework are designed and implemented, namely: reaching definitions, detection of signs, interval analysis and security analysis.

The analysis are designed and implemented for the While language, therefore, firstly, a parser for this language is implemented. Reaching definitions, detection of signs and interval analysis are accompanied with definition, proofs, discussion of correctness.

The project shows that in order to achieve precise analysis implementation a lot of programmer/hours are required. Therefore, limitations and difficulties are described in the report as well. The wise class hierarchy and design of a wise software structure in advance can help to overcome stated limitation and achieve more precision with less code and higher computation efficiency.

8. Contributions

The project was organized as a three-person project. All the solutions, models presented in the report are discussion results by the group. All students contribute to each part of the report. However each member in the group might have bigger contributions to certain aspects.

Anusha Sivakumar was concentrate on the design of the worklist algorithms, the computation of free variables, the improvement and implementation of the program slicing. She has researched and wrote the corresponding sections in the report.

Nikita Martynov was responsible for the construction of the program graph, the improvement and implementation of buffer overflow – detection of signs, the improvement and implementation of security analysis. He has researched and wrote the corresponding sections in the report.

Zhen Li was responsible for modifying the parser, the design of the flow graph, the proof of instance of Monotone Framework, the improvement and implementation of buffer overflow - interval analysis. She has researched and wrote the corresponding sections in the report.

9. Instructions to run the application

The analysis application implemented takes a While program file and a command as inputs. The commands are numbered with the following order: 1 – program slice, 2- detection of signs, 3 – interval analysis, and 4 – security analysis. The interval analysis takes two additional inputs – the lower and upper boundaries. The lower and upper boundaries provided by the user should be valid integers. The program then prints the required analysis result for the user. Some example input could be:

```
file_xx 1
file_xx 2
file_xx 3 0 4
```

Appendix

Appendix A The syntax of the while language

$a ::= n \mid x \mid A[a] \mid a_1 op_a a_2 \mid -a \mid (a)$

$b ::= true \mid false \mid a_1 op_r a_2 \mid b_1 op_b b_2 \mid !\, b \mid (b)$

$S ::= x ::= a; \mid skip; \mid A[a_1] := a_2; \mid read\ x; \mid read\ A[a]; \mid write\ a;$

$\mid S_1 S_2 \mid if\ b\ then\ S_1\ else\ S_2\ fi \mid while\ b\ do\ S\ od$

$L ::= \epsilon \mid high \mid low$

$D ::= L\ int\ x; \mid L\ int\ A[n]; \mid \epsilon \mid D_1 D_2$

$P ::= program\ D\ S\ end$

Operators:

$op_a \in \{+, -, *, /\}$ int $*$ int $\rightarrow$ int

$op_r \in \{<, >, <=, >=, =, !=\}$ int $*$ int $\rightarrow$ bool

$op_b \in \{\&, |\}$ bool $*$ bool $\rightarrow$ bool

Notation:

x $\in$ Var: variable names n $\in$ Z: integer constant

A $\in$ Arr: array names

Appendix B 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 B.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 $RD_{\circ}(9)$;
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 $RD_{\circ}(6)$;
udchain(n,6) = 1 from $RD_{\circ}(6)$;
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) = 4,10 from $RD_{\circ}(10)$;
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) = 2,8 from $RD_{\circ}(7)$;
udchain(f2,7) = 3,9 from $RD_{\circ}(7)$;
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) = 3,8;
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) = 8;
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) = 3,9 from $RD_{\circ}(8)$;
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.