Danmarks Tekniske Universitet

02242 Program Analysis

# Exam Project

## Program analysis tool

Ibrahim Nemli
s093477

Kim Rostgaard Christensen
s084283

Peter Gammelgaard Poulsen
s093263

December 2nd, 2013

# Contents

# Chapter 1 Introduction

Program analysis is the discipline of extracting and deriving information about the structure, and potential behavior of computer programs. It has many applications, such as; compiler construction, bug finding, optimizations, code reuse, formal validation and model mapping. In this report, we take a closer look at some of the specialized and general theory used for transforming source code text into analysis-friendly structures, and then performing the actual analysis on them. This is done by parsing a program into an abstract syntax tree, before mapping it into a flow graph that can be used together with a worklist algorithm to solve the equations from an analysis.

The next section will cover the choices of data structures for the abstract syntax tree and the flow graph. Then the analysis reaching definitions will be defined and it will be presented how to use it to calculate a program slice. Next detection of signs analysis will be used to find buffer overflows and followed by how interval analysis can be used to detect buffer overflows as well. The next section will present how these analysis' have been implemented in a java application. Finally the conclusion will sum up the important findings.

# Chapter 2 Data structures

This chapter will cover the language considered followed by the selected data structures which will be used for the abstract syntax tree, which will serve as the first step of our intermediate representation. Furthermore it will cover the data structures used to store the flow graph.

## 2.1 The WHILE language

For the project the `WHILE` language will be used. It is defined below.

a ::= n | x | A[a] | $a_1$ $op_a$ $a_2$ | -a | (a)
b ::= true | false | $a_1$ $op_r$ $a_2$ | $b_1$ $op_b$ $b_2$ | !b | (b)
S ::= x :=a; | skip; | A[$a_1$]:= $a_2$; | read x; | read A[a]; | write a; | S1 S2 | if b then S1 else S2 fi | while b do S od
L ::= $\epsilon$ | high | low
D ::= L int x; | L int A[n]; | $\epsilon$ | $D_1$ $D_2$
P ::= program D S end

## 2.2 Abstract syntax tree data structure

Before any analysis can be performed on a program written in the language in the previous section it needs to be parsed into an abstract syntax tree.

We have decided that we could use an unbalanced tree for storage of our abstract syntax tree in our target programming language. Constraining the tree construction will give us the necessary check for syntax errors. A simple tree-traversal will give us the appropriate unique label numbers - or the ability to assign labels to statements in the right order.

In our implementation language the statements from the `WHILE` language will be mapped into the following abstract classes.

- **Statement** - Figure 2.1 shows the structures of statements.
- **Variable** - Figure 2.2 shows the structures of variables.
- **Condition** - Figure 2.3 shows the structures of conditions.
- **Expression** - Figure 2.4 shows the structures of expressions.

Each token from the grammar is then mapped to a realization of a class corresponding to its defined type. E.g. an Assignment class will derive from a Statement class.

Declarations will be handled separately from statements, and will be held in a dedicated declaration list until needed. Only one constraint exists for declarations: Their identifiers must be unique.

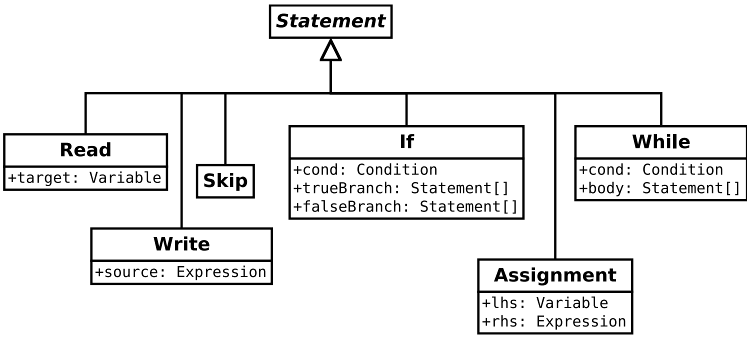

Figure 2.1: Statements and their derived types

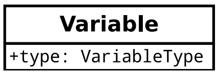

Figure 2.2: Variables and their derived types

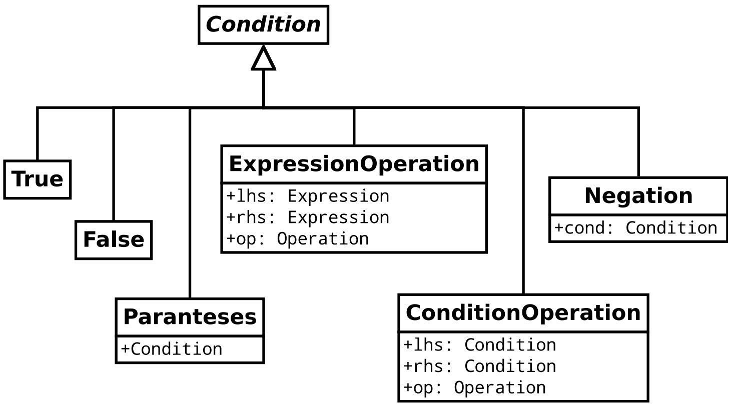

Figure 2.3: Conditions and their derived types

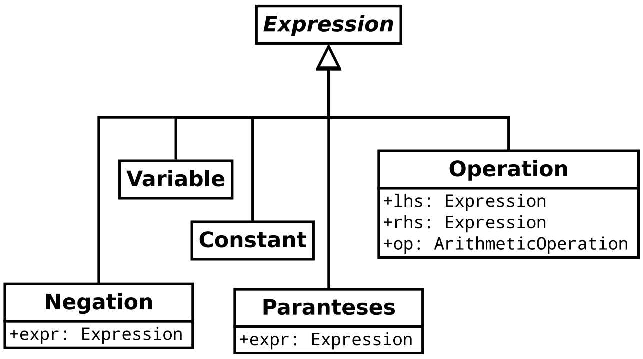

Figure 2.4: Expressions and their derived types

| | |
|---|---|
| *label(S)* | The set of nodes of the flow graph $S$ |
| *init(S)* | The initial node of the flow graph $S$.<br>Unique node where the execution of the program starts. |
| *final(S)* | The final node of the flow graph $S$.<br>A set of nodes where the execution of the program may terminate. |
| *flow(S)* | The edges of the flow graph for $S$. A set of pairs is returned. |
| *block(S)* | A set of the blocks/statements in the program under inquisition. |

Table 2.1: Function definitions

## 2.3 Flow graph data structure

For the flow graph, we have chosen a network graph data structure. This gives an intuitive way of linking statements to potential paths.

It is convenient to transform the abstract syntax tree into a flow graph when performing an analysis. For that reason a data structure for the flow graph as well as mapping from a abstract syntax tree to a flow graph is needed

Figure 2.5 shows the data structure used to represent a flow graph. The idea is that a flow graph consists of a set of flow. Each flow is between two nodes. A source and a target node. A node can be a part of several flows. Each node contains a statement.

### 2.3.1 Defining flow graphs

Before any analysis can take place we need to create a flow graph based on our abstract syntax tree. A flow graph should consists of the elements presented in table 2.1.

The objective is to establish a flow graph using the abstract syntax tree. Using the intermediate representation of the abstract syntax tree, the types of different individual statements is defined. Rather than explicitly parsing the AST to a flow graph, we instead define a function for each statement and let every statement handle it's own branch by recursion using the

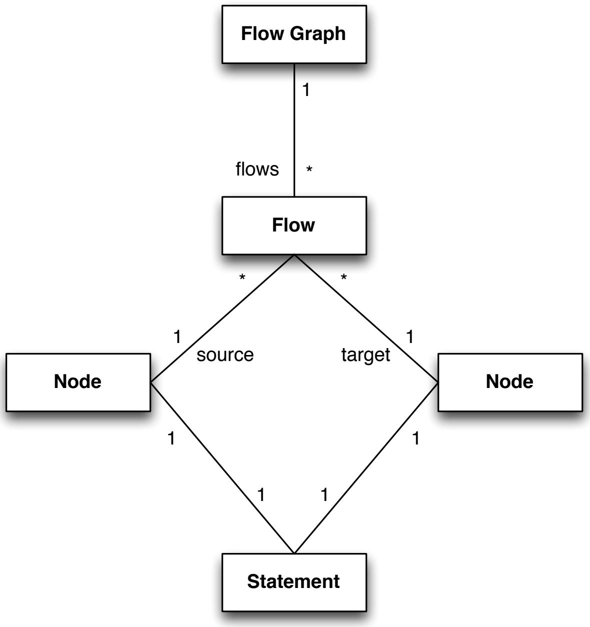

Figure 2.5: Data structures used to represent a flow graph

function defined in table 2.2.

## 2.4 Program graph data structure

Another approach for structuring a program is to use a program graph which is similar to flow graphs but instead of storing the statements in the nodes they are stored in the edges. This has some advantages in certain analysis but in general both flow graph and programs graph can be used since it is possible to convert between the two.

In the rest of the report the focus will be on flow graphs.

| $S$ | $labels(S)$ | $init(S)$ | $final(S)$ | $flow(S)$ | $blocks(S)$ |
|---|---|---|---|---|---|
| $[\mathrm{x} := \mathrm{a}]^{\ell}$ | $\{\ell\}$ | $\ell$ | $\{\ell\}$ | $\varnothing$ | $\{[\mathrm{x} := \mathrm{a}]^{\ell}\}$ |
| $[\mathrm{A}[\mathrm{a}_1] := \mathrm{a}_2]^{\ell}$ | $\{\ell\}$ | $\ell$ | $\{\ell\}$ | $\varnothing$ | $\{[\mathrm{A}[\mathrm{a}_1] := \mathrm{a}_2]^{\ell}\}$ |
| $[\mathrm{skip}]^{\ell}$ | $\{\ell\}$ | $\ell$ | $\{\ell\}$ | $\varnothing$ | $\{[\mathrm{skip}]^{\ell}\}$ |
| $S_1;S_2$ | $labels(S_1) \cup labels(S_2)$ | $init(S_1)$ | $\mathrm{final}(\mathrm{S}_2)$ | $flow(S_1) \cup flow(S_2)$<br>$\cup\{(\ell,init(S_2)) \mid \ell \in final(S_1)\}$ | $block(S_1)$<br>$\cup\ block(S_2)$ |
| if $[b]^{\ell}$ then<br>$S_1$ else $S_2$ | $\{\ell\} \cup labels(S_1)$<br>$\cup\ labels(S_2)$ | $\ell$ | $final(S_1)$<br>$\cup\ final(S_2)$ | $flow(S_1) \cup flow(S_2)$<br>$\cup\{(\ell,init(S_1)),(\ell,init(S_2))\}$ | $\{[b]^{\ell}\} \cup block(S_1)$<br>$\cup\ block(S_2)$ |
| while $[b]^{\ell}$ do $S$ | $\{\ell\} \cup labels(S)$ | $\ell$ | $\{\ell\}$ | $\{(\ell,init(S))\} \cup flow(S)$<br>$\cup\ \{(\ell',\ell) \mid \ell' \in final(S)\}$ | $\{[b]^{\ell}\} \cup block(S)$ |
| $[\mathrm{read}\ x]^{\ell}$ | $\{\ell\}$ | $\ell$ | $\{\ell\}$ | $\varnothing$ | $\{[\mathrm{read}\ x]^{\ell}\}$ |
| $[\mathrm{read\ A}[\mathrm{a}_1]]^{\ell}$ | $\{\ell\}$ | $\ell$ | $\{\ell\}$ | $\varnothing$ | $\{[\mathrm{read\ A}[\mathrm{a}_1]]^{\ell}\}$ |
| $[\mathrm{write}\ x]^{\ell}$ | $\{\ell\}$ | $\ell$ | $\{\ell\}$ | $\varnothing$ | $\{[\mathrm{write}\ x]^{\ell}\}$ |
| $[\mathrm{write\ A}[\mathrm{a}_1]]^{\ell}$ | $\{\ell\}$ | $\ell$ | $\{\ell\}$ | $\varnothing$ | $\{[\mathrm{write\ A}[\mathrm{a}_1]]^{\ell}\}$ |

Table 2.2: Flow graph function definitions

CHAPTER 3

# Program Slicing

A program slicing is a set of statements that may affect the values at some point of interest. Is often used in debugging to find errors.

## 3.1 Example program

Figure 3.1 shows an example program written in the `WHILE` language. This example program will be used as an example for this section. The program consists of 3 declared variables which are assigned several times and a while loop.

For this program the program slice can be calculated for a point of interest. Below we have discarded the declarations of the variables in the result.

- If the point of interest is label 8. The result of program slice analysis would be; [y:=x]$^4$, [y:=y-1]$^8$.
- If instead the point of interest is label 7. The result of program slice analysis would be; [y:=x]$^4$, [z:=1]$^5$, [z:=z*y]$^7$, [y := y-1]$^8$.

## 3.2 Reaching Definitions Analysis

Our algorithm for calculating the Program Slice will use the result of a Reaching Definition Analysis. So first we need to define what a reaching definition analysis is and how to compute it. A reaching definition analysis determines statically which definitions may reach a point of interests.

```
program
[int x]^1
[int y]^2
[int z]^3
[y := x]^4
[z := 1]^5
while [y>0]^6 do
   [z:= z*y]^7
   [y:= y-1]^8
od
[y:=0]^9
end
```

Listing 3.1:

Figure 3.1: Code used in to calculate program slice example.

Table 3.1 shows the Reaching Definitions Analysis table for the extended `WHILE` language. The `kill` follows the function:

$$Blocks_* \rightarrow P(Var_* \times Lab_*) \tag{3.1}$$

Meaning that a set of pairs of variables and labels are destroyed by the function.

The `gen` function is as follows:

$$Blocks_* \rightarrow P(Var_* \times Lab_*) \tag{3.2}$$

Where a set of pairs of variables and labels are added by the function.

Table 3.1 shows the `kill` and `gen` functions for the different blocks.

| | |
|---|---|
| **A[a$_1$] = a$_2$** | kill$_{RD}$([A[a$_1$]]$^{\ell}$) = ∅<br>gen$_{RD}$([A[n]$^{l}$) = $\{(A, l)\}$ |
| **read x** | kill$_{RD}$([read x]$^{\ell}$) = $\{(x, l'\} \mid B^{l'}$ is an assignment to x } <br>gen$_{RD}$([read x]$^{\ell}$) = $\{(x, l)\}$ |
| **read A[a]** | kill$_{RD}$([read A[a]]$^{\ell}$) = ∅<br>gen$_{RD}$([read A[a]]$^{\ell}$) = $\{(A, l)\}$ |
| **write a** | kill$_{RD}$([write a]$^{l}$) = ∅<br>gen$_{RD}$([write a]$^{l}$) = ∅ |
| **write A[a]** | kill$_{RD}$([write A[a]]$^{l}$) = ∅<br>gen$_{RD}$([write a]$^{l}$) = ∅ |
| **x:=a** | kill$_{RD}$([x:=a]$^{\ell}$) = { (x,l' }$\mid B^{l'}$ is an assignment to x }<br>gen$_{RD}$([x:=a]$^{\ell}$) = $\{(x, l)\}$ |
| **skip** | kill$_{RD}$([skip]$^{l}$) = ∅<br>gen$_{RD}$([skip]$^{l}$) = ∅ |
| **b** | kill$_{RD}$([b]$^{l}$) = ∅<br>gen$_{RD}$([b]$^{l}$) = ∅ |

Table 3.1: RD Equations

It is worth noticing the over-approximation with regards the arrays, in the sense that we do not care about the individual positions in the array, but the values of *any* position of the array. In this context, an array inherits some of the properties of an assignment, but as we cannot guarantee anything about which index was changed, we can not kill anything when an index in the array is assigned.

Furthermore table 3.2 shows the Reaching Definitions Analysis. By using the analysis and equations it is possible for a specific program to calculate the reaching definitions.

| | |
|---|---|
| $RD_\circ(l)$ | $\begin{cases} \{(x,?) \mid x \in Decl(S_*)\} & \text{if } \ell = init(S_*) \\ \cup\{RD_{exit}(\ell') \mid (\ell',\ell) \in flow(S_*)\} & \text{otherwise} \end{cases}$ |
| $RD_\bullet(l)$ | $(RD_{entry}(l) \backslash kill_{RD}(B^l)) \cup gen_{RB}(B^l)$<br>where $B^l \in blocks(S_*)$ |

Table 3.2: Analysis definitions

| | |
|---|---|
| Label($S$) | $\{1,2,3,4,5,6,7,8,9\}$ |
| Initial($S$) | 1 |
| Final($S$) | $\{9\}$ |
| Blocks($S$) | {int x, int y, int z y:=x, z:=1, y>0, z:=$z \cdot y$, y:=y-1, y:=0 } |
| Flow($S$) | $\{(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8)(8,6),(6,9)\}$ |

Table 3.3: Example flow graph table

## 3.3 Flow graph for program

In order to determine the reaching definitions for our example program written in the WHILE language the definitions for table 2.1 must be used. Table 3.3 shows the flow graph for our specific program. Notice how there is a flow from the last final block in the while (label 8) to the condition in the while (label 6).

A nice property of flow graphs is that they can easily be presented visually. Figure 3.2 shows the Flow Graph for our example program.

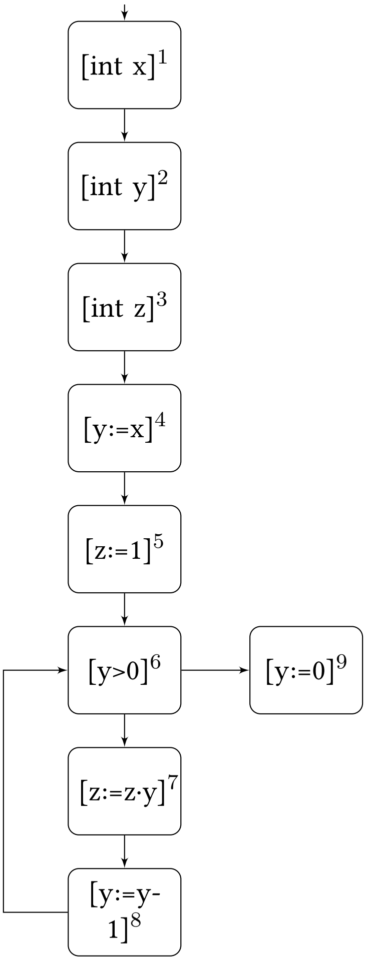

Figure 3.2: Flow graph for example program

## 3.4 Equations

Using the analysis definitions from table 3.2 we can establish a set of equations for the reaching definitions of our example program. They are shown in table 3.4.

$$\begin{aligned}
RD_\circ(1) &= \emptyset\\
RD_\circ(2) &= RD_\bullet(1)\\
RD_\circ(3) &= RD_\bullet(2)\\
RD_\circ(4) &= RD_\bullet(3)\\
RD_\circ(5) &= RD_\bullet(4)\\
RD_\circ(6) &= RD_\bullet(5) \cup RD_\bullet(8)\\
RD_\circ(7) &= RD_\bullet(6)\\
RD_\circ(8) &= RD_\bullet(7)\\
RD_\circ(9) &= RD_\bullet(6)
\end{aligned}$$

$$\begin{aligned}
RD_\bullet(1) &= (RD_\circ(1) \backslash \{(x,1)\}) \cup \{(x,1)\}\\
RD_\bullet(2) &= (RD_\circ(2) \backslash \{(y,2),(y,4),(y,8),(y,9)\}) \cup \{(y,2)\}\\
RD_\bullet(3) &= (RD_\circ(3) \backslash \{(z,3),(z,5),(z,7)\}) \cup \{(z,3)\}\\
RD_\bullet(4) &= (RD_\circ(4) \backslash \{(y,2),(y,4),(y,8),(y,9)\}) \cup \{(y,4)\}\\
RD_\bullet(5) &= (RD_\circ(5) \backslash \{(z,3),(z,5),(z,7)\}) \cup \{(z,5)\}\\
RD_\bullet(6) &= RD_\circ(6)\\
RD_\bullet(7) &= (RD_\circ(7) \backslash \{(z,3),(z,5),(z,7)\}) \cup \{(z,7)\}\\
RD_\bullet(8) &= (RD_\circ(8) \backslash \{(y,2),(y,4),(y,8),(y,9)\}) \cup \{(y,8)\}\\
RD_\bullet(9) &= (RD_\circ(9) \backslash \{(y,2),(y,4),(y,8),(y,9)\}) \cup \{(y,9)\}
\end{aligned}$$

Table 3.4: Equations

These equations can be solved using the equations from 3.1. The solutions can be found in table 3.5 and table 3.6. For instance is it possible to see that the reaching definitions going out of label 8 is for the variable $x$ label 1, the variable $z$ label 7 and finally the variable $y$ label 8. It is worth noticing that a the value of a single variable can depend on several labels in the program.

| l | $RD_\circ(l)$ |
|---|---|
| 1 | $\emptyset$ |
| 2 | $\{(x,1)\}$ |
| 3 | $\{(x,1),(y,2)\}$ |
| 4 | $\{(x,1),(y,2),(z,3)\}$ |
| 5 | $\{(x,1),(y,4),(z,3)\}$ |
| 6 | $\{(x,1),(y,4),(z,5),(z,7),(y,8)\}$ |
| 7 | $\{(x,1),(y,4),(z,5),(z,7),(y,8)\}$ |
| 8 | $\{(x,1),(y,4),(z,7),(y,8)\}$ |
| 9 | $\{(x,1),(y,4),(z,5),(z,7),(y,8)\}$ |

Table 3.5: Solution to equation

| l | $RD_\bullet(l)$ |
|---|---|
| 1 | $\{(x,1)\}$ |
| 2 | $\{(x,1),(y,2)\}$ |
| 3 | $\{(x,1),(y,2),(z,3)\}$ |
| 4 | $\{(x,1),(y,2),(z,3)\}$ |
| 5 | $\{(x,1),(y,4),(z,5)\}$ |
| 6 | $\{(x,1),(y,4),(z,5),(z,7),(y,8)\}$ |
| 7 | $\{(x,1),(y,4),(z,7),(y,8)\}$ |
| 8 | $\{(x,1),(z,7),(y,8)\}$ |
| 9 | $\{(x,1),(z,5),(z,7),(y,9)\}$ |

Table 3.6: Solution to equation

## 3.5 Program slice calculation algorithm

Now with the result of a reaching definitions analysis available it is possible to use that result to calculate a program slice for our program at a point of interest. An algorithm for doing so is given in algorithm 1.

The algorithm takes the label of the point of interest as input. The output is an array with the labels that is a part of the program slice. It is assumed that the Reaching Definitions Analysis has been performed beforehand and the result is available in the variable `RD`. The algorithm uses a queue where the point of interest is added to and continues until the queue is empty. For every iteration the label in the queue is added to the `result` array. Then all the variables that this label depends on is retrieved. The labels where these variables are used from the result of the Reaching Definitions Analysis is added to the queue. It is assumed that once a label has been added to the queue it can not be added again. This will then guarantee that the algorithm terminates.

The algorithm can calculate the Program Slice for the example program from section 3.1. We have the solution to the reaching definition analysis from table 3.5 available. To find the Program Slice with point of interest at label 8.

- First the point of interest is added to the queue. $queue = [8]$, $result = []$
- The condition is checked. The first element from the queue is removed and added to the result. $queue = []$, $result = [8]$
- The variables from label 8 is retrieved. This is only $y$.
- The result of the reaching definition analysis for label 8 is retrieved. This is $\{(x,1),(y,4),(z,7),(y,8)\}$.
- Each of these labels are iterated. If any of them contains the variable $y$ they will be added to the queue. This is 4 and 8. Since is it assumed that a label can not be added to a queue more than once only 4 will be added. $queue = [4]$, $result = [8]$.

**Algorithm 1** Calculate Program Slice

```
procedure PROGRAM SLICE($point\_of\_interest$)
    Array $result$
    Queue $queue$
    $queue.enqueue(point\_of\_interest)$
    while $queue.not\_empty()$ do
        Label $l$ := $queue.dequeue()$
        $result.add(l)$
        Array $variables$ := $l.get\_variables()$
        Array RD = $RD_{entry}(l)$
        for label in RD do
            if $variables.contains(label.get\_variables())$ then
                queue.enqueue(label)
            end if
        end for
    end while
    return $result$
end procedure
```

- The condition is checked again. The first element from the queue is removed and added to the result. $queue = []$, $result = [8, 4]$
- The variables from label 4 is retrieved. This is $x$.
- The result of the reaching definition analysis for label 4 is retrieved. This is $\{(x, 1), (y, 2), (z, 3)\}$.
- Each of these labels are iterated. If any of them contains the variable $x$ they will be added to the queue. This is 1. $queue = [1]$, $result = [8, 4]$.
- The condition is checked again. The first element from the queue is removed and added to the result. $queue = []$, $result = [8, 4, 1]$
- The variables from label 1 is retrieved. There is none.
- The condition is checked again. It evaluates to false and the result is returned. $result = [8, 4, 1]$

The reached result is the same as the presented solution in section 3.1.

# Chapter 4 Buffer overflowwith sign detection

Buffer overflow is historically one of the most common programming errors. It is the cause of security breaches, data corruption and program crashes. Detecting them is therefore preferred.

A way of doing so is using sign detection analysis. For a point of interest it is possible for each variable to see the signs that the variable may have. Using this information we can detect potential overflows.

## 4.1 Monotone framework

To establish our analysis we will define a monotone framework as generic framework that can be used to solve several analysis using a generic worklist algorithm.

A monotone framework consists of the following:

- L - A complete lattice satisfying the Ascending Chain Condition.
- $\mathcal{F}$ - A set of monotone functions.
- F - A flow
- E - A set of extremal labels
- $\iota$ - An extremal value for the extremal labels.
- f - Mapping of labels and transfer functions in $\mathcal{F}$.

In our analysis the flow F will be defined as:

**Flow** $F$: flow ($S_*$) (forward analysis)

The extremal labels E will be defined as:

**Extremal label** $E$: {init($S_*$)}

Table 4.1 defines the analysis. The monotone framework is shown in table 4.2. The lattice used in show in figure 4.1. This is a complete lattice, as every subset in the partial ordered set have a least upper bound and a greatest lower bound.
It also holds the Ascending Chain Property, because the lattice has a finite height, and will thus, eventually stabilize.

The top of the lattice gives the least precise solution and the bottom the most precise.

We define the mapping of labels to transfer functions for the analysis in table 4.3.

| | |
|---|---|
| Analysis$_\circ(\ell)$ | $\bigsqcup\{\text{ Analysis}_\bullet(\ell') \mid (\ell',\ell) \in F\} \cup \iota_E^\ell$<br>where $\iota_E^\ell = \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases}$ |
| Analysis$_\bullet(\ell)$ | f$_\ell$(Analysis$_\circ(\ell)$) |

Table 4.1: Analysis definitions

| Instance | $(L, \sqsubseteq, \bigsqcup, \bigsqcap, \top, \bot)$ |
|---|---|
| Lattice | $\{-, 0, +\}$ |
| $\sqsubseteq$-Ordering | $\subseteq$ (subset ordering). |
| Partial ordered set | $\mathcal{P}(\{-, 0, +\}, \subseteq)$ |
| Least upper bound | $\bigsqcup Y = \bigcup Y$ |
| Greatest lower bound | $\bigsqcap Y = \bigcap Y$ |
| $\bot$ | Bottom, least element : $\varnothing$ |
| $\top$ | Top, greatest element : $\{-, 0, +\}$ |

Table 4.2: Sign detection instance

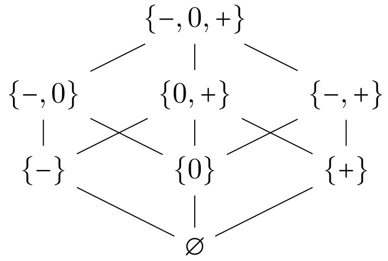

Figure 4.1: Complete lattice of sign detection instance

| Statement | Function |
|---|---|
| $[\text{x := a}]^\ell$ | $f_\ell^S(\widehat{\sigma}) = \widehat{\sigma}[x \mapsto \mathcal{A}_S[[a]]\widehat{\sigma}]$ |
| $[\text{skip}]^\ell$ | $f_\ell^S(\widehat{\sigma}) = \widehat{\sigma}$ |
| $[\text{b}]^\ell$ | $f_\ell^S(\widehat{\sigma}) = \widehat{\sigma}$ |
| $[\text{A[a}_1\text{] := a}_2]^\ell$ | $f_\ell^S(\widehat{\sigma}) = \widehat{\sigma}[A \mapsto \widehat{\sigma}(A) \cup \mathcal{A}_S[[a_2]]\widehat{\sigma}]$ |
| $[\text{read x}]^\ell$ | $f_\ell^S(\widehat{\sigma}) = [x \mapsto \top]$ |
| $[\text{read A[a]}]^\ell$ | $f_\ell^S(\widehat{\sigma}) = \widehat{\sigma}[A \mapsto \widehat{\sigma}(A) \cup \top]$ |
| $[\text{write x}]^\ell$ | $f_\ell^S(\widehat{\sigma}) = \widehat{\sigma}$ |
| $[\text{write A[n]}]^\ell$ | $f_\ell^S(\widehat{\sigma}) = \widehat{\sigma}$ |

Table 4.3: Transfer functions for sign detection

## 4.2 Determine signs

For the analysis to work the signs need to be determined for a given program. This is done by evaluating arithmetic expressions and boolean expressions.

### 4.2.1 Evaluate arithmetic expressions

In order to determine the sign of an expression we need to define the semantics of expressions in the `WHILE` language for the signs analysis. $\mathcal{A}_S[[a]] : \widehat{\mathbf{State}_S} \to \mathbf{Sign}$
Where **Sign** = $P(\{-, 0, +\})$
We have the following monotone functions:
$\mathcal{A}_S[[x]]\widehat{\sigma} = \widehat{\sigma}(x)$

$$\mathcal{A}_S[[n]]\widehat{\sigma} = \begin{cases} \{-\} & \text{if } n < 0 \\ \{0\} & \text{if } n = 0 \\ \{+\} & \text{if } n > 0 \end{cases}$$

$\mathcal{A}_S[[a_1 \text{ op}_A a_2]]\widehat{\sigma} = \mathcal{A}_S[[a_1]]\widehat{\sigma}\ \widehat{\text{op}_A}\mathcal{A}_S[[a_2]]\widehat{\sigma}$, where $\widehat{\text{op}_A}$ : **Sign** × **Sign** → **Sign** is given by :
$\widehat{\text{op}_A}(S_1, S_2) = \{s_1 \text{ op}_A s_2 | s_1 \in S_1, s_2 \in S_2\}$
$\widehat{\text{op}_A}$is a specific arithmetic operator from the set $\text{op}_a\{+, -, *, /\}$.
Table 4.4 defines for each of the operators what possible signs may be the outcome when evaluating an expression.

### 4.2.2 Evaluate boolean expressions

Similarly booleans expressions should be evaluated. We that that:
$BExp \to \widehat{\mathbf{State}_S} \to$ **P({tt,ff})** which will be given by:
$\mathcal{B}_S[[a_1 op_r a_2]]\widehat{\sigma}' = \mathcal{A}_S[[a_1]]\widehat{\sigma}'\widehat{op_r}\mathcal{A}_S[[a_2]]\widehat{\sigma}'$
$\mathcal{B}_S[[a_1 op_b a_2]]\widehat{\sigma}' = \mathcal{B}_S[[b_1]]\widehat{\sigma}'\widehat{op_b}\mathcal{B}_S[[b_2]]\widehat{\sigma}'$
$\mathcal{B}_S[[\neg b]]\widehat{\sigma}' = \{\neg t | t \in \mathcal{B}_S[[b]]\widehat{\sigma}'\}$

Where $\widehat{\text{op}_r}$ is a specific operator from the set $\text{op}_r$ $\{ \leq, \geq, \neq, =, >, < \}$, and where $\widehat{\text{op}_b}$ is a specific operator from the set $\text{op}_b$ $\{\&, |\}$. The possible outcome of these operators are define in

| + | - | 0 | + |
|---|---|---|---|
| - | $\{-\}$ | $\{-\}$ | $\{-,0,+\}$ |
| 0 | $\{-\}$ | $\{0\}$ | $\{+\}$ |
| + | $\{-,0,+\}$ | $\{+\}$ | $\{+\}$ |

| x | - | 0 | + |
|---|---|---|---|
| - | $\{+\}$ | $\{0\}$ | $\{-\}$ |
| 0 | $\{0\}$ | $\{0\}$ | $\{0\}$ |
| + | $\{-\}$ | $\{0\}$ | $\{+\}$ |

| - | - | 0 | + |
|---|---|---|---|
| - | $\{-,0,+\}$ | $\{-\}$ | $\{-\}$ |
| 0 | $\{-\}$ | $\{0\}$ | $\{-\}$ |
| + | $\{+\}$ | $\{+\}$ | $\{-,0,+\}$ |

| \ | - | 0 | + |
|---|---|---|---|
| - | $\{+\}$ | $\varnothing$ | $\{-\}$ |
| 0 | $\{0\}$ | $\varnothing$ | $\{0\}$ |
| + | $\{-\}$ | $\varnothing$ | $\{+\}$ |

Table 4.4: Operator effect on variable state.

| & | tt | ff |
|---|---|---|
| tt | $\{tt\}$ | $\{ff\}$ |
| ff | $\{ff\}$ | $\{ff\}$ |

| \| | tt | ff |
|---|---|---|
| tt | $\{tt\}$ | $\{ff\}$ |
| ff | $\{tt\}$ | $\{ff\}$ |

Table 4.5: Possible outcomes of AND and OR operator.

table 4.5 and table 4.6.

## 4.3 Buffer underflow calculation algorithm

Based on the detection of signs analysis described above, we can suggest an algorithm that using the result of a detection of signs analysis can warn about possible underflows. This is done by checking for an assignment which includes an operation with an array and determine looking at the possible signs for the expression used as the index. Algorithm 2 shows how this could be implemented.

**Algorithm 2** Calculate buffer underflow based on detection of signs analysis.

```
procedure BUFFER UNDERFLOW($l$)
    A:=$l.get\_array()$
    a:=$A.get\_index\_expression()$
    $signs := A_S[a]$
    if $signs == \{-\}$ then
        return Bufer_Underflow
    else if $signs.contains(-)$ then
        return Potential_Bufer_Underflow
    else
        return No_Buffer_Underflow
    end if
end procedure
```

| < | - | 0 | + |
|---|---|---|---|
| - | $\{tt, ff\}$ | $\{tt\}$ | $\{tt\}$ |
| 0 | $\{ff\}$ | $\{ff\}$ | $\{tt\}$ |
| + | $\{ff\}$ | $\{ff\}$ | $\{tt, ff\}$ |

| > | - | 0 | + |
|---|---|---|---|
| - | $\{tt, ff\}$ | $\{ff\}$ | $\{ff\}$ |
| 0 | $\{tt\}$ | $\{ff\}$ | $\{ff\}$ |
| + | $\{tt\}$ | $\{tt\}$ | $\{tt, ff\}$ |

| ≤ | - | 0 | + |
|---|---|---|---|
| - | $\{tt, ff\}$ | $\{tt\}$ | $\{tt\}$ |
| 0 | $\{ff\}$ | $\{tt\}$ | $\{tt\}$ |
| + | $\{ff\}$ | $\{ff\}$ | $\{tt, ff\}$ |

| ≥ | - | 0 | + |
|---|---|---|---|
| - | $\{tt, ff\}$ | $\{ff\}$ | $\{ff\}$ |
| 0 | $\{tt\}$ | $\{tt\}$ | $\{ff\}$ |
| + | $\{tt\}$ | $\{tt\}$ | $\{tt, ff\}$ |

| = | - | 0 | + |
|---|---|---|---|
| - | $\{tt, ff\}$ | $\{ff\}$ | $\{ff\}$ |
| 0 | $\{ff\}$ | $\{tt\}$ | $\{ff\}$ |
| + | $\{ff\}$ | $\{ff\}$ | $\{tt, ff\}$ |

| ≠ | - | 0 | + |
|---|---|---|---|
| - | $\{tt, ff\}$ | $\{tt\}$ | $\{tt\}$ |
| 0 | $\{tt\}$ | $\{ff\}$ | $\{tt\}$ |
| + | $\{tt\}$ | $\{tt\}$ | $\{tt, ff\}$ |

Table 4.6: Possible outcomes of compare operators.

```
program
[int A[2]]^1
[int i]^2
[i := -1]^3
while [i<3]^4 do
   [A[i] := i+1]^5
   [i := i+1]^6
od
end
```

Listing 4.1:

Figure 4.2: Example program for demonstrating buffer overflow using sign detection

### 4.3.1 Example solution

The analysis described above can be used in practice to determine the potential buffer underflow in a program. In figure A.1 is defined a program written in the WHILE language. The analysis should be performed on this program in order to establish equations. They are shown in table 4.7. The solution to these equations if shown in table 4.8. It can be seen that there may be a buffer underflow at label 5 since the value of $i$ may be negative.

| | |
|---|---|
| $\mathrm{A}_\circ(1) = \iota\{\mathrm{init}(S_*)\}$ | $\mathrm{A}_\bullet(1) = \mathrm{f}_{\mathrm{int\ A[2]}}(\mathrm{A}_\circ(1))$ |
| $\mathrm{A}_\circ(2) = \mathrm{A}_\bullet(1)$ | $\mathrm{A}_\bullet(2) = \mathrm{f}_{\mathrm{int\ i}}(\mathrm{A}_\circ(2))$ |
| $\mathrm{A}_\circ(3) = \mathrm{A}_\bullet(2)$ | $\mathrm{A}_\bullet(3) = \mathrm{f}_{\mathrm{i\ :=\ -1}}(\mathrm{A}_\circ(3))$ |
| $\mathrm{A}_\circ(4) = \mathrm{A}_\bullet(3) \bigcup \mathrm{A}_\bullet(6)$ | $\mathrm{A}_\bullet(4) = \mathrm{f}_{\mathrm{i\ <\ 3}}(\mathrm{A}_\circ(4))$ |
| $\mathrm{A}_\circ(5) = \mathrm{A}_\bullet(4)$ | $\mathrm{A}_\bullet(5) = \mathrm{f}_{\mathrm{A[i]\ :=\ i\ +\ 1}}(\mathrm{A}_\circ(5))$ |
| $\mathrm{A}_\circ(6) = \mathrm{A}_\bullet(5)$ | $\mathrm{A}_\bullet(6) = \mathrm{f}_{\mathrm{i\ :=\ i\ +\ 1}}(\mathrm{A}_\circ(6))$ |

Table 4.7: Equations for example program

| $\mathrm{A}_\bullet(\cdot)$ | A | i |
|---:|:---:|:---:|
| 3 | $\{0\}$ | $\{-\}$ |
| 4 | $\{-,0,+\}$ | $\{-,0,+\}$ |
| 5 | $\{-,0,+\}$ | $\{-,0,+\}$ |
| 6 | $\{-,0,+\}$ | $\{-,0,+\}$ |

Table 4.8: Solution to equations

Chapter 5

# Buffer Overflow Interval Analysis

Interval analysis establishes intervals. In the sign detection analysis from chapter 3.5 a solution to the buffer overflow problem by only considering the lower bound was provided. Using interval analysis it is possible to consider the upper bound as well by keeping intervals for each variable as determine if the max of the interval is potentially larger than the bounds.

## 5.1 Monotone framework

The analysis is based on a monotone framework, which has the form:

$$(L, \mathcal{F}, F, E, \iota, \mathrm{f.}) \tag{5.1}$$

Table 5.1 define our lattice for this analysis. In figure 5.1 the lattice for the analysis is shown. This is a complete lattice, as every subset in the partial ordered set have a least upper bound and a greatest lower bound.
It also holds the Ascending Chain Property, because the lattice has a finite height, and will thus, eventually stabilize.

| Instance | $(L, \sqsubseteq, \bigsqcup, \bigsqcap, \top, \bot)$ |
|---|---|
| Lattice | (Interval, $\sqsubseteq$) |
| Interval | $\{\bot\} \bigcup \{[z_1, z_2] \mid z_1 < z_2, z_1, z_2, \in Z' \bigcup \{-\infty, \infty\}\}$<br>where $Z' = \{z \mid \min \leq z \leq \max\}$ and $-\infty \leq \infty$, and $\min, \max \in Z$ |
| $\sqsubseteq$-Ordering | $\subseteq$ (subset ordering).<br>Interval$_1$ $\sqsubseteq$ Interval$_2$ if, and only if $\beta(\text{Interval}_1) \subseteq \beta(\text{Interval}_2)$, where<br>$\beta(\bot) = \varnothing$<br>$\beta([z_1, z_2]) = \{z \in Z \mid z_1 \leq z \leq z_2\}$ if $z_1 \in Z' \bigcup \{-\infty\}$ and $z_2 \in Z' \bigcup \{\infty\}$<br>$\beta(\{-\infty, -\infty\}) = \{z \mid z < \min\}$<br>$\beta(\{\infty, \infty\}) = \{z \mid z > \max\}$ |
| Least upper bound | $\bigsqcup Y = \bigcup Y$ |
| Greatest lower bound | $\bigsqcap Y = \bigcap Y$ |
| $\bot$ | Bottom, least element : $\varnothing$ |
| $\top$ | Top, greatest element : $[-\infty, \infty]$ |

Table 5.1: Lattice definition

For this analysis we define the flow as:

**Flow** $F$: flow $(S_*)$ (forward analysis)

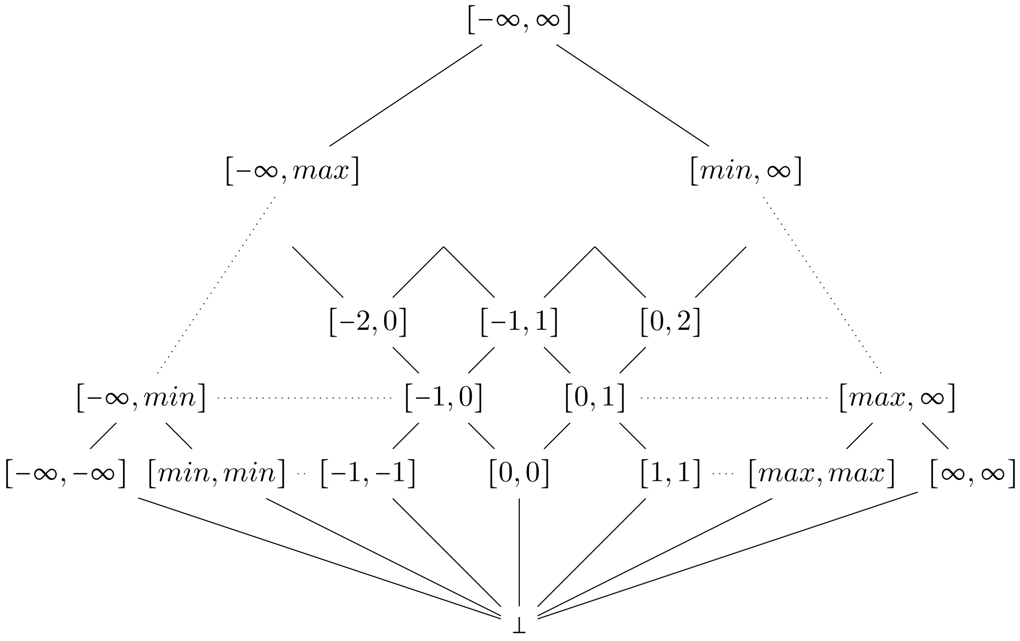

Figure 5.1: Complete lattice of interval analysis instance

**Extremal label** $E$: {init($S_*$)}

The analysis is defined as:

| | |
|---|---|
| Analysis$_\circ(\ell)$ | $\bigsqcup\{$ Analysis$_\bullet(\ell')\|(\ell',\ell) \in F\} \cup \iota_E^\ell$<br>where $\iota_E^\ell = \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases}$ |
| Analysis$_\bullet(\ell)$ | f$_\ell$(Analysis$_\circ(\ell)$) |

Table 5.2: Analysis definitions

Table 5.3 shows our transfer function for the extended `WHILE` language.

### 5.1.1 Evaluate arithmetic expressions

In order to determine the interval of an expression we need to define the semantics of expressions in the `WHILE` language for the interval analysis.

$$A_I : AExp \to (\widehat{\text{State}_\text{I}} \to \textbf{Interval}) \tag{5.2}$$

Where **Interval**= $\{\bot\} \cup \{[z_1,z_2]|z_1 \le z_2, z_1, z_2 \in Z' \cup \{-\infty,\infty\}\}$

Further we define the following functions:

| Statement | Function |
|---|---|
| $[\text{x := a}]^\ell$ | $\text{f}^I_\ell(\widehat{\sigma}) = \widehat{\sigma}[x \mapsto \mathcal{A}_I[[a]]\widehat{\sigma}]$ |
| $[\text{skip}]^\ell$ | $\text{f}^I_\ell(\widehat{\sigma}) = \widehat{\sigma}$ |
| $[\text{b}]^\ell$ | $\text{f}^I_\ell(\widehat{\sigma}) = \widehat{\sigma}$ |
| $[\text{A}[\text{a}_1] := \text{a}_2]^\ell$ | $\text{f}^I_\ell(\widehat{\sigma}) = \widehat{\sigma}[A \mapsto \widehat{\sigma}(A) \cup \mathcal{A}_I[[a_2]]\widehat{\sigma}]$ |
| $[\text{read x}]^\ell$ | $\text{f}^I_\ell(\widehat{\sigma}) = [x \mapsto \top]$ |
| $[\text{read A[a]}]^\ell$ | $\text{f}^I_\ell(\widehat{\sigma}) = \widehat{\sigma}[A \mapsto \widehat{\sigma}(A) \cup \top]$ |
| $[\text{write x}]^\ell$ | $\text{f}^I_\ell(\widehat{\sigma}) = \widehat{\sigma}$ |
| $[\text{write A[n]}]^\ell$ | $\text{f}^I_\ell(\widehat{\sigma}) = \widehat{\sigma}$ |

Table 5.3: Transfer functions for sign detection

$$A_I[[x]]\widehat{\sigma} = \widehat{\sigma}(x)$$

$$A_I[[n]]\widehat{\sigma} = \begin{cases} [n;n] & \text{if } \min \leq n \leq \max \\ [-\infty;-\infty] & \text{if } n < \min \\ [\infty;\infty] & \text{if } n > \max \end{cases}$$

$$A_I[[a_1 op_a a_2]] = A_I[[a_1]]\widehat{\text{op}_a}A_I[[a_1]]\sigma$$

where $\widehat{\text{op}_a}$ : interval × interval → interval is an abstract operator on intervals and $\widehat{\text{op}_A}$ is a specific arithmetic operator from the set $\text{op}_a\{+,-,*,/\}$. The definitions for these operators at listen below.

### 5.1.2 Addition on intervals

$$[Z_{11}, Z_{12}] + [Z_{21}, Z_{22}] = [Z_1, Z_2] = \begin{cases} Z_{1i} + Z_{2i} & \text{if } \min \leq Z_{1i} + Z_{2i} \leq \max \\ -\infty & \text{if } Z_{1i} + Z_{2i} < \min \\ \infty & \text{if } Z_{1i} + Z_{2i} > \max \end{cases}$$

### 5.1.3 Subtraction on intervals

$$[Z_{11}, Z_{12}] - [Z_{21}, Z_{22}] = [Z_1, Z_2] = \begin{cases} Z_{1i} - Z_{2i} & \text{if } \min \leq Z_{1i} - Z_{2i} \leq \max \\ -\infty & \text{if } Z_{1i} - Z_{2i} < \min, \text{ or } Z_12 = -\infty \text{or } Z_21 = -\infty \\ \infty & \text{if } Z_{1i} - Z_{2i} > \max, \text{ or } Z_12 = \infty \text{or } Z_21 = \infty \end{cases}$$

### 5.1.4 Multiplication on intervals

First, we define:
$a = \min(z_{11} \cdot z_{21}, z_{11} \cdot z_{22}, z_{12} \cdot z_{21}, z_{12} \cdot z_{22})$
$b = \max(z_{11} \cdot z_{21}, z_{11} \cdot z_{22}, z_{12} \cdot z_{21}, z_{12} \cdot z_{22})$

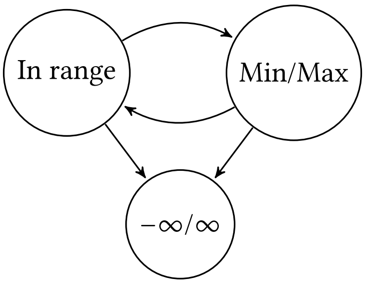

Figure 5.2: Interval states

$$z_1 = \begin{cases} a & \text{if } \min \leq a \leq \max \\ -\infty & \text{if } a < \min\text{, or } \exists(i,j) : Z_{ij} = -\infty \\ \infty & \text{if } a > \max\text{, or } \exists(i,j) : Z_{ij} = \infty \end{cases}$$

$$z_2 = \begin{cases} b & \text{if } \min \leq b \leq \max \\ -\infty & \text{if } b < \min\text{, or } \exists(i,j) : Z_{ij} = -\infty \\ \infty & \text{if } b > \max\text{, or } \exists(i,j) : Z_{ij} = \infty \end{cases}$$

### 5.1.5 Division on intervals

First, we define:
$[z_{11}, z_{12}]/[z_{21}, z_{22}] = [z_1, z_2]$ $[z_1, z_2] = \top$ when 0not $\in [z_{21}, z_{22}]$
$a = \min(z_{11}/z_{21}, z_{11}/z_{22}, z_{12}/z_{21}, z_{12}/z_{22})$
$b = \max(z_{11}/z_{21}, z_{11}/z_{22}, z_{12}/z_{21}, z_{12}/z_{22})$

$$Z_1 = \begin{cases} a & \text{if } \min \leq a \leq \max \\ -\infty & \text{if } a < \min\text{, or } \exists(i,j) : Z_{ij} = -\infty \\ \infty & \text{if } a > \max\text{, or } \exists(i,j) : Z_{ij} = \infty \end{cases}$$

$$Z_2 = \begin{cases} b & \text{if } \min \leq b \leq \max \\ -\infty & \text{if } b < \min\text{, or } \exists(i,j) : Z_{ij} = -\infty \\ \infty & \text{if } b > \max\text{, or } \exists(i,j) : Z_{ij} = \infty \end{cases}$$

## 5.2 Algorithm for calculating buffer overflow

The result of an interval analysis can be used to calculate whenever a buffer overflow is present. Algorithm 3 returns true if a buffer overflow is present for a specific label. It is assumed that the provided label contains an array operation.

**Algorithm 3** Calculate buffer overflow

```
procedure BUFFER OVERFLOW($l$)
    A:=$l.get\_array()$
    a:=$A.get\_index\_expression()$
    $index\_interval := A_I[a]$
    $array\_bounds := [0, A.get\_size() - 1]$
    if $index\_interval.is\_within(array\_bounds)$ then
        return false
    else
        return true
    end if
end procedure
```

## 5.3 Example solution

The above analysis can be used to check for buffer overflows in the program from figure A.1. The equations is shown in figure 5.4. The solution to these constraints is shown in table 5.5. It can be seen that there might be an underflow at label 5.

| | |
|---|---|
| $\mathrm{A}_\circ(1) = \iota\{\mathrm{init}(S_*)\}$ | $\mathrm{A}_\bullet(1) = \mathrm{f}_{\mathrm{int\ A[2]}}(\mathrm{A}_\circ(1))$ |
| $\mathrm{A}_\circ(2) = \mathrm{A}_\bullet(1)$ | $\mathrm{A}_\bullet(2) = \mathrm{f}_{\mathrm{int\ i}}(\mathrm{A}_\circ(2))$ |
| $\mathrm{A}_\circ(3) = \mathrm{A}_\bullet(2)$ | $\mathrm{A}_\bullet(3) = \mathrm{f}_{\mathrm{i\ :=\ -1}}(\mathrm{A}_\circ(3))$ |
| $\mathrm{A}_\circ(4) = \mathrm{A}_\bullet(3) \cup \mathrm{A}_\bullet(6)$ | $\mathrm{A}_\bullet(4) = \mathrm{f}_{\mathrm{i\ <\ 3}}(\mathrm{A}_\circ(4))$ |
| $\mathrm{A}_\circ(5) = \mathrm{A}_\bullet(4)$ | $\mathrm{A}_\bullet(5) = \mathrm{f}_{\mathrm{A[i]\ :=\ i\ +\ 1}}(\mathrm{A}_\circ(5))$ |
| $\mathrm{A}_\circ(6) = \mathrm{A}_\bullet(5)$ | $\mathrm{A}_\bullet(6) = \mathrm{f}_{\mathrm{i\ :=\ i\ +\ 1}}(\mathrm{A}_\circ(6))$ |

[h]

Table 5.4: Equations for the program from figure A.1

| $\mathrm{A}_\bullet(\cdot)$ | A | i |
|---|---|---|
| 1 | $[0, 2]$ | $\bot$ |
| 2 | $[0, 2]$ | $[0, 0]$ |
| 3 | $[0, 2]$ | $[-1, 0]$ |
| 4 | $[-1, 2]$ | $[-1, 1]$ |
| 5 | $[-1, 2]$ | $[-1, 1]$ |
| 6 | $[-1, 2]$ | $[-1, 1]$ |

Table 5.5: Solution to equations

# Chapter 6 General about implementation

This chapter will first present the implementation of the parser for the `While` language. Then the implementations of the analysis' described in the previous section will be discussed.

The Java language has been chosen as the implementation language.

For the implementation we have chosen to map the label $\ell$ to a Node object. The label is still present within the node and is used for representation but also contains additional information, such as the which statement it maps to.

## 6.1 Parsing the program

Before any analysis can take place it necessary that the program is parsed. This is done by having a parser. ANTLR is used as the framework for constructing a lexer and a parser that can take a program written in the extended `While` language and construct an abstract syntax tree (AST) using the data structures described in section 2.2.

- **The Lexer** - We've implemented the lexer as a grammar file for ANTLR, supplying us with a basic tokenizer that is able to attach extra information to the streamed tokens. The grammar file is located in the appended code for those interested.

- **The parser** - The parser is created by extending the grammar file hereby enabling it to expand the grammar to a generated parser. The parser harvests information from the lexer in order to provide an AST containing the information we need for out analysis' later on.

## 6.2 Generalizing the analysis

To abstract away the analysis from the general parser tool, we've created an interface that implemented by our abstract Statement class. The interface basically maps the *label(S)*, *init(S)*, *final(S)* and *flow(S)* functions from table 2.2 to abstract statement, so we can use it later on to harvest flows and labels via recursion. This is done using the interface `Analyzable` as defined in figure 6.1. Specific objects has been created to hold all the definitions and statements. The class `StatementList` holds all the statements in the program. By simple asking for the `flow` of the `statementSet` the complete flow graph will be returned.

In order to perform the analysis an abstract interface `Lattice` has been created. This defines the basic elements for performing an analysis such as $\iota$ and $\perp$. For each of the tree analysis' a class that implements this interface has been created.

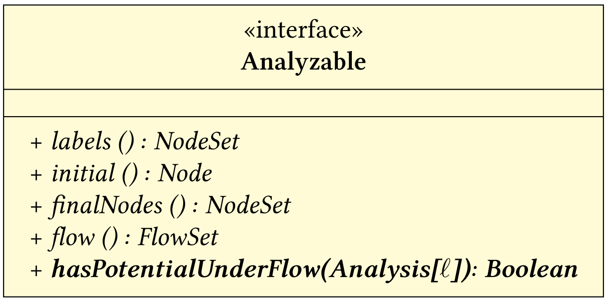

Figure 6.1: The Analyzable interface

.

- `RDLattice` - Reaching definitions analysis.
- `SignsLattice` - Detection of signs analysis.
- `IntervalLattice` - Interval analysis.

If a new similar analysis needed to be implemented I would be easy to add a new class implementing this interface.

### 6.2.1 Calculating a solution

In order to solve the equations provided by the analysis' the general worklist algorithm for computing the MFP solution has been implemented. The algorithm is placed as the method `execute` in a `Program`. It takes a specific analysis as a parameter and returns a result of the analysis for each label in the program as a result. All the flows are added to the worklist. In running time of the algorithm could be optimized by sorting the flow according to reverse post order. The transfer functions are called on the nodes during execution of the algorithm in order to calculate the result of the analysis. As an addition compared to how we defined the analysis, also the label of the node that the flows goes to is send when using the transfer function. This is done in order to have two different transfer functions for `if` and `while` statements. An `if` statement can then check if the flow is going to the true or the false branch and then use determine the new state by evaluating the condition. This makes a more precise solution.

# Chapter 7
# Improving precision

## 7.1 Overflow detection optimizations

The interval analysis we are performing for while-loops only takes account that the body is executed once, hence the result of the interval analysis performed in the body is an under approximation, since there is potential for that the body will be executed multiple times. However the obtained results are indeed a part of the precise result, but on the other hand we might omit some of the results for interval analysis which might result an unprecise result for detection of buffer overflow. If we could guarantee a number of iterations that the body would execute, we could do an optimization such as we could get a more exact result for the interval analysis by performing the analysis as the same number of iterations for the statements in the body. A reason for why we can not calculate the number of iterations for how many times the loop is executed is because it might be depended on the body, and it would require advanced analysis to calculate this number. Another approach for how the interval analysis could be performed for while-loop is to a assume that the body will run infinity often. However, during the analysis the statements of the body has to know that they are in the context of a while-loop. The result obtain form this analysis will be an over approximation.

## 7.2 Underflow detection

For underflow detection we've started by extending the Analyzable interface as illustrated in figure 6.1

We then loop through each label in the set of labels of the program, and if the test fails, we push the problematic node to a NodeSet which then can be used to inform the callee (probably Main procedure) of places where problems could arise. This problem is handled for each program statements as well for the boolean expressions as those can contain array operations. Table 7.1, Table 7.2 and Table 7.3 shows the functions for underflow detection.

## 7.3 Overflow detection optimizations

The interval analysis we are performing for while-loops only takes account that the body is executed once, hence the result of the interval analysis performed in the body is an under approximation, since there is potential for that the body will be executed multiple times. However the obtained results are indeed a part of the precise result, but on the other hand we might omit some of the results for interval analysis which might result an unprecise result for detection of buffer overflow. If we could guarantee a number of iterations that the body would execute, we could do an optimization such as we could get a more exact result for the interval analysis by performing the analysis as the same number of iterations for the statements

| Statement | Function |
|---|---|
| $[\mathrm{A}[\mathrm{a}_1] := \mathrm{a}_2]^\ell$ | $\mathrm{f}_\ell^{UF}(\widehat{\sigma}) = \begin{cases} true & \text{if } \{-\} \in \mathcal{A}_S[\![\mathrm{a}_1]\!]\widehat{\sigma} \vee \mathrm{f}_\ell^{UF}(\mathrm{a}_2, \widehat{\sigma}) \\ false & \text{otherwise} \end{cases}$ |
| $[\text{read A}[\mathrm{a}]]^\ell$ | $\mathrm{f}_\ell^{UF}(\widehat{\sigma}) = \begin{cases} true & \text{if } \{-\} \in \mathcal{A}_S[\![\mathrm{a}]\!]\widehat{\sigma} \\ false & \text{otherwise} \end{cases}$ |
| $[\text{write A}[\mathrm{n}]]^\ell$ | $\mathrm{f}_\ell^{UF}(\widehat{\sigma}) = \begin{cases} true & \text{if } \{-\} \in \mathcal{A}_S[\![\mathrm{n}]\!]\widehat{\sigma} \\ false & \text{otherwise} \end{cases}$ |
| $[\text{write a}]^\ell$ | $\mathrm{f}_\ell^{UF}(\widehat{\sigma}) = \mathrm{f}_\ell^{UF}(a, \widehat{\sigma})$ |
| $[\mathrm{b}]^\ell$ | $\mathrm{f}_\ell^{UF}(\widehat{\sigma}) = \mathrm{f}_\ell^{UF}(b, \widehat{\sigma})$ |
| $[\text{skip}]^\ell$ | $\mathrm{f}_\ell^{UF}(\widehat{\sigma}) = \text{false}$ |
| $[\text{read x}]^\ell$ | $\mathrm{f}_\ell^{UF}(\widehat{\sigma}) = \text{false}$ |
| $[\mathrm{x} := \mathrm{a}]^\ell$ | $\mathrm{f}_\ell^{UF}(\widehat{\sigma}) = \mathrm{f}_\ell^{UF}(a, \widehat{\sigma})$ |

Table 7.1: Underflow functions for statements

| Expression | Motivation |
|---|---|
| $\mathrm{f}_\ell^{UF}(\mathrm{x}, \widehat{\sigma}) = false$ | No underflow can happen in identifiers for variables. |
| $\mathrm{f}_\ell^{UF}(\mathrm{n}, \widehat{\sigma}) = false$ | Numerals don't underflow. |
| $\mathrm{f}_\ell^{UF}((\mathrm{a}), \widehat{\sigma}) = \mathrm{f}_\ell^{UF}(\mathrm{a}, \widehat{\sigma})$ | Parentheses expression merely forwards. |
| $\mathrm{f}_\ell^{UF}(\mathrm{a}_1 \mathrm{op}_a \mathrm{a}_2, \widehat{\sigma}) = \mathrm{f}_\ell^{UF}(\mathrm{a}_1, \widehat{\sigma}) \vee \mathrm{f}_\ell^{UF}(\mathrm{a}_2, \widehat{\sigma})$ | If either expression has an underflow, the entire expression overflows. |
| $\mathrm{f}_\ell^{UF}(-\mathrm{a}, \widehat{\sigma}) = \mathrm{f}_\ell^{UF}(\mathrm{a}, \widehat{\sigma})$ | Underflow is unaffected by negation. |
| $\mathrm{f}_\ell^{UF}(\mathrm{A}[\mathrm{a}], \widehat{\sigma}) = \begin{cases} true & \text{if } \{-\} \in \mathcal{A}_S[\![\mathrm{a}]\!]\widehat{\sigma} \\ false & \text{otherwise} \end{cases}$ | An index is defined as being non-negative. |

Table 7.2: Underflow functions for expressions

in the body. A reason for why we can not calculate the number of iterations for how many times the loop is executed is because it might be depended on the body, and it would require advanced analysis to calculate this number. Another approach for how the interval analysis could be performed for while-loop is to a assume that the body will run infinity often. However, during the analysis the statements of the body has to know that they are in the context of a while-loop. The result obtain form this analysis will be an over approximation.

Our code implementation of overflow detection is implemented by the `rangeCheck()` method which will loop through each label in the set of labels of the program, and if the test fails, we push the problematic node to a NodeSet which is retuned to the callee. The functions for how we are detecting overflows for statements, expressions and boolean expressions are shown in Table 7.4, Table 7.5 and in Table 7.6.

| Boolean expression | Motivation |
|---|---|
| $\mathrm{f}_\ell^{UF}(\mathrm{true}, \widehat{\sigma}) = false$ | No underflow can happen here |
| $\mathrm{f}_\ell^{UF}(\mathrm{false}, \widehat{\sigma}) = false$ | No underflow can happen here |
| $\mathrm{f}_\ell^{UF}((\mathrm{b}), \widehat{\sigma}) = \mathrm{f}_\ell^{UF}(\mathrm{b}, \widehat{\sigma})$ | Parentheses expression merely forwards. |
| $\mathrm{f}_\ell^{UF}(!\mathrm{b}, \widehat{\sigma}) = \mathrm{f}_\ell^{UF}(\mathrm{b}, \widehat{\sigma})$ | Negation merely forwards. |
| $\mathrm{f}_\ell^{UF}(\mathrm{b}_1 \mathrm{op}_b \mathrm{b}_2, \widehat{\sigma}) = \mathrm{f}_\ell^{UF}(\mathrm{b}_1, \widehat{\sigma}) \vee \mathrm{f}_\ell^{UF}(\mathrm{b}_2, \widehat{\sigma})$ | If either expression has an underflow, the entire expression underflows. |
| $\mathrm{f}_\ell^{UF}(\mathrm{a}_1 \mathrm{op}_r \mathrm{a}_2, \widehat{\sigma}) = \mathrm{f}_\ell^{UF}(\mathrm{a}_1, \widehat{\sigma}) \vee \mathrm{f}_\ell^{UF}(\mathrm{a}_2, \widehat{\sigma})$ | If either expression has an overflow, the entire expression overflows. |

Table 7.3: Underflow functions for boolean expressions

| Statement | Function |
|---|---|
| $[\mathrm{A}[\mathrm{a}_1] := \mathrm{a}_2]^\ell$ | $\mathrm{f}_\ell^{OB}(\widehat{\sigma}) = \begin{cases} true & \text{if } \mathcal{A}_I[\![\mathrm{a}_1]\!]\widehat{\sigma} \not\subseteq \mathcal{A}_I[\![\mathrm{A}]\!]\widehat{\sigma} \vee f_\ell^{OB}(\mathrm{a}_2, \widehat{\sigma}) \\ false & \text{otherwise} \end{cases}$ |
| $[\text{read A}[\mathrm{a}]]^\ell$ | $\mathrm{f}_\ell^{OB}(\widehat{\sigma}) = \begin{cases} true & \text{if } \mathcal{A}_I[\![\mathrm{a}]\!]\widehat{\sigma} \not\subseteq \mathcal{A}_I[\![\mathrm{A}]\!]\widehat{\sigma} \\ false & \text{otherwise} \end{cases}$ |
| $[\text{write A}[\mathrm{n}]]^\ell$ | $\mathrm{f}_\ell^{OB}(\widehat{\sigma}) = \begin{cases} true & \text{if } \mathcal{A}_I[\![\mathrm{n}]\!]\widehat{\sigma} \not\subseteq \mathcal{A}_I[\![\mathrm{A}]\!]\widehat{\sigma} \\ false & \text{otherwise} \end{cases}$ |
| $[\text{write a}]^\ell$ | $\mathrm{f}_\ell^{OB}(\widehat{\sigma}) = \mathrm{f}_\ell^{OB}(a, \widehat{\sigma})$ |
| $[\mathrm{b}]^\ell$ | $\mathrm{f}_\ell^{OB}(\widehat{\sigma}) = \mathrm{f}_\ell^{OB}(b, \widehat{\sigma})$ |
| $[\text{skip}]^\ell$ | $\mathrm{f}_\ell^{OB}(\widehat{\sigma}) = \text{false}$ |
| $[\text{read x}]^\ell$ | $\mathrm{f}_\ell^{OB}(\widehat{\sigma}) = \text{false}$ |
| $[\mathrm{x} := \mathrm{a}]^\ell$ | $\mathrm{f}_\ell^{OB}(\widehat{\sigma}) = \mathrm{f}_\ell^{OB}(a, \widehat{\sigma})$ |

Table 7.4: Overflow functions for statements

| Expression | Motivation |
|---|---|
| $\mathrm{f}_\ell^{OB}(\mathrm{x}, \widehat{\sigma}) = false$ | No overflow can happen in identifiers for variables. |
| $\mathrm{f}_\ell^{OB}(\mathrm{n}, \widehat{\sigma}) = false$ | Numerals don't overflow. |
| $\mathrm{f}_\ell^{OB}((\mathrm{a}), \widehat{\sigma}) = \mathrm{f}_\ell^{OB}(\mathrm{a}, \widehat{\sigma})$ | Parentheses expression merely forwards. |
| $\mathrm{f}_\ell^{OB}(\mathrm{a}_1 \mathrm{op}_a \mathrm{a}_2, \widehat{\sigma}) = \mathrm{f}_\ell^{OB}(\mathrm{a}_1, \widehat{\sigma}) \vee \mathrm{f}_\ell^{OB}(\mathrm{a}_2, \widehat{\sigma})$ | If either expression has an overflow, the entire expression overflows. |
| $\mathrm{f}_\ell^{OB}(-\mathrm{a}, \widehat{\sigma}) = \mathrm{f}_\ell^{OB}(\mathrm{a}, \widehat{\sigma})$ | Overflow is unaffected by negation. |
| $\mathrm{f}_\ell^{OB}(\mathrm{A}[\mathrm{a}], \widehat{\sigma}) = \begin{cases} \widehat{\sigma}[true & \text{if } \mathcal{A}_I[\![\mathrm{a}]\!]\widehat{\sigma} \not\subseteq \mathcal{A}_I[\![\mathrm{A}]\!]\widehat{\sigma} \\ \widehat{\sigma}[false & \text{otherwise} \end{cases}$ | An index is defined as being subset<br>of the interval for A. |

Table 7.5: Overflow functions for expressions

| Boolean expression | Motivation |
|---|---|
| $\mathrm{f}_{\ell}^{OB}(\mathrm{true}, \widehat{\sigma}) = false$ | No overflow can happen here |
| $\mathrm{f}_{\ell}^{OB}(\mathrm{false}, \widehat{\sigma}) = false$ | No overflow can happen here |
| $\mathrm{f}_{\ell}^{OB}((\mathrm{b}), \widehat{\sigma}) = \mathrm{f}_{\ell}^{OB}(\mathrm{b}, \widehat{\sigma})$ | Parentheses expression merely forwards. |
| $\mathrm{f}_{\ell}^{OB}(!\mathrm{b}, \widehat{\sigma}) = \mathrm{f}_{\ell}^{OB}(\mathrm{b}, \widehat{\sigma})$ | Negation merely forwards. |
| $\mathrm{f}_{\ell}^{OB}(\mathrm{b}_1 \mathrm{op}_b \mathrm{b}_2, \widehat{\sigma}) = \mathrm{f}_{\ell}^{OB}(\mathrm{b}_1, \widehat{\sigma}) \vee \mathrm{f}_{\ell}^{OB}(\mathrm{b}_2, \widehat{\sigma})$ | If either expression has an overflow, the entire expression overflows. |
| $\mathrm{f}_{\ell}^{OB}(\mathrm{a}_1 \mathrm{op}_r \mathrm{a}_2, \widehat{\sigma}) = \mathrm{f}_{\ell}^{OB}(\mathrm{a}_1, \widehat{\sigma}) \vee \mathrm{f}_{\ell}^{OB}(\mathrm{a}_2, \widehat{\sigma})$ | If either expression has an overflow, the entire expression overflows. |

Table 7.6: Overflow functions for boolean expressions

CHAPTER 8

# Trying it out

In this chapter we will provide some results from running the implemented tool on some concrete programs.

First we will consider the program listed below to determine the program slice:

```
x:=  0;
x:=  3;
if z=x then
   z:=0;
else
   z:=x;
fi
y:=  x;
x:=y+z;
```

Listing 8.1:

Figure 8.1: Reaching definition example

Using the tool to calculate the program slice gives the following output:

```
==== Program slice ====
POI: 1 result: [1]
POI: 2 result: [2]
POI: 3 result: [3]
POI: 4 result: [4]
POI: 5 result: [5, 2]
POI: 6 result: [6, 2]
POI: 7 result: [7, 4, 5, 6, 2]
```

Listing 8.2:

Figure 8.2: Output from analysis on code from figure 8.1

The result shows that the program slice for label 7 is the following labels 7, 4, 5, 6 and 2.

Next we consider the program below:

```
[i:=-1]^1
while [i<2]^2 do
  [a[i]:=i+1]^3
  [i:=i+1]^4
od
[a[-1]:=1]^5
if [a[i]=0]^6 then
  [skip]^7
else
  [skip]^8
fi
while [a[i]<0]^9 do
  [i:=i+a[3]]^10
od
```

Listing 8.3:

Figure 8.3: Juicy array example

Where `a` is defined as an array with two elements and `i` is in integer variable. The thing – or rather things – that make this program extra interesting is that is contains loads of errors. Jumping the gun, we can observe the output (see figure 8.4) our tool given this program as input.

```
==== Buffer Underflow =====
 Potential underflow detected at label: 3
 Potential underflow detected at label: 5
 Potential underflow detected at label: 6
 Potential underflow detected at label: 9
==== Interval =====
Range check failed at: 3
Range check failed at: 5
Range check failed at: 6
Range check failed at: 9
Range check failed at: 10
```

Listing 8.4:

Figure 8.4: Output from analysis on code from figure 8.3

We correctly detect the potential underflows on the assignments at label 3, 5, 6 and 9, which we can verify is correct by manually inspecting the code. What is more interesting however, is the fact that we detect the same labels with the interval analysis – plus a fifth one; the one with the overflow.

The attached folder with programs contains more programs that can be tested.

# Chapter 9 Conclusion

Program analysis is principles for analyzing source-code and derive information out of it which can give some useful information, before the code it actually executed. This is useful when debugging errors in the source code. It can also be used for optimizing the source code in various ways such as reducing the execution time or memory space consumptions.

We have during this project focused on various kinds of program analysis' in order to solve problems like buffer overflows and program slicing. Here we have used analysis as reaching definitions, detection of signs analysis and interval analysis, where each analysis has been implemented as instances of the Monotone framework, with complete lattices and a set of transfer functions. The results of the analysis has then been calculated by implementing the MFP (Maximum Fixed Point) algorithm.

The results we have obtained is documented throughout this report. We have also successfully implemented a program that is able to run the analysis explained in the report based on the theory gained in this course. Our analysis' are based on approximations that can be improved. For instance the lack of context awareness in the body of a while statement is limited by our under approximated solution.

# Appendices

# Appendix A How to run the application?

Open the command line from the application folder and type in:

java -jar AnalysisTool.jar $< programfile >$

Replace ” $< programfile >$ ” with the name of your while-language program file.

**Example**:
java -jar AnalysisTool.jar sample.lang

The output to the console will be formatted as following:

```
==== Program =====
(While-language code)

==== End program =====
==== Program slice ====
(Result of program slice)
==== Buffer Underflow =====
(Result of buffer underflow)
==== Interval =====
(Result of buffer overflow)
```

Listing A.1:

Figure A.1: Example program for demonstrating buffer overflow using sign detection

The first part will print the input source file with labels.
The second part will print the result of program slicing together with point of interest (POI)
The third part print results for found buffer overflows and finally the last part will print the results for buffer overflow.

# Appendix B

# Who did what?

This project has been created in a well functioning group where every member has been willing and able to contribute appropriately in all areas. The content of this chapter will not represent the actual work distribution as almost every section or file has been worked on by more than one person.

- **Ibrahim Nemli (s093477)** - Introduction, The WHILE language, Flow graph for program, Equations, Monotome framework(Sign detection), Example solution, Generalizing the analysis, Overflow detection optimizations, Overflow detection optimizations.
- **Kim Rostgaard Christensen (s084283)** - Abstract syntax tree data structure, Reaching Definitions analysis, Buffer underflow calculation algorithm, Monotome framework(Interval analysis), Underflow detection, Trying it out.
- **Peter Gammelgaard Poulsen (s093263)** - Flow graph data structure, Program graph data structure, Program slice calculation algorithm, Determine signs, Algorithm for calculating buffer overflow, Parsing the program, Functions for underflow detection.

# Bibliography

[1] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis.* Springer, 2005.

[2] Hanne Riis Nielson. *02242 slides.* 2013.