Abstract Interpretation of Unstructured Imperative Languages on Unbounded Domains

Pavle Subotic
Abstract

Abstract Interpretation of Unstructured Imperative Languages on Unbounded Domains

Pavle Subotic

In this thesis we present a novel program analysis technique that applies abstract interpretation to low-level intermediate languages with unbounded abstract domains. Unbounded abstract domains in program analysis occur in applications such as finding ranges of variables and its applications include elimination of assertions in programs, automatically deducing numerical stability, and array bounds checking. Unbounded abstract domains impose a major challenge in program analysis because it is difficult to ensure the termination of the analysis in the presence of program cycles.

State-of-the-art methods propose a technique that relies on a structured composition of programs, e.g., they permit abstract interpretation with unbounded abstract domains for program languages that have while and if statements but do not have goto statements.

This work demonstrates that unbounded abstract domains can be employed in program analysis for low-level intermediate languages commonly used in program language tools including compilers, integrated-development environments, and bug-checkers, where loops are implicit within the control flow graph.

Our work combines methods of elimination-based data flow analysis with abstract interpretation. We have implemented the proposed framework in the LLVM compiler framework as a program analysis pass and have conducted experiments with a suite of test programs to show the feasibility of our approach.
# Contents

<table>
<thead>
<tr>
<th>1 Introduction</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Background</td>
<td>14</td>
</tr>
<tr>
<td>2.1 Basic Domain Theory</td>
<td>14</td>
</tr>
<tr>
<td>2.1.1 Properties of Functions</td>
<td>15</td>
</tr>
<tr>
<td>2.1.2 Fixpoints</td>
<td>16</td>
</tr>
<tr>
<td>2.2 Abstract Interpretation</td>
<td>16</td>
</tr>
<tr>
<td>2.3 Data Flow Analysis</td>
<td>17</td>
</tr>
<tr>
<td>2.3.1 Properties of the Frameworks</td>
<td>20</td>
</tr>
<tr>
<td>2.3.2 Elimination-Based Data Flow Analysis</td>
<td>21</td>
</tr>
<tr>
<td>3 Abstract Domains</td>
<td>23</td>
</tr>
<tr>
<td>3.1 Analysis Property: Variable Ranges</td>
<td>23</td>
</tr>
<tr>
<td>3.1.1 Improving Bounds with Symbolic Intervals</td>
<td>25</td>
</tr>
<tr>
<td>3.2 Low-Level Language Syntax</td>
<td>27</td>
</tr>
<tr>
<td>3.3 Concrete Semantics</td>
<td>27</td>
</tr>
<tr>
<td>3.3.1 Concrete Denotational Semantics of Basic Blocks</td>
<td>28</td>
</tr>
<tr>
<td>3.3.2 An Operational Semantics of CFG</td>
<td>28</td>
</tr>
<tr>
<td>3.4 Abstract Semantics: Intervals</td>
<td>29</td>
</tr>
<tr>
<td>3.4.1 Interval Abstract Semantics of a Basic Block</td>
<td>29</td>
</tr>
<tr>
<td>3.4.2 Abstract Operational Semantics of CFG</td>
<td>31</td>
</tr>
<tr>
<td>3.5 Abstract Semantics: Symbolic Interval Domain</td>
<td>32</td>
</tr>
<tr>
<td>3.5.1 Interval Abstract Semantics of a Basic Block</td>
<td>32</td>
</tr>
<tr>
<td>4 Abstract Interpretation of Unstructured Languages</td>
<td>36</td>
</tr>
<tr>
<td>4.1 Finding Fixpoints with Existing Techniques</td>
<td>36</td>
</tr>
<tr>
<td>4.2 Abstract Interpretation in an Elimination Framework</td>
<td>38</td>
</tr>
<tr>
<td>4.2.1 Elimination</td>
<td>40</td>
</tr>
<tr>
<td>4.2.2 Solving</td>
<td>41</td>
</tr>
<tr>
<td>4.3 Correctness of Framework</td>
<td>42</td>
</tr>
<tr>
<td>5 Implementation</td>
<td>46</td>
</tr>
<tr>
<td>5.1 Architecture</td>
<td>47</td>
</tr>
<tr>
<td>5.2 LLVM Range Analysis Pass</td>
<td>48</td>
</tr>
<tr>
<td>5.3 Function Space</td>
<td>51</td>
</tr>
<tr>
<td>5.4 Domains and Environments</td>
<td>54</td>
</tr>
<tr>
<td>5.5 Implementation Summary</td>
<td>56</td>
</tr>
</tbody>
</table>
6 Results

6.1 Case Study 1: Branching .......................................................... 58
6.2 Case Study 2: Irreducible Graphs .............................................. 59
6.3 Case Study 3: Widening and Narrowing ................................. 60
6.4 Case Study 4: Bubble Sort ...................................................... 61
6.5 Case Study 5: Symbolic Elimination ...................................... 63
6.6 Case Study 6: Symbolic Elimination with Branching ............... 65
6.7 Results Summary ................................................................. 66

7 Related Work .............................................................................. 67

7.1 Abstract interpretation in Literature ...................................... 67
   7.1.1 Range Analysis based Abstract Interpretation ................. 67
   7.1.2 Loop Detection ............................................................... 68

7.2 Abstract Interpretation-Based Tools ..................................... 68

8 Conclusion .................................................................................. 69

8.1 Summary .......................................................... 69
8.2 Future work ................................................................. 69
8.3 Reflection ................................................................. 70

A LLVM Analysis Output ............................................................. 71

B Abstract Interpretation Proofs .................................................. 85

B.1 Monotonicity ................................................................. 85
   B.1.1 Monotonicity of statements ........................................ 85

B.2 Galois Connection ............................................................ 87
   B.2.1 Monotonicity of γ and α ................................................ 87
   B.2.2 Galois connection for intervals .................................... 88

B.3 Soundness proof .............................................................. 88
   B.3.1 Initial State ................................................................. 89
   B.3.2 Soundness of Semantic Functions ............................... 89
   B.3.3 Statements .............................................................. 90
   B.3.4 Statement list .......................................................... 90
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Motivating Example</td>
<td>10</td>
</tr>
<tr>
<td>1.2 Design of the Analysis in LLVM</td>
<td>12</td>
</tr>
<tr>
<td>1.3 Running LLVM opt</td>
<td>12</td>
</tr>
<tr>
<td>2.1 Widening and Narrowing</td>
<td>18</td>
</tr>
<tr>
<td>2.2 MFP Algorithm</td>
<td>19</td>
</tr>
<tr>
<td>2.3 Solutions to DFA</td>
<td>20</td>
</tr>
<tr>
<td>3.1 Motivating Example</td>
<td>24</td>
</tr>
<tr>
<td>3.2 Syntactic Algebra</td>
<td>27</td>
</tr>
<tr>
<td>3.3 Syntactic Domains</td>
<td>27</td>
</tr>
<tr>
<td>3.4 Valuation Functions</td>
<td>28</td>
</tr>
<tr>
<td>3.5 Abstract Valuation Functions</td>
<td>30</td>
</tr>
<tr>
<td>3.6 Meet and Join Operators</td>
<td>31</td>
</tr>
<tr>
<td>3.7 Interval Arithmetic Functions</td>
<td>31</td>
</tr>
<tr>
<td>3.8 Symbolic Abstract Valuation Functions</td>
<td>33</td>
</tr>
<tr>
<td>3.9 Symbolic Expressions</td>
<td>34</td>
</tr>
<tr>
<td>3.10 Symbolic Expressions</td>
<td>34</td>
</tr>
<tr>
<td>4.1 While Program in a Structured Language</td>
<td>37</td>
</tr>
<tr>
<td>4.2 While Program in a Unstructured Language</td>
<td>37</td>
</tr>
<tr>
<td>4.3 Converging Analysis</td>
<td>40</td>
</tr>
<tr>
<td>4.4 Diverging Analysis</td>
<td>40</td>
</tr>
<tr>
<td>4.5 Converging Fixpoint Computation</td>
<td>42</td>
</tr>
<tr>
<td>4.6 Diverging Fixpoint Computation</td>
<td>42</td>
</tr>
<tr>
<td>4.7 Widening and Narrowing on a While Program</td>
<td>43</td>
</tr>
<tr>
<td>5.1 High-Level Implementation Diagram</td>
<td>46</td>
</tr>
<tr>
<td>5.2 Component Dependency Diagram</td>
<td>47</td>
</tr>
<tr>
<td>5.3 Equation System Construction</td>
<td>49</td>
</tr>
<tr>
<td>5.4 Range Analysis Class Diagram</td>
<td>51</td>
</tr>
<tr>
<td>5.5 Range Analysis Class Diagram</td>
<td>54</td>
</tr>
</tbody>
</table>
Listings

1.1 A Bubble Sort Example ............................................................... 9
2.1 A C Implementation of the Gaussian Elimination Method ................. 22
3.1 A simple range analysis ............................................................. 24
3.2 Range Analysis with a Function Call ........................................... 26
3.3 Symbolic Range Analysis with Odd and Even Values ......................... 26
5.1 Creating of the Equation System .................................................. 48
5.2 A Visitor Function for the LLVM ICmp Instruction .......................... 48
5.3 Creating of the Equation System .................................................. 50
5.4 Creating of the Equation System .................................................. 50
5.6 Solving in Fixpoints ................................................................... 52
5.7 Solving in Fixpoints ................................................................... 53
5.8 Max GiNaC class ...................................................................... 55
5.9 A Rewrite Rule for the Min Class ................................................... 55
6.1 A Branching Example .................................................................. 58
6.2 A Irreducible Graph Example ....................................................... 59
6.3 A Widening and Narrowing Example ............................................. 60
6.4 A Bubble Sort Example ............................................................... 61
6.5 A Symbolic Interval Example ......................................................... 63
6.6 A Symbolic Interval Example with Branching .................................... 65
A.1 A Branching Example ................................................................ 71
A.2 Results ...................................................................................... 71
A.3 A Irreducible Example ................................................................ 73
A.4 Results ...................................................................................... 73
A.5 A Widening/Narrowing Example .................................................. 75
A.6 Results ...................................................................................... 75
A.7 The Motivating Example .............................................................. 77
A.8 Results ...................................................................................... 77
A.9 Symbolic Interval Example .......................................................... 81
A.10 Results ..................................................................................... 81
A.11 Symbolic Interval Example .......................................................... 83
A.12 Results ..................................................................................... 83
Chapter 1

Introduction

This thesis presents a technique for determining lower and upper bound values for program variables in an imperative program. Such an analysis is known as a range analysis [83]. Range analysis has various applications in program analysis including assertion elimination [1], determining the numerical stability of algorithms [11], eliminating array out of bound checks [17], and integer overflow detection [81]. The technique presented in this thesis combines the theories of abstract interpretation [12] and data flow analysis [38] to obtain precise bounds for variables in an unstructured low-level language i.e. a low-level language with goto statements. We have implemented our technique as a program analysis pass in the LLVM framework [42] and have conducted experiments with a suite of test programs to show the feasibility of our approach.

```
int i = 0;
int k = 0;
int arr[5] = {9, 5, 2, 3, 1};

while (i < 5) {
    int j = 0;
    while (j < 5) { // invariant 1: (j >= 0 & j <= 3) does not hold!
        if (arr[j] > arr[j+1]){
            swap(j, j+1); k++;
        }
        j++;
    }
    i++;
} // invariant 2: (i == 5 & j >= 4 & k <= 5*5);
```

Listing 1.1: A Bubble Sort Example

In Listing 1.1 we present an example of a range analysis application. An incorrect implementation of a bubble sort program is shown in the listing. The program traverses an array multiple times, swapping every adjacent item that is in the wrong order. However, the array access on line 8, namely arr[j+1] will “overshoot” the maximum array index of 4 during the last loop iteration. The correct behaviour is described by invariant (1) which will be violated when j is equal to 4.

The design of program analysis frameworks to detect program errors, such as the error depicted in Listing 1.1 is an ongoing research challenge [2, 21, 43, 61]. These program analysis frameworks are automated tools that determine the satisfaction of inter program properties. They are useful for software verification, program optimisation and software debugging. Program analysis tools are classified as either dynamic or static. Dynamic program analysis performs an analysis at run-time as a side-effect of the program execution. It has absolute precision as it gathers information on the ac-
tual path of a program’s execution. The analysis, however, is performed only for a single program input. Generally, it must be re-run for an exceedingly large number of inputs to observe the errant behaviour. Dynamic program analysis imposes high overheads, for example, in inserting instrumentation code for performing run-time checks and collecting run-time meta-data \cite{72}. The introduced overhead may alter program behaviour which may result in an incorrect analysis. Another drawback of dynamic program analysis is that it allows only a postmortem analysis to be conducted. It is thus not applicable for mission-critical systems, where unexpected termination has a catastrophic consequence. For these reasons, the dynamic approach is implemented mainly in tools that target non-critical applications \cite{56,14}.

```
int I, k = 0
int arr[5] = ...
if i < 5
int j = 0
if j < 5
i++
invariant (2)
j++;
invariant (1)
if arr[j] > arr[j+1]
swap(j, j+1)
k++
```

*invariant (1) is that j >= 0 && j <= 3
*invariant (2) is that i == 5 && j >= 4 && k <= 25*2

**Figure 1.1**: Flow graph and analysis results of motivating example

*Static program analysis* on the other hand, discovers properties of programs at compile-time on program source code. It formally establishes whether program properties hold on all possible paths of execution. This is advantageous because all program inputs are considered. However, in the case of infinite path programs, namely, programs with loops, asserting the fulfilment of non-trivial properties becomes undecidable \cite{63}. As a result, infinite path analyses are abstracted to find decidable properties. By abstracting the properties, however, information is lost and so abstraction comes at the expense of precision. Static program analysis is particularly well-suited for critical systems \cite{11,31,41} due to its static nature and its coverage of all program paths. Several types of static program analysis techniques exist, including model checking \cite{4}, abstract interpretation \cite{12}, constraint solving \cite{24}, and data flow analysis \cite{38}. Typically, in static program analysis a program is converted into a Control Flow Graph (CFG), shown on the left hand side of Figure 1.1. A CFG is a representation of a program that models all possible paths a program may execute. It is composed of several basic blocks that represent groups of sequential program statements. Basic blocks are connected by edges, where edges represent the flow of control from one block to another. Several edges coming out of a block indicate a fork in the control flow of the program. As shown in Figure 1.1, block (b3) is code that is executed
if the condition at (b2) holds, and (b8) is executed if the condition does not hold. Likewise condition 
(b3) is true in (b3) and false in (b6). Condition (b4) is true for (b5) and false for (b7). For our example 
our program analysis framework will detect that invariant (1) is false because \( j \) becomes too large. If 
this program anomaly is left undetected, an array-out-of-bound access will occur at run-time. 

A poorly implemented static analysis tool may not terminate, or may take a very long time to 
terminate in situations. One situation where this may occur is if we have no knowledge of what pro-
grams points are inside a loop. In this case, we will not be able to detect the cycle induced by the loop. 
Termination is however, a desired property for any program analysis. The fact that all execution paths 
must be considered when there are infinite paths causes a divergence in our analysis. As outlined 
in Rice’s theorem \[63\], proving non-trivial properties in any given program (assuming the program-
ning language is Turing complete) is undecidable. Static analysis addresses this by abstracting the 
semantics of the program to a decidable and sound program representation that ensures termination. 

By abstracting the semantics of a programming language, we abstract away the complexity of the 
properties that may be examined. For example, the property what value a program variable has, may 
be abstracted to the property what sign a program variable has, where sign means positive, negative 
or zero. In the analysis in Figure 1.1 the abstraction is what range of values does a program variable 
have. This understanding of static program analysis was formalised by Cousot and Cousot through 
the theory of abstract interpretation \[12\]. However, the operational aspects existed much earlier in the 
form of iterative data flow analysis \[38\]. Abstract interpretation explicitly formalises the connection 
between the real or concrete semantics of a program and its abstract semantics. It can be viewed as 
an execution of an abstracted computer program on all program paths and permits an analysis on un-
bounded domains. Iterative data flow analysis, on the other hand, relies on bounded domains and by 
virtue of Tarski’s fixpoint theorem \[78\] termination is guaranteed. In iterative methods the flow inform-
ation is first initialised to an unsafe value. The analysis iterates through each basic block, gathering 
information until no further information can be gathered, resulting in a solution. Abstract interpreta-
tion and data flow analysis, cannot adequately perform the analysis in Figure 1.1 for different reasons. 
The iterative approach takes a long time to terminate in the example. This is due to the fact that our 
abstraction is unbounded (the range of integers have no bounds). The iterative approach will continue 
to iterate without terminating. For example, in the program in Listing 1.1 data flow analysis would not 
be able to detect the bound of \( k \). Furthermore, it would continue to iterate until the maximum value for 
an integer. For deeply nested loops, this could take a long time to terminate. Abstract interpretation 

improves on the iterative data flow approach by introducing the theory of widening/narrowing \[10\]. 
It allows an analysis to incorporate unbounded domains by providing a method to accelerate con-
vergence to an approximation of a fixpoint. Widening/narrowing, however, cannot be applied on 
unstructured low-level languages. In practice, it relies on programming languages to contain explicit 
syntactical constructs to identify loops, something data flow-level languages do not have. For input 
in the form of a data flow level-language or a CFG as in Figure 1.1 there is no syntactic loop con-
struct and abstract interpretation cannot detect which variables are in a loop. However, an increasing 
number of compilers have multiple front-ends. For example, the GCC compiler has multiple front-end 
languages that are translated into a Register Transfer Language (RTL) \[62\] before optimising and gen-
erating target back-end assembly. The .Net framework likewise has multiple front-ends, while using 
the CIL intermediate language \[47\] as an intermediary. Various applications use the Java byte code as 
a intermediate language \[29\] and Apple uses the LLVM language as an intermediate language in all

\[1\] In any reasonable amount of time.

\[2\] It is bounded in practice by the maximum representation of a number, for Int in a C program this would be defined by 
MININT and MAXINT, but still the analysis would take a very long time to terminate.
its products [34]. For such compiler frameworks unbounded abstract interpretation analyses introduce redundancy and inconsistencies as the same analysis must be implemented on several front-ends.

In this thesis we present a solution that combines elimination-based data flow analysis with abstract interpretation. Elimination-based data flow analysis is a technique that solves data flow equations by algebraic means. The key feature of this approach is that loops are identified by the analysis itself. Our solution embeds widening and narrowing inside algebraic operations of the elimination-based data flow analysis. This enables unbounded abstract domains to be employed in a program analysis for low-level intermediate languages that have no syntactic loop constructs.

We implemented our technique as a range analysis pass in the LLVM compiler. LLVM is an increasingly popular compiler framework with an intermediate representation language that is used as an intermediate representation (IR) for many high-level languages. A large number of popular languages can now be compiled to LLVM’s IR through compilers or language bindings [28, 57, 58]. We implemented our range analysis using intervals and symbolic intervals. For the implementation we used an efficient elimination-based algorithm [75]. We used GiNaC [20], a flexible Computer Algebra System (CAS) library, for the implementation of symbolic intervals. Our range analysis is easily extended for a variety of program analyses, including assertion elimination, the detection of numerical instability, array bounds checking and integer overflow.

Figure 1.2 outlines the build structure of our implementation. We integrated our range analysis into the LLVM analysis pass framework using the elimination framework and GiNaC external libraries. After the build this will result in the LLVM opt binary. In Figure 1.3 we see that a C program such as the one in Figure 1.1 is first compiled to LLVM byte-code. The LLVM byte-code is run through opt to produce an analysis output on the standard output of the terminal.

The thesis contributions are summarised as follows:
1. Introduce a method for conducting an abstract interpretation for unstructured low-level languages on unbounded domains.

2. Demonstrate this technique on two unbounded domains, namely discrete intervals and symbolic intervals.

3. Implement the technique as an analysis in the LLVM compiler framework.

This thesis is structured as follows; first we present the fundamentals of program analysis in Chapter 2. In Chapter 3 we describe two unbounded domains that are used to demonstrate the new approach to abstract interpretation. In Chapter 4 a method of abstract interpretation on a low-level language is proposed. In Chapter 5 we present the LLVM implementation. In Chapter 6 we present several case studies that demonstrate feasibility of the analysis. In Chapter 7 some related work in the field is examined and the thesis concludes in Chapter 8.
Chapter 2

Background

This chapter of the thesis outlines background material fundamental to range analysis and program analysis. A brief introduction to domain theory is presented along with the three program analysis techniques that are required to understand the thesis solution, namely, abstract interpretation, iterative data flow analysis and elimination-based data flow analysis.

2.1 Basic Domain Theory

Program analysis discovers program properties. These properties are represented by algebraic structures called partially ordered sets (posets). This representation allows comparisons between elements of the structure’s base set. For example, the power set of all natural numbers can have a partial order where elements are compared by containment. In such a poset the element which is the set of all natural numbers is greater than the element which contains only a single number. In a partial order, however, not all elements can be compared. For example, the set containing the number 3 cannot be compared with the set containing the number 1. Such elements are called incomparable elements. Several fundamental definitions used in this thesis are defined below. The reader may consult [25, 77] for a more in-depth study of basic domain theory.

Definition 2.1.1. (Poset) A poset is defined as a set $\mathcal{L}$ for which a binary relation $\sqsubseteq$ is defined, which satisfies the following axioms, where $l_1, l_2, l_3$ be any element of $\mathcal{L}$. The relation must be:

- Reflexive - $l_1 \sqsubseteq l_1$.
- Transitive - if $l_1 \sqsubseteq l_2$ and $l_2 \sqsubseteq l_3$ then $l_1 \sqsubseteq l_3$.
- Anti-symmetrical - $l_1 \sqsubseteq l_2$ and $l_2 \sqsubseteq l_1$ then $l_1 = l_2$.

We omit the subscript $\mathcal{L}$ from $\sqsubseteq$ and instead simply write $\sqsubseteq$ when the lattice $\mathcal{L}$ is clear from the context.

Definition 2.1.2. (Least Element) Let $\mathcal{L}$ be a poset. We say $a \in \mathcal{L}$ is the least element of $\mathcal{L}$ iff $\forall x \in \mathcal{L}. a \sqsubseteq x$.

Definition 2.1.3. (Greatest Element) Let $\mathcal{L}$ be a poset. We say $a \in \mathcal{L}$ is the greatest element of $\mathcal{L}$ iff $\forall x \in \mathcal{L}. x \sqsubseteq a$. 
Definition 2.1.4. (Least Upper Bound) Let $L$ be a poset and $L' \subseteq L$. Then $a \in L$ is an upper bound of $L'$ iff $\forall x \in L'.x \sqsubseteq a$ and $a$ is the least upper bound (lub) iff $a \sqsubseteq y$ for all upper bounds $y$ of $L'$. We denote the least upper bound as $\sqcap_L L'$.

Definition 2.1.5. (Greatest Lower Bound) Let $L$ be a poset and $L' \subseteq L$, then $a \in L$ is a lower bound of $L'$ iff $\forall x \in L'.a \sqsubseteq x$ and $a$ is the greatest lower bound (glb) iff $y \sqsubseteq a$ for all lower bounds $y$ of $L'$. We denote the greatest lower bound as $\sqcup_L L'$.

$\sqcap_L$ is called the meet operator, and $\sqcup_L$ is called the join operator. We write $\sqcap_L \{l_1, l_2\}$ as $l_1 \sqcap_L l_2$ and $\sqcup_L \{l_1, l_2\}$ as $l_1 \sqcup_L l_2$. We would also omit the subscript $L$ when it is clear from the context.

Definition 2.1.6. (Lattice) A lattice $L$ is a poset such that for any two elements, $l_1$ and $l_2 \in L$ there is a greatest lower bound and a least upper bound.

Definition 2.1.7. (Complete Lattice) A complete lattice is a lattice $L$ such that, $\forall L' \subseteq L$. $L'$ has a least upper bound and a greatest lower bound.

We represent the least upper bound and the greatest lower bound of a complete lattice $L$ as $\sqcap_L$ and $\sqcup_L$. Again, we often omit the subscripts. We also represent complete lattice $L$ together with its components as a tuple $(L, \sqsubseteq, \sqcap, \sqcup, \bot, \top)$.

Observation 2.1.1. (Finite Lattice) We note that every finite lattice must be complete but not every complete lattice is finite.

Definition 2.1.8. (Chain) Let $L$ be a poset, then $L$ is a chain iff for any two elements $l_1, l_2 \in L$ either $l_1 \sqsubseteq l_2$ or $l_2 \sqsubseteq l_1$.

We write a chain $l_1 \sqsubseteq l_2 \sqsubseteq l_3 \ldots$ as $(l_n)_n$.

Definition 2.1.9. (Domain) A domain \cite{domain} is a lattice.

Definition 2.1.10. (Unbounded Domain) An unbounded domain is a lattice with out a glb.

2.1.1 Properties of Functions

A function is a relationship between two sets. It maps an element in one set to an element in another set. The input set of a function is called the domain (not to be confused with the algebraic structure domain) and the output set is called the range. Two important properties functions in poset algebras often have is monotonicity and distributivity.

Monotonicity is a property that ensures that the relation between the two inputs is maintained after they are both transformed by the same function.

Definition 2.1.11. (Monotonicity) A function $f : L_1 \to L_2$, where $L_1$ and $L_2$ are posets, is monotonic, iff given two elements, $d_1$ and $d_2$ of the poset $L_1$, we have:

$$d_1 \sqsubseteq_{L_1} d_2 \to f(d_1) \sqsubseteq_{L_2} f(d_2).$$

Distributivity is an even stronger property. It guarantees that when there is a merging of information no precision is lost. Note that distributivity implies monotonicity.

Definition 2.1.12. (Distributivity) A function $f : L_1 \to L_2$, where $L_1$ and $L_2$ are posets, is distributive iff given two elements, $d_1$ and $d_2$ of the poset $L_1$, we have:

$$f(d_1 \sqcap_{L_1} d_2) = f(d_1) \sqcap_{L_2} f(d_2).$$
2.1.2 Fixpoints

In program analysis we typically deal with recurrence equations or relations. The solution to such equations is called a fixpoint solution. Consider a monotone function \( f : L \to L \) on a complete lattice \( L \). We write the set of fixpoints of a function as \( \text{Fix}(f) = \{ l \mid f(l) = l \} \). Now, the set which \( f \) is reductive at its elements is given by \( \text{Red}(f) = \{ l \mid f(l) \subseteq l \} \), and the set at which \( f \) is extensive at its elements is given by \( \text{Ext}(f) = \{ l \mid l \subseteq f(l) \} \).

An important theorem is the Knaster-Tarski theorem. It guarantees the existence of at least one fixed point of a function \( f \) on any complete lattice.

**Theorem 2.1.1. (Knaster-Tarski)** If \( L \) is a complete lattice and \( f \) is a monotonic function on \( L \), then \( f \) has a fixpoint and the set of fixpoints of \( f \) has a maximal element.

Intuitively this theorem states that if we move up (or down) a chain in the lattice and the lattice is bound we will eventually hit this bound which will result in a fixpoint. We refer the reader to [5] for a well-explained proof of the theorem.

2.2 Abstract Interpretation

Abstract interpretation [12] is a general theory for constructing conservative approximations of programming language semantics and inferring dynamic program properties for several or all program runs [6]. In abstract interpretation concrete program data is replaced with abstract data and the concrete semantics are replaced with abstract semantics. By simplifying certain properties which are undecidable or are very expensive to decide, we can define easily decidable properties. This however, comes at the expense of precision and may result in situations where there are false negatives. For example, an analysis that aims to discover the sign of each variable defines a decidable abstract semantics where the domain is no longer the unbounded value domain but a simpler bounded sign domain. In the abstract semantics the addition operation would result in a positive for two positive numbers, and a negative for two negative values. Despite being computable, the abstract semantics results in the loss of precision. In the sign example, positive and negative arguments to the addition operation would result in an unknown result. A sound analysis must report all potential errors even though there is a possibility that the result might be a positive expression for many values in the concrete domain. If the property of the analysis is to determine whether all expressions that are positive, a positive and negative addition must therefore be deemed an error, even though it may not be.

To construct an abstract interpretation the two semantics must relate to one another so that the abstract meaning can be understood in terms of the concrete meaning. This abstract relation is defined as an isomorphism called the Galois connection [12]. It translates the abstract domain to the concrete domain and vice versa. The Galois connection can be seen as two functions \((\alpha, \gamma)\) that map each domain to the other. We define \( \alpha \) to be an abstraction function that extracts information in the abstract domain from the program’s concrete domain. We define \( \gamma \) to be the concretisation function which produces all the possible concrete values that could lie in the corresponding approximated abstract value.

**Definition 2.2.1. (Galois Connection)** Let \((L_1, \sqsubseteq_1, \sqcap_1, \sqcup_1)\) and \((L_2, \sqsubseteq_2, \sqcap_2, \sqcup_2)\) be two lattices. A pair of monotonic functions \( \alpha : L_1 \to L_2 \) and \( \gamma : L_2 \to L_1 \) is a Galois connection iff

\[
\forall x_1 \in L_1, \forall x_2 \in L_2, \alpha(x_1) \sqsubseteq_2 x_2 \iff x_1 \sqsubseteq_1 \gamma(x_2)
\]
An additional feature of abstract interpretation is the ability to accelerate fixpoint convergence. This is described in the theory of widening and narrowing \[12\]. Widening and narrowing speeds up termination of an analysis when in a recursive semantic function such as a while loop.

Widening helps approximate least fixpoints. A widening operator is defined as follows:

**Definition 2.2.2.** (Widening) \( \nabla : \mathcal{L} \times \mathcal{L} \rightarrow \mathcal{L} \) where \( \mathcal{L} \) is a lattice and the following holds:

- \( \nabla \) is a widening operator if and only if it is an upper bound operator
- for all ascending chains \((l_n)_n\) the ascending chain \((l_n^{\nabla})_n\) eventually stabilises.

Narrowing attempts to find a better fixpoint approximation after widening. A narrowing operator is defined as follows:

**Definition 2.2.3.** (Narrowing) \( \Delta : \mathcal{L} \times \mathcal{L} \rightarrow \mathcal{L} \) where \( \mathcal{L} \) is a lattice and the following holds:

- \( \Delta \) is a widening operator if and only if \( l_2 \sqsupseteq l_1 \rightarrow l_2 \sqsupseteq (l_1 \Delta l_2) \sqsupseteq l_1 \), for all \( l_1, l_2 \in \mathcal{L} \)
- for all descending chains \((l_n)_n\) the sequence \((l_n^{\Delta})_n\) eventually stabilises.

The theory of widening enables abstract interpretation to not only accelerate convergence for particularly high domains, it also allows an analysis to be performed on infinite domains, given that the semantics of any loop is defined within the language.

As we describe Section 2.1, Tarski’s theorem (See definition 2.1.1) guarantees a fixpoint will be reached given a complete lattice and a monotone function. In the case that the domain lattice is particularly high or not bounded this could take a large or infinite amount of time.

Figure 2.1 illustrates what widening and narrowing achieves in the lattice structure.

### 2.3 Data Flow Analysis

*Data flow analysis* can be seen as an instance of abstract interpretation for imperative programs. It is a technique that collects execution time information about a program at compile time before program execution (statically). Data flow analysis is performed on a directed graph called a *control flow graph* (CFG).

**Definition 2.3.1.** (CFG) A CFG is a finite directed graph \( G(V, E, s) \), with \( V \) a set of nodes, \( E \) a set of edges, and \( s \) an initial node, where every node in \( V \) is reachable by a path from \( s \). A CFG node, referred to as a block, is a maximal list of sequential program instructions that reaches the end of the program or is terminated by a branching instruction. We refer to all nodes that have an incoming edge to a node \( x \) as the predecessors of \( x \), denoted pred(\( x \)). We refer to all nodes that have an incoming edge from node \( x \) as the successors of \( x \), denoted succ(\( x \)).

In data flow analysis, problems are reduced to a system of flow equations. Each equation can be seen as a relationship between the output \( (out_b) \) information valid after the basic block has been processed and the input information \( (in_b) \) before the basic block has been processed. This relationship is established by a monotone *flow function* \( f_b \), that transforms the input information to some output information. In general, a data flow framework consists of a set of equations with a distinct *boundary value equation* (init) that defines values at the entry (or exit) blocks of the program.

What constitutes input and output depends on the direction of the analysis. A forward problem starts from the first statement of the initial basic block and follows the graph in the direction of the final statement. The backward graph does just the opposite, it follows the graph in the opposite direction of program execution. We define a forward DFA and backward DFA as follows:
Figure 2.1: A Graphical Representation of Widening and Narrowing
Require: $\forall b \in V. \text{out}_b = \top$

while (Changes to out for some b) do
  for each ($b \in V$) do
    $\text{in}_b \leftarrow \bigcap_{x \in \text{pred}(b)} \text{out}_x$
    $\text{out}_b \leftarrow f_b(\text{in}_b)$
  end for
end while

Figure 2.2: MFP Algorithm

**Definition 2.3.2. (Forward DFA)** In a forward DFA flow-equations have the form:

$$\text{in}_b = \begin{cases} \bigcap_{b'\in\text{pred}(b)} \text{out}_{b'} & \text{if pred}(b) \neq \emptyset \\ \text{init} & \text{otherwise.} \end{cases}$$

**Definition 2.3.3. (Backward DFA)** In a backward DFA flow-equations have the form:

$$\text{out}_b = \begin{cases} \bigcap_{b'\in\text{succ}(b)} \text{in}_{b'} & \text{if succ}(b) \neq \emptyset \\ \text{init} & \text{otherwise.} \end{cases}$$

Popular forward data flow analysis include constant propagation analysis reaching definitions analysis [37] and common sub expression analysis [37]. Backwards problems include liveness analysis [37] which is frequently used for register allocation in modern compilers. It is important to note that forward and backward data flow analysis are equivalent and can be transformed into one another by reversing the direction of the flow edges and inverting the flow functions.

There are numerous methods of finding solutions to a system of data flow equations. The Meet-Over-All-Paths (MOP) solution meets over all possible paths from the start node to a given point. A path here is assumed to be a sequence of basic blocks (CFG nodes), where there is an edge in the CFG connecting a block with the subsequent block. The length of the path is the number of the basic blocks in the path. As an example, let us consider a path of execution that begins at basic block $s$ and ends at some final basic block $b$. There are most likely several different paths that reach the basic block $b$. The set of all paths from $s$ to $b$ is denoted $\Pi(b)$. Given $p \in \Pi(b)$, we define $f_p$ to be the function composition $f_b \cdot \ldots \cdot f_s$ corresponding to the basic blocks in the path $p$. Here we assume that $f_s = \top$. Now, the MOP solution meets all the different paths at $b$. However, the MOP solution is only computable in instances where there are finite paths (no loops) and is generally uncomputable in most programming languages. For this reason it is usually not possible to attain a MOP solution for cyclic programs. We define MOP for all paths that end in the basic block $b$ as follows:

**Definition 2.3.4. (MOP Solution)** $\text{MOP}(b) = \bigcap_{p \in \Pi(b)} f_p$.

The maximum fixpoint (MFP) approximates the MOP solution. The maximum fixpoint performs a meet as soon as two paths merge and keeps iterating over the CFG until a fixpoint is reached. Essentially, this means merging all paths into one data flow equation and repeating this process until we hit a fixpoint. This is guaranteed by Knaster-Tarski’s fixpoint theorem (Theorem 2.1.1), if we have a monotonic data flow analysis framework with a domain that is a complete lattice. The maximum-fixpoint algorithm is described in Fig. 2.2. In the case of distributive data flow problems the analysis can achieve a MOP solution as $\text{MFP} = \text{MOP}$ and computing the MFP will give the same result as an MOP solution.
Different from the two mentioned types of solutions, the ideal solution IDEAL is similar to the MOP solution, but it only meets all executable paths. However, this set is not computable. Generally speaking, the solutions for a data flow analysis have the following relation: \( \text{FP} \subseteq \text{MFP} \subseteq \text{MOP} \subseteq \text{IDEAL} \). As pointed out, when the flow functions are distributive \( \text{MFP} = \text{MOP} \). Figure 2.3 summarises the accuracy of each type of solution. The higher up the lattice the solution is, the more precise it is.

### 2.3.1 Properties of the Frameworks

In this section we prove that for a data flow framework the MFP solution is an approximation of a MOP solution and for a distributive framework, \( \text{MFP} = \text{MOP} \). The reader may refer to [5] for a summary of the proofs.

**Theorem 2.3.1.** If \((\mathcal{L}, \cap, f)\) and \((G, s, e)\) constitute an instance of a monotonic data flow analysis framework, then \( \text{MFP} \subseteq \text{MOP} \).

**Proof.** We let \( \Pi(b) \) denote the set of all paths from \( s \) to \( b \), and \( \Pi_f(b) \) denote the set of paths from \( s \) to...
b of length at most j. Thus, \( \Pi(b) = \cup \Pi_j(b) \), and

\[
\text{MOP}(b) = \bigcap_{p \in \Pi(b)} f_p = \bigcap_{j} \bigcap_{p \in \Pi_j(b)} f_p
\]

By induction we show that the following holds for all \( j \):

\[
\text{MFP}(b) \subseteq \bigcap_{p \in \Pi_j(b)} f_p. \tag{2.1}
\]

For \( j = 1 \), then necessarily \( b = s \), and the rhs of \( \text{(2.1)} \) is \( f_s = \top \). Now suppose \( \text{(2.1)} \) is true for \( j = n \); we want to show it is true for \( j = n + 1 \). Notice that from the MFP algorithm (Fig. 2.2), \( \text{MFP}(b) = \bigcap_{x \in \text{pred}(b)} f_b(\text{MFP}(x)) \). The proof goes as follows:

\[
\text{MFP}(b) = \bigcap_{x \in \text{pred}(b)} f_b(\text{MFP}(x)) \leq \bigcap_{x \in \text{pred}(b)} f_b\left(\bigcap_{p \in \Pi_b(x)} f_p\right) \quad \text{Induction hypothesis}
\]
\[
\leq \bigcap_{x \in \text{pred}(b)} \bigcap_{p \in \Pi_b(x)} f_b \cdot f_p \quad \text{Monotonicity of } f_b
\]
\[
= \bigcap_{p \in \Pi_{b+1}(b)} f_p \quad \text{Distributivity of } f_b
\]

\[\square\]

**Theorem 2.3.2.** In an instance of a distributive data flow analysis framework satisfying the descending chain condition, \( \text{MFP} = \text{MOP} \).

**Proof.** In any case \( \text{MOP}(s) = \top = \text{MFP}(s) \). For \( b \neq s \), and with the same notation as in the previous proof, we have \( \text{MOP}(b) = \bigcap_{p \in \Pi(b)} f_p \). Now, given \( T(y) = \bigcap_{x \in \text{pred}(b)} f_b(y) \), we know that \( \text{MFP}(b) \) is the maximal solution for \( y \) such that \( T(y) = y \). In case of \( \text{MOP}(b) \),

\[
T(\text{MOP}(b)) = \bigcap_{x \in \text{pred}(b)} f_b(\text{MOP}(x)) \quad \text{Definition of } T
\]
\[
= \bigcap_{x \in \text{pred}(b)} f_b\left(\bigcap_{p \in \Pi_b(x)} f_p\right) \quad \text{Definition of } \text{MOP}
\]
\[
= \bigcap_{x \in \text{pred}(b)} \bigcap_{p \in \Pi_b(x)} f_b \cdot f_p \quad \text{Distributivity of } f_b
\]
\[
= \bigcap_{p \in \Pi_b(b)} f_p \quad \text{Definition of } \text{MOP}
\]

Therefore, \( \text{MOP}(b) \) is a fixpoint of \( T \). And since \( \text{MFP}(b) \) is the maximal fixpoint, \( \text{MOP}(b) \subseteq \text{MFP}(b) \), hence, using Theorem 2.3.1 we may conclude that \( \text{MOP}(b) = \text{MFP}(b) \). \[\square\]

### 2.3.2 Elimination-Based Data Flow Analysis

This thesis focuses on the elimination-based data flow analysis \([65]\). This approach to data flow analysis is defined algebraically. It has two algebraic operations, namely, substitution and loop-breaking that are performed on a set of flow equations. Substitution is defined as a transformation of the equation system, where an occurrence of a variable is replaced by its associated term. Loop-breaking eliminates self-referential equations with another equation that a solution to the modified system is also a solution to the original equation system.

**Definition 2.3.5. (Elimination DFA)** Let \( Q \) be a system of equations. Each equation \( Q_m \) is associated with a basic block \( m \) in the flow graph.

- \( s(Q, i, j) \) is the result of substituting the right-hand side of \( Q_i \) for a term \( Z_i \), on the right-hand side of equation \( Q_j \) such that \( m \neq j \).
• $b(Q, i)$ is the result of replacing the self-referential equation $Q_i$ with another equation $q_i$ such that every solution of $q_i$ is also a solution of $Q_i$ and $q_i$ itself is not self-referential.

The elimination-based approach can solve data flow problems that the iterative approach cannot \cite{67} and also naturally discovers recurrences through graph reduction. In Chapter 4 we exploit this fact. It is also worth noting that while the iterative approach compute the MFP solution, the elimination based approach computes the MOP solution \cite{67}, although in practice, some form of approximation is required when loops exist.

All elimination-based data flow analysis methods are akin to the Gaussian elimination approach to solving a system of equations. The Gaussian elimination approach is a general mathematical method for solving a system of linear equations. It is used widely in a wide range of scientific problems that go beyond program analysis. The Gaussian elimination method is the least efficient out of all the methods with $O(n^3)$ complexity, however, several very efficient variations exist, including, Sreedhar-Gao-Lee DJ graph bases analysis \cite{75} that we utilise in our implementation. Listing 2.1 presents an algorithm for a Gaussian elimination.

```
\// forward elimination
for(int i = 1; i <= n-1; i++)
{
    Q = b(i);
    for(int j = i + 1; j <= n; j++)
    {
        Q = s(Q, i, j);
    }
}

\// backward propagation
for(int i = n; i >= 2; i--)
{
    for(int j = i - 1; j >= 1; j--)
    {
        Q = s(Q, i, j);
    }
}
```

Listing 2.1: A C Implementation of the Gaussian Elimination Method

We can see as defined in Definition 2.3.5 there are two operations called in the algorithm of Listing 2.1. The algorithm is performed in two passes, namely, a forward elimination pass followed by a backward propagation pass. Loop-breaking is called only during the forward elimination and once backward propagation is performed, all loops are resolved. The elimination-based approach can be used on irreducible graphs, something that is of particularly important when performing an analysis on arbitrary program control flow topologies.
Chapter 3

Abstract Domains

The purpose of this chapter is to describe an abstract interpretation for range analysis. Chapter 1 introduced an example range analysis for an (incorrect) bubble sort program. In this chapter the mechanics of range analysis is described with respect to two unbounded domains, namely, discrete intervals and symbolic intervals. A canonical low-level language is specified for which a concrete and two abstract semantics are defined. We present the proofs of the Galois connection and soundness in Appendix B.

The process of constructing an abstract interpretation can be summarised in five steps:

1. Identify the analysis property.
2. Specify the syntactic algebra.
3. Specify the concrete domain and program semantics that captures the property.
4. Specify the computable abstract domain and program semantics that approximates the property.
5. Prove a precise connection between the two semantics (i.e. Galois connection) exists and show soundness.

In this chapter we will address each of the above steps with the aim of constructing an abstract interpretation.

3.1 Analysis Property: Variable Ranges

During the execution of a program, variables may assume several different values depending on the path of program execution. In static analysis all paths of a program are executed and as a result we are able to gather all the different values a variable may take on at a program point. A natural abstraction for such an analysis is a set of values. However, in the presence of loops, such a set is uncountable\(^1\) and inefficient to store sets that may contain all possible numerical values. A more feasible abstraction is an *interval* consisting of just the lower and upper bound of a variable. Two fundamental mechanisms of interval analysis are the merging and constraining of interval data. In interval analysis when several paths meet the intervals from these paths are merged together, this results in a *meet* operation on each of the variable definitions present in each of the paths. When a path is forked and constrained, for example, due to a conditional branching statement, a *join* operation

\(^{1}\)Theoretically speaking, practically the set size is limited by the maximum number representation.
is performed by applying the appropriate constraint to the constrained variable on each of the forked paths. In interval analysis performing a meet between two two intervals is defined as taking the lowest lower bound and largest upper bound of the two intervals, typically resulting in a wider interval. A join operation is defined as taking the highest lower bound and smallest upper bound, typically resulting in an interval narrowing.

The program in Listing 3.1 illustrates how intervals widen and narrow in a typical control flow.

```
1 \(\text{assume} \: a == [1, 4]\)
2 if (a < 3) // join \[1,4\] and \[minint, 2\], a is \[1,2\]
3 { 
4 \(a = 5;\)
5 }
6 \(\text{join} \[1,4\] and \[3, \text{maxint}\], a is \[3,4\]\)
7 \(\text{meet} \[5,5\] and \[3,4\], a is \[3,5\]\)
```

Listing 3.1: A simple range analysis

In this example, the variable \(a\) is assumed to have an interval value of \([1, 4]\) from previous operations. The if statement, at line 2, produces a branch in the control flow. One path assigns \(a\) the value of \([5, 5]\). The other path skips the assignment and a join between the inverse condition and the value of \([1, 4]\) is taken, resulting in the interval value of \([3, 4]\).

At line 6, both paths are merged. A meet is performed between the intervals \([3, 4]\) and \([5, 5]\), resulting in the possible values for \(a\) to widen to \([3, 5]\). Figure 3.1 shows this process through a data flow perspective.

Figure 3.1: Merging Information in a Flow Graph

The motivating example from Listing 1.1 in Chapter 1 contains several while loops. Loops pose two difficulties for range analysis. Firstly, fast termination becomes hard to attain as loops introduce
infinite paths. Secondly, although analysis techniques may produce sound approximations they may result in imprecise information that is not useful to the user, even though it is sound. Precision, is an important research challenge in program analysis, and can be improved through techniques [12, 35, 3].

The termination and precision difficulties are shown through the following trace of several iterations of the inner while loop of the example in Listing 1.1. In explaining the example, we informally define an environment to be a function from a set of variables to the set of intervals which again are simply pairs \([a, b]\) of integers such that \(a \leq b\). Now, when the environment maps a variable \(v\) to an interval \([a, b]\), we write \(v \mapsto [a, b]\). In the example, when the inner loop is first visited, we have that \(j \mapsto [0, 0]\) and \(k \mapsto [0, 1]\). In subsequent visits,

\[
\begin{align*}
j &\mapsto [0, 1] \quad \text{and} \quad k \mapsto [0, 2], \\
j &\mapsto [0, 3] \quad \text{and} \quad k \mapsto [0, 3], \\
j &\mapsto [0, 4] \quad \text{and} \quad k \mapsto [0, 4], \\
&\vdots \\
j &\mapsto [0, 4] \quad \text{and} \quad k \mapsto [0, \infty].
\end{align*}
\]

The example program, besides sorting an array, calculates how many swaps are performed in a single program execution. This is achieved through incrementing the variable \(k\) whenever a swap is performed. The range analysis, however, cannot simply iterate through the inner while loop as a fixpoint will only be reached after the interval widens to the maximum integer value. That is, the interval value assigned to \(k\) slowly continues to widen until it is \([0, \infty]\). With deeply nested loops and large word sizes could potentially result in unacceptably slow termination.

To mitigate the above problem, we employ the techniques of widening and narrowing. Widening is applied whenever we detect an interval widening of a variable value. In the above example, the value of \(j\) changes from \([0, 0]\) to \([0, 1]\) after one iteration of the inner loop. Here we widen the intervals into another interval \([0, +\infty]\) that encompasses both intervals. The new interval also has a property that it is a fixpoint: and executing the loop for another iteration would just result in the same interval. In this way we obtain a stable interval faster. However, \([0, +\infty]\) is extremely imprecise, allowing \(j\) to be any natural number, while in reality, the maximum value that it can take is five. To further mitigate the problem, we narrow the interval by executing another iteration with \(j\) having an initial value of \([0, \infty]\). The condition \(j < 5\) of the while loop restricts the interval \([0, +\infty]\) to \([0, 4]\) by a join operation, and the increment of \(j\) then enlarges the interval to \([0, 5]\), which is a stable interval, as confirmed by another iteration around the loop.

While widening and narrowing significantly improves the bounds of the variable \(j\), for \(k\), narrowing does not achieve any improvements and as a result its interval remains at \([0, +\infty]\). We therefore cannot determine that the swap is performed no more than 25 times. This scenario demonstrates the limitations of the basic interval domain and abstraction in general. Several domains exist that attempt to improve the bounds of intervals by incorporating the relationships between variables [3, 35].

### 3.1.1 Improving Bounds with Symbolic Intervals

Simple intervals have particularly imprecise bounds when external function calls or arguments are present in code. Function calls return the bottom element \((-\infty, \infty)\) to variables and all subsequent computations are usually over approximated.

In such situations, the symbolic domain can mitigate this problem and achieve more accurate bounds than the interval domain. A symbolic interval is defined as a pair of symbolic expressions rep-

---

2In this thesis we will use simple intervals and discrete intervals interchangeably.
resenting lower and upper bounds. A symbolic expression is an algebraic term consisting of symbols, algebraic functions and integers. Symbols represent any possible integer value and follow general arithmetic laws.

Symbolic intervals correspond to an inequality which generates a set of intervals in the discrete interval lattice. An immediate advantage symbolic intervals have over regular intervals is the interpretation of function calls. In the regular interval domain, assignments from a function call result in the return value being evaluated as the interval $[-\infty, \infty]$ denoting any possible values (bottom). The problem is that we cannot eliminate two infinite intervals because of absorption that may occur in previous computations. For example if a variable $i$ is $[-\infty, \infty]$, $i + k$ where $k$ denotes any number results in $[-\infty, \infty]$ again. It is therefore incorrect to allow the arithmetic rule that $[-\infty, \infty] - [-\infty, \infty] = 0$ because the addition of $k$ is ignored. The symbolic interval domain, however, assigns an interval symbols to its lower and upper bounds when it is assigned the return value of a function call. This becomes advantageous, because these symbols may be eliminated in future program statements. The program in Listing 3.2 demonstrates this advantage. The function call to $\text{foo}$ results in $x$ having an interval value of $[s_l, s_u]$. Here the discrete interval would simply assign $[-\infty, +\infty]$ to the variable. Both $a$ and $b$ have the same interval assigned to them and in the next computation $c$ is assigned the expression $b + 5$. In the last statement $b$ is eliminated, resulting in $d$ obtaining the value $[5, 5]$. The interval domain, on the other hand, would have to settle for the safe but over approximated value of $[-\infty, +\infty]$.

Another increase in precision with symbolic intervals is in the fact that they can in some circumstances exclude values within the bounds. For example, in Listing 3.3 the value for the variable even is $[2s, 2s]$. We can assert that the family of intervals that the symbolic generates is the set of all intervals with equal even numbers as the upper and lower bounds. Additionally, the set of odd intervals can be generated by the value of the variable odd by adding one to the even interval. Symbolic intervals can also generate family of intervals with constraints. For example the interval $[2k, k]$ generates intervals where both bounds are less than or equal to 0 (as intervals less than 0 have invalid intervals) and where the lower bound is always odd and two times less than the upper bound. The union of all these ranges would simple be the predicate all values less than or equal to 0. However, with the use of min and max this can be improved to not include values less than a given number and to ensure that the union of all generated intervals retains some numeric property. For example, $[\text{max}(0, 2k), 2k]$ generates intervals larger than or equal to 0 that consist of identical even numbers. If a variable contains this interval value we can safely assert that it must be an even number.

Symbolic intervals are much more powerful when relationships between variables are incorporated and recurrences are discovered in loops. However, this is not covered in this thesis and we refer the reader to [11, 3, 35, 71, 23, 69].
3.2 Low-Level Language Syntax

In this section we present the syntactical grammar of a low-level language that resembles most assembly and intermediate level languages such as the LLVM intermediate representation language. Figure 3.2 describes an abstract syntax for the language. The language is defined per basic block. As with most data flow level languages it has the property that branching instructions such as conditional and non-conditional gotos may only be placed last in the statement sequence of a basic block.

3.3 Concrete Semantics

The concrete semantics describes the precise mathematical meaning for programs written in our language. We first provide a denotational semantics of a basic block in our language, whose syntax is described in Figure 3.2. We then relate a program with a control-flow graph (see Definition 2.3.1), and provide the semantics operationally as a state transition system.
### Valuation Functions

![Valuation Functions](image)

#### Concrete Denotational Semantics of Basic Blocks

We define the syntactic domains of our programming language in Figure 3.3. We note that the lexical formation of identifiers (Id) and labels (Label) are left unspecified, as it varies from language to language, and we also assume that all identifier tokens are correct.

The denotational semantics of a basic block is given as a number of valuation functions. Some of them are applied on an environment \( \sigma \) representing the program’s execution state. An environment itself is given the following signature:

\[
\text{Env} = \text{Var} \rightarrow \mathbb{Z}.
\]

Several validation functions are defined in Figure 3.4 following the syntactic algebra in Figure 3.2. As control flow is defined operationally later as a transition rule, we exclude the Branch syntactical construct from the denotational semantics. Similarly, for the sake of simplicity, we omit the semantics of \( \phi \) operator for the concrete domain.

#### An Operational Semantics of CFG

The transition semantics of the program control flow describes transitions between basic blocks with guarded labels restricting the transition. The operational semantics of a simple imperative language have been described previously in [25, 44].

We can first define a syntax of a program as a sequence of \( l \) blocks as follows:

\[
\text{Program} ::= \text{Block}_1 \ldots \text{Block}_l
\]

We also assume that a program’s execution starts from \( \text{Block}_1 \). Similarly, we add a syntactic construct GenCond, which is a generalized form of a condition, together with its syntactic domain GenCond. The syntax of GenCond is defined as follows:

\[
\text{GenCond} ::= \text{Cond} | \neg \text{Cond}
\]
Now the corresponding valuation function is given as follows:

\[
C : \text{GenCond} \rightarrow \text{Env} \rightarrow \{0, 1\}
\]

\[
C[\text{Expr}_1 \text{ Pred} \text{ Expr}_2] = \lambda \sigma : \text{Env} \cdot \begin{cases} 
1 & \text{if } \text{Pred}(\mathcal{E}[\text{Expr}_1]_{\sigma}, \mathcal{E}[\text{Expr}_2]_{\sigma}) \\
0 & \text{otherwise}
\end{cases}
\]

\[
C[\neg(\text{Expr}_1 \text{ Pred} \text{ Expr}_2)] = \lambda \sigma : \text{Env} \cdot 1 - C[\text{Expr}_1 \text{ Pred} \text{ Expr}_2]_{\sigma}.
\]

For our purposes, we define a program to be a CFG (see Definition 2.3.1) with the signature \(G(V, E, s, \zeta, \xi)\). Compared to Definition 2.3.1 here the CFG has two additional components: the statement binding \(\zeta\) and the condition binding \(\xi\). The statement binding \(\zeta : V \rightarrow \text{StmtList}\) maps the vertices of the CFG to a sequence of statements, while \(\xi : E \rightarrow \text{GenCond}\) maps the edges of the CFG to generalized conditions. We say that a basic block \(\text{Block}_1 \in V\) is unconditionally connected to a basic block \(\text{Block}_2 \in V\) if \(\text{Block}_1\) is of the syntax \(\text{Label}_1 : \ldots; \text{goto} \text{ Label}_2\) and \(\text{Block}_2\) is of the syntax \(\text{Label}_2 : \ldots\). Given three basic blocks \(\text{Block}_1, \text{Block}_2, \text{Block}_3 \in V\) with \(\text{Block}_1\) having the syntax \(\text{Label}_1 : \ldots; \text{goto} \text{ Cond Label}_2\text{ Label}_3\), \(\text{Block}_2\) having the syntax \(\text{Label}_2 : \ldots\) and \(\text{Block}_3\) having the syntax \(\text{Label}_3 : \ldots\), we say that \(\text{Block}_1\) is connected to \(\text{Block}_2\) with \(\text{Cond}\) and that \(\text{Block}_1\) is connected to \(\text{Block}_3\) with \(\neg(\text{Cond})\). The CFG’s components can now be more precisely defined as follows:

\[
V = \{\text{Block}_i | \text{Block}_i \in \text{Program}\}
\]

\[
E = \{(\text{Block}_1, \text{Block}_2) | \text{Block}_1 \text{ is connected to } \text{Block}_2\}
\]

\[
\zeta = \{\text{Block} \mapsto \text{StmtList} | \text{Block} \in V \text{ and Block has syntax } \text{Label} : \text{StmtList}; \ldots\}
\]

\[
\xi = \{(\text{Block}_1, \text{Block}_2) \mapsto \text{Cond} | \text{Block}_1 \text{ is connected to } \text{Block}_2 \text{ with } \text{Cond}\}
\]

\[
\cup \{(\text{Block}_1, \text{Block}_2) \mapsto \neg(\text{Cond}) | \text{Block}_1 \text{ is connected to } \text{Block}_2 \text{ with } \neg(\text{Cond})\}
\]

\[
\cup \{(\text{Block}_1, \text{Block}_2) \mapsto \top | \text{Block}_1 \text{ is unconditionally connected to } \text{Block}_2\}
\]

We now define the semantics of the program as composition of state transition steps. We formalise the notion of the program’s state as a pair \(\langle \text{Block}, \sigma \rangle\) for \(\text{Block}\) denoting the basic block to be executed, and \(\sigma\) an environment. We write \(\langle \text{Block}, \sigma \rangle \parallel \langle \text{Block}', \sigma' \rangle\) to represent the execution of basic block \(\text{Block}\) resulting in a state where \(\text{Block}'\) is ready to be executed and the environment \(\sigma\) is updated to \(\sigma'\). We formalise the transition step in the concrete semantics as follows:

\[
e = (\text{Block}, \text{Block}') \in E \land \sigma' = S^* \mathcal{E}[\text{Block}](\sigma) \land C[\mathcal{E}[\text{Block}]](\sigma') = 1
\]

\[
\langle \text{Block}, \sigma \rangle \parallel \langle \text{Block}', \sigma' \rangle
\]

The rule states that a transition can only occur if the guard associated with an edge evaluates to 1. Effectively this describes that a concrete transition only takes the states corresponding to the elements of the ideal set.

### 3.4 Abstract Semantics: Intervals

The purpose of this section is to describe the domain structure and abstract interval semantics for the data flow level language in Figure 3.2. We reuse several definitions from the concrete semantics and present semantic aspects that are unique.

#### 3.4.1 Interval Abstract Semantics of a Basic Block

The abstract denotational semantics of a basic block is given as a number of abstract valuation functions in Figure 3.8.
In the interval abstract semantics, we replace integers $\mathbb{Z}$ with integer intervals $I(\mathbb{Z})$. Integer intervals are defined algebraically as a lattice structure denoted as:

$$I(\mathbb{Z}) = \langle [\mathbb{Z}, \mathbb{Z}], \sqsubseteq, \sqcup, \sqcap \rangle$$

The base set $([\mathbb{Z}, \mathbb{Z}])$ is a pair of integers such that the lower bound is smaller or equal to the upper bound.

$$[\mathbb{Z}, \mathbb{Z}] = \{ [l, u] \mid l \in \mathbb{Z} \cup \{\infty\} \land u \in \mathbb{Z} \cup \{\infty\} \land l \leq u \} \sqcup \{\top\}$$

Intervals are ordered by inclusion. The partial ordering $\sqsubseteq$ satisfies the condition

$$[a, b] \sqsubseteq [c, d] \text{ holds if } a \leq c \land d \leq b$$

The binary meet ($\sqcap$) and join ($\sqcup$) operators are defined in Figure 3.6. In this definition $\sqcap$ merges information resulting in a loss of precision and $\sqcup$ constrains information resulting in more accurate information.

Arithmetic operations on intervals follow standard interval arithmetic rules [53] summarised in Figure 3.7.

In the abstract semantics two mappings are defined. An environment defined as

$$\langle \text{Env}^\#, \sqsubseteq_{\text{Env}^\#}, \sqcup_{\text{Env}^\#}, \sqcap_{\text{Env}^\#} \rangle$$

where

$$\text{Env}^\# = \text{Id} \rightarrow I(\mathbb{Z})$$

And a transition mapping, which maps a label to an environment

$$\text{Trans}^\# = \text{Id} \rightarrow \text{Env}^\#$$
\[ [a, b] \cap [c, d] = [\min(a, c), \max(b, d)] \]
\forall I \in I(\mathbb{Z}) I \cap \flat I = I
\forall I \in I(\mathbb{Z}) \cap \flat I \cap \flat I = I
\forall I \in I(\mathbb{Z}) \cap \flat I \cap \flat I = [-\infty, \infty]
\forall I \in I(\mathbb{Z}) \cap \flat I \cap \flat I = [\min(a, c), \max(b, d)]
\forall I \in I(\mathbb{Z}) I \cup \flat \top \flat I = I
\forall I \in I(\mathbb{Z}) I \cup \flat \top \flat I = I
\forall I \in I(\mathbb{Z}) I \cup \flat \top \flat I = I
\forall I \in I(\mathbb{Z}) I \cup \flat \top \flat I = I

Figure 3.6: Meet and Join Operators

\[ +_\sharp : I(\mathbb{Z}) \times I(\mathbb{Z}) \to I(\mathbb{Z}) \]
\[ \lambda[a, b]. \lambda[c, d]. [a + c, b + d] \]
\[ -_\sharp : I(\mathbb{Z}) \times I(\mathbb{Z}) \to I(\mathbb{Z}) \]
\[ \lambda[a, b]. \lambda[c, d]. [a - d, b - c] \]
\[ *_\sharp : I(\mathbb{Z}) \times I(\mathbb{Z}) \to I(\mathbb{Z}) \]
\[ \lambda[a, b]. \lambda[c, d]. [\min(a * c, a * d, b * c, b * d), \max(a * c, a * d, b * c, b * d)] \]
\[ \div_\# : I(\mathbb{Z}) \times I(\mathbb{Z}) \to I(\mathbb{Z}) \]
\[ \lambda[a, b]. \lambda[c, d]. [\min(a \div c, a \div d, b \div c, b \div d), \max(a \div c, a \div d, b \div c, b \div d)] \]
\[ +_\# : I(\mathbb{Z}) \to I(\mathbb{Z}) \]
\[ \lambda[a, b]. [a, b] \]
\[ -_\# : I(\mathbb{Z}) \to I(\mathbb{Z}) \]
\[ \lambda[a, b]. [-b, -a] \]

Figure 3.7: Interval Arithmetic Functions

3.4.2 **Abstract Operational Semantics of CFG**

We extend the definition of the concrete CFG operational semantics to include condition constraints. Additionally, two functions are defined, that widen lower or upper bounds for an interval.

- **lwr : I(\mathbb{Z}) \to I(\mathbb{Z})**
  \[ \lambda[a, b]. [-\infty, b] \]

- **upr : I(\mathbb{Z}) \to I(\mathbb{Z})**
  \[ \lambda[a, b]. [a, \infty] \]

These functions are used constrain interval values.
We formalize the transition step in the interval abstract semantics as follows:

\[ C^d : \text{GenCond} \rightarrow \text{Env}^d \rightarrow \text{Env}^d \]

\[ C^d[\text{Expr}_1 < \text{Expr}_2] = \]

\[ \lambda \sigma : \text{Env}^d.\mathcal{A}^d[\text{Expr}_1] \sigma(lwr(\mathcal{E}^d((\text{Expr}_2)\|\sigma)) - [1, 1]) \cup (\mathcal{A}^d[\text{Expr}_2] \sigma(upr(\mathcal{E}^d((\text{Expr}_1)\|\sigma)) + [1, 1]) \]

\[ C^d[\text{Expr}_1 > \text{Expr}_2] = \]

\[ \lambda \sigma : \text{Env}^d.\mathcal{A}^d[\text{Expr}_1] \sigma(lwr(\mathcal{E}^d((\text{Expr}_2)\|\sigma)) + [1, 1]) \cup (\mathcal{A}^d[\text{Expr}_2] \sigma(upr(\mathcal{E}^d((\text{Expr}_1)\|\sigma)) - [1, 1]) \]

\[ C^d[\text{Expr}_1 \leq \text{Expr}_2] = \]

\[ \lambda \sigma : \text{Env}^d.\mathcal{A}^d[\text{Expr}_1] \sigma(lwr(\mathcal{E}^d((\text{Expr}_2)\|\sigma))) \cup (\mathcal{A}^d[\text{Expr}_2] \sigma(upr(\mathcal{E}^d((\text{Expr}_1)\|\sigma)) \]

\[ C^d[\text{Expr}_1 \geq \text{Expr}_2] = \]

\[ \lambda \sigma : \text{Env}^d.\mathcal{A}^d[\text{Expr}_1] \sigma(lwr(\mathcal{E}^d((\text{Expr}_2)\|\sigma))) \cup (\mathcal{A}^d[\text{Expr}_2] \sigma(upr(\mathcal{E}^d((\text{Expr}_1)\|\sigma)) \]

\[ C^d[\text{Expr}_1 \neq \text{Expr}_2] = \]

\[ \lambda \sigma : \text{Env}^d.\mathcal{A}^d[\text{Expr}_1] \sigma(lwr(\mathcal{E}^d((\text{Expr}_2)\|\sigma))) \cup (\mathcal{A}^d[\text{Expr}_2] \sigma(upr(\mathcal{E}^d((\text{Expr}_1)\|\sigma)) \]

\[ C^d[\text{Expr}_1 \leq \text{Expr}_2] \cup C^d[\text{Expr}_1 \geq \text{Expr}_2] \]

We define the validation function \( \mathcal{A}^d \) as:

\[ \mathcal{A}^d : \text{Expr}^d \rightarrow \text{I}(\mathbb{Z}) \rightarrow \text{Env}^d \rightarrow \text{Env}^d \]

\[ \mathcal{A}^d[\text{BinOp}\text{Expr}_1\text{Expr}_2] = \lambda i : \text{I}(\mathbb{Z}).\lambda \sigma : \text{Env}^d.\mathcal{A}^d[i\text{Expr}_1\text{BinOp}\text{Expr}_2](\text{BinOp}(i)(\mathcal{E}^d((\text{Expr}_2)\|\sigma))\sigma \]

\[ \mathcal{A}^d[\text{UnOpExpr}] = \lambda i : \text{I}(\mathbb{Z}).\lambda \sigma : \text{Env}^d.\mathcal{A}^d[\text{UnopExpr}](\text{Expr}(i))\sigma \]

\[ \mathcal{A}^d[\text{id}] = \lambda i : \text{I}(\mathbb{Z}).\lambda \sigma : \text{Env}^d.[\text{id} \mapsto \sigma(\text{id}) \cup \{i\}]\sigma \]

\[ \mathcal{A}^d[\text{Const}] = \lambda i : \text{I}(\mathbb{Z}).\lambda \sigma : \text{Env}^d.\top \]

We formalize the transition step in the interval abstract semantics as follows:

\[
\begin{align*}
  e = (\text{Block}, \text{Block'}) \in E & \land \sigma' = S^d[\zeta(\text{Block})]\|\sigma \land C^d[\xi(e)\|\sigma] = \sigma' \land \text{Label}_{\text{Block}}' \mapsto \sigma' \in \rho' \\
  (\text{Block}, \rho) & \Downarrow (\text{Block'}, \rho')
\end{align*}
\]

This rule states that given an expression a transition mapping is produced. Given a basic block a state is produced such that the successor label is mapped to the state in the transition mapping. Then the next basic block will contain an environment associated with its label in the transition mapping.

### 3.5 Abstract Semantics: Symbolic Interval Domain

The purpose of this section is to describe the domain structure and semantics of the symbolic interval abstraction. The notion of a symbolic interval domain is described, including the lattice structure and operations on the domain. The semantic functions are then described denotationally.

#### 3.5.1 Interval Abstract Semantics of a Basic Block

The symbolic intervals denotational semantics of a basic block is defined in Figure 3.8. The semantics are largely the same as for basic intervals aside from the notable difference of a symbolic interval domain and a functional call generates a new symbolic interval with the `newsymbol()` function.
Definition 3.5.1. (Symbolic Domain)

\[ \mathcal{I}(\mathbb{Z})^\dagger = \langle Expr^\dagger \times Expr^\dagger, \sqsubset^\dagger, \sqcup^\dagger, \sqcap^\dagger \rangle \]

Because of the symbolic interval domain generates a set of intervals we call the symbolic interval domain a *generating* domain. The function \( \varrho \) expresses the generating nature of the symbolic interval domain. We let \( \varrho \) be a function from \( \mathcal{I}(\mathbb{Z})^\dagger \to Set_\mathcal{I}(\mathbb{Z}) \)

Definition 3.5.2. (Generating Function)

\[ \varrho([l, u]) = \begin{cases} 
  l \in \text{Symbol} \land u \in \mathbb{Z} \rightarrow \{(a, b) \in \mathcal{I}(\mathbb{Z}) : -\infty \leq a \land b = u\} \\
  l \in \mathbb{Z} \land u \in \text{Symbol} \rightarrow \{(a, b) \in \mathcal{I}(\mathbb{Z}) : a = l \land b \leq \infty\} \\
  l \in \text{Symbol} \land u \in \text{Symbol} \rightarrow \{(a, b) \in \mathcal{I}(\mathbb{Z}) : -\infty \leq a \land b \leq \infty\} \\
  l \in \mathbb{Z} \land u \in \mathbb{Z} \rightarrow [l, u] 
\end{cases} \]

We define the symbolic interval domain in terms of the \( \varrho \) function. The base set is defined as a pair of symbolic expressions.

\[ \langle Expr^\dagger, Expr^\dagger \rangle \]

We define a partial order in terms of inclusion of the generated sets:

\[ \forall i_a, i_b \in \mathcal{I}(\mathbb{Z})^\dagger \cdot \text{if } i_a \sqsubseteq^\dagger i_b \leftrightarrow \varrho(i_a) \subseteq \varrho(i_b) \]

The meet(\( \sqcap^\dagger \)) operator is defined as the union of the generated sets:
A symbolic expression $Expr^\dagger$ is defined inductively, consisting of integers, variables or an operation on expressions. Figure 3.9 gives the BNF syntax of a symbolic expression.

A symbolic expressions itself forms a term algebra, obeying the traditional arithmetic laws. Additionally we extend this arithmetic with min and max expressions. For expressions to be compared they must be reduced to a canonical form. An expression is in normal form if it cannot be reduced any further. We denote set of normalised expressions as $Expr_N$. We define the normal form expressions with the syntax in Figure 3.10.

Symbolic expressions obey the axioms of general arithmetic algebra. In this thesis we do not preset the standard algebraic rules, as we assume this is intuitive to the reader. However, symbolic expressions contain two non standard algebraic operations, namely, min and max. For our min and max functions we introduce additional rewrite equations. We define $Reduce$ to be rewrite function that rewrites according to the following rules:

\[
\forall i_a \in I(\mathbb{Z})^\dagger. \forall i_b \in I(\mathbb{Z})^\dagger \cap^\dagger \ i_b = g(i_a) \cup g(i_b)
\]

and algebraically as:

\[
\forall [l_a, u_a] \in I(\mathbb{Z})^\dagger. \forall [l_b, u_b] \in I(\mathbb{Z})^\dagger [l_a, u_a] \cap^\dagger [l_b, u_b] = [\min(g(l_a), g(l_b)), \max(g(u_a), g(u_b))]
\]

The join($\sqcup$) operator is defined as the intersection of the generated sets:

\[
\forall i_a \in I(\mathbb{Z})^\dagger. \forall i_b \in I(\mathbb{Z})^\dagger \cup^\dagger \ i_b = g(i_a) \cap g(i_b)
\]

and algebraically as:

\[
\forall [l_a, u_a] \in I(\mathbb{Z})^\dagger. \forall [l_b, u_b] \in I(\mathbb{Z})^\dagger [l_a, u_a] \cup^\dagger [l_b, u_b] = [\max(g(l_a), g(l_b)), \min(g(u_a), g(u_b))]
\]
\[
\text{Reduce}(e) = \begin{cases}
\min(-\infty, x) \rightarrow -\infty \\
\min(\infty, x) \rightarrow x \\
\min(x, x + \text{num}) \rightarrow x \\
\min(x, x - \text{num}) \rightarrow x - \text{num} \\
\min(x, x) \rightarrow x \\
\min(x, -x) \rightarrow -x \\
\min(x, x * (+\text{num})) \rightarrow x \\
\min(x, x \div (+\text{num})) \rightarrow x \div (+\text{num}) \\
\min(x, x * (-\text{num})) \rightarrow x * (-\text{num}) \\
\min(x, x \div (-\text{num})) \rightarrow x \\
\min(\min(x, y), \max(x, y)) \rightarrow \min(x, y) \\
a \star \min(x, y) \rightarrow \min(x \star a, y \star a)
\end{cases}
\]

where \(\star\) is a binary operation +, *, -, \(\div\)

Normalised expressions therefore are defined in terms of the Reduce function as follows:

\[
\text{Expr}_N\text{ where } \forall e \in \text{Expr}_N. \text{Reduce}(e) = e
\]

Finally, we define a normalised symbolic interval is a pair of normalised expressions.

**Definition 3.5.3. (Normalised Symbolic Interval)**

\[
\langle \text{Expr}_N, \text{Expr}_N \rangle
\]

When equating or operating on symbolic intervals we must ensure they are in an initial (normalised) form. Otherwise, symbolically, two intervals may not be regarded as equal as they are differentiated by their symbolically unreduced forms containing extra symbolic information\(^3\). Theoretically, by normalising the expressions we ensure that we operate only on terms from the initial algebra and hence guarantee that two normalised forms that are symbolically different cannot semantically be the same.

\(^3\)Fully reduced terms are said to adhere to the “no confusion and no junk property”.
Chapter 4

Abstract Interpretation of Unstructured Languages

This chapter of the thesis describes a proposed solution to perform range analysis on unstructured imperative languages. Unstructured languages typically do not have structural composition consisting of statements such as `while` and `if`, rather, they have `goto`-like statements which do not restrict the topology of the program control flow graph. We first describe the limitations of current analysis techniques that use widening and narrowing and then present our solution that combines widening and narrowing with the algebraic elimination-based data flow analysis technique. We prove the correctness of our approach by proving that all produced solutions for a system of equations always lies within the solution space.

4.1 Finding Fixpoints with Existing Techniques

The existing widening and narrowing techniques are described in Chapter 2 on Page 16. Loops in control flow ultimately result in recursive data flow equations. Widening/narrowing is a technique for resolving such recursive data flow equations iteratively by attempting to find an approximate solution.

For the interval domain, widening and narrowing is defined in the following way:

**Definition 4.1.1. (Interval Widening)**

\[
\top \vee X = X \\
X \vee \top = X \\
[l_l, u_l] \vee [l_u, u_u] = \text{if } l_u < l_l \text{ then } -\infty \text{ else } l_l, \text{if } u_u > u_l \text{ then } +\infty \text{ else } u_l
\]

and narrowing is defined in the following way:

**Definition 4.1.2. (Interval Narrowing)**

\[
\top \triangle X = \top \\
X \triangle \top = \top \\
[l_l, u_l] \triangle [l_u, u_u] = \text{if } l_l < l_u \text{ then } l_l \text{ else } l_u, \text{if } u_l > u_u \text{ then } u_l \text{ else } u_u
\]

In Figures 4.1 and 4.2 we present two representations of a simple while loop program. Cousot and Cousot’s widening and narrowing approach [12] can resolve the while loop recurrence function \(F(X)\) iteratively. Let \(F(X) = ([1, 1] \cap X + [1, 1]) \cup [-\infty, 100]\) where \(F(X)\) is a function within a loop operation. Let \(X^i_w\) denote a variable \(X\) widened at iteration \(i\) and let \(X^i_n\) denote a variable \(X\) narrowed at iteration \(i\). The widening and narrowing process can be summarised as follows:
The idea is to now narrow by decreasing iterations:

\[
\begin{align*}
  X^0_w &= \top \\
  X^1_w &= X^0_w \lor ([1, 1] \cap (X^0_w + [1, 1])) \sqcup [-\infty, 100] \\
  &= \top \lor ([1, 1] \cap (X^0_w + [1, 1])) \sqcup [-\infty, 100] \\
  &= ([1, 1] \cap \top) \sqcup [-\infty, 100] \\
  &= [1, 1] \\
  X^2_w &= X^1_w \lor ([1, 1] \cap (X^1_w + [1, 1])) \sqcup [-\infty, 100] \\
  &= [1, 1] \lor ([1, 1] \cap ([1, 1] + [1, 1])) \sqcup [-\infty, 100] \\
  &= [1, 1] \lor ([1, 1] \cap [2, 2]) \sqcup [-\infty, 100] \\
  &= [1, 1] \lor ([1, 2]) \sqcup [-\infty, 100] \\
  &= [1, 1] \lor [1, 2] \\
  &= [1, \infty]
\end{align*}
\]

However, the same program written in the language in Figure 3.2 of Chapter 3 presents difficulties for this method. The language is unstructured and does not contain a specific loop operation such as `while` or `for`. Instead, this language relies on a `goto` statement to form loops and branches. Statements such as `goto`, however, do not constrain the program topology and cannot guarantee loops are eventually formed. When performing the traditional abstract interpretation on unstructured languages and unbounded domains the widening and narrowing approaches cannot accelerate termination as it is impossible to know where widening and narrowing should be applied. Unlike structured languages the syntax and semantics of unstructured languages do not explicitly identify loops in a program.
These problems are shown in the semantic function below. Here we have a semantic function of structured language, which forms loops through a while loop construct, denotationally be defined as:

$$S[\text{while } c \text{ do } y] = \lambda \sigma. \, \text{Env} \, \text{if } E[c] \sigma = 1 \text{ then } S[y] \sigma \text{ else } \text{skip}.$$ 

The while construct defines a cycle in the corresponding program graph. All semantic sub-components are guaranteed to be inside the cycle. In such definitions, widening and narrowing can be applied inductively to the statements denoted by $y$.

The unstructured imperative language in Figure 3.2 allows any arbitrary topological structures to be formed with the goto statement. Its semantic function is defined as follows:

$$\mathcal{J}^{\sharp}[\text{goto Cond Label}_1 \text{ Label}_2] = \lambda \sigma : \text{Env}^{\sharp}.\{\text{Label}_1 \mapsto (\mathcal{A}^{\sharp}[\text{Cond}]\sigma)\}(\text{Label}_2 \mapsto (\mathcal{A}^{\sharp}[\neg \text{Cond}]\sigma))\text{newtrans()}$$

In the goto semantic function there is no explicit semantic or syntactic element that can guarantee that a loop will be formed. As a result widening and narrowing cannot be applied to sub-statements or successor statements. For example, the target label may not be in a loop. Applying widening and narrowing to any increase in an interval value is inefficient and may further diminish the precision of the entire analysis. Further, if a loop is formed it may be of any complexity and topology, further creating difficulties for applying widening and narrowing in ad hoc ways.

In light of the above example, we can see that when performing an analysis on unstructured languages, abstract interpretation is in no way superior to data flow analysis as indicated in [12]. Without widening and narrowing, abstract interpretation on unstructured iterative language is no more than a theoretical formalisation of iterative data flow analysis.

### 4.2 Abstract Interpretation in an Elimination Framework

The elimination-based data flow technique, described in Chapter 2, is an algebraic data flow method that instead of iterating through the CFG, reduces the equation system to an initial form before finding a solution. Our approach utilises the algebraic properties of the elimination data flow method and delegates the algebraic requirements of structured languages to the analysis technique itself. We no longer rely on language operations such as $\text{if}$ and $\text{while}$, instead, we place the algebraic requirement on the static computation, namely, the data flow analysis technique. This involves embedding widening/narrowing within the loop-breaking operation. Using our solution an abstract interpretation can be performed on unstructured languages regardless of the domain bounds of the analysis.

The elimination-based data flow technique reduces the equation system by performing a sequence of algebraic operations, namely, substitution and loop-breaking.

**Definition 4.2.1. (Elimination Framework)** We define the elimination framework as a free algebra \( \langle \text{Eqs}, \text{sub}, \text{lb} \rangle \), where the carrier set \( \text{Eqs} \) is the set of data flow equations and the operations substitution (sub) and loop-breaking (lb).

The algebraic operations \( \text{sub} \) and \( \text{lb} \) transform the equation system to a initial form (normalised form). We denote \( \text{Eqs}' \) as the equation system resulting from an operation on the equation system \( \text{Eqs} \). Each operation has an associated pre-condition that specifies if it can alter the equation system. The operation \( \text{sub} \) requires that the substituted term has a corresponding variable on the right-hand-side of the equation being acted upon. Similarly, \( \text{lb} \) requires that the equation acted upon is a recurrence relation (recursively defined). The order in which these operations are called largely depends on the specific algorithm, in the theoretical section of this thesis we refer to the Gaussian algorithm of Chapter 2, however as mentioned most algebra-based data flow analysis are variations of Gaussian elimination.
Definition 4.2.2. \((EQS')\)  
\[EQS' = \begin{cases} 
\text{sub}(EQS, i, j) & \text{if } i \neq j \text{ and } X_i \in EQS \text{ and } X_j \in \text{args}_i \\
\text{lb}(EQS, i) & \text{if } X_i \in EQS \text{ and } X_i \in \text{args}_i 
\end{cases}\]

An equation is defined as a left-hand-side variable and a right-hand-side function which constrains a set of terms as arguments. These terms can be variables or terms defined by functions.

We define a system of equations as the set of equations that have a general form defined in Definition 4.2.3.

Definition 4.2.3. \((Equation System)\) We define an equation system EQS over a lattice \(\mathcal{L}\) with a function space \(\mathcal{F}_s\) as \(x_i \sqsubseteq f_i(\bigcap_{k \in \text{pred}(i)} X_k)\) where \(f_i \in \mathcal{F}_s\) and \(X_i\) are variables whose values are in \(\mathcal{L}\).

We define the concept that every variable is mapped to a solution that is in the analysis domain (in our case the lattice of variable to real functions).

Definition 4.2.4. \((Solution)\) We define a solution in the EQS as an assignment of variables \(\psi : X \rightarrow \mathcal{L}\), i.e, \(\text{sol} = \{(x_1, \mathcal{L}_1), (x_2, \mathcal{L}_2), ..., (X_n, \mathcal{L}_n)\}\).

By reducing the equation system through \(\text{sub}\) and \(\text{lb}\) operations we obtain a solution. However, we are interested in the admissible solution, that is, the solution where all variables are eliminated and replaced by corresponding lattice values. When we have a admissible solution we can stop performing the reductions using the algebraic operations and can solve the system of equations.

Definition 4.2.5. \((Admissible Solution)\) An admissible solution is a solution that all equations hold by replacing all occurrences of \(X_i\) by \(\mathcal{L}_i\) \(\psi(X_i) \sqsubseteq f_i(\bigcap_{k \in \text{pred}(i)} \psi(X_k))\).

Further, we say that an admissible solution space is the set of all possible admissible solutions.

Definition 4.2.6. \((Admissible Solution Space)\) An admissible solution space \(\text{SOL}(EQS)\) is the set of all admissible solutions.

We can now formally define the algebraic operations. The substitution operation \(\text{sub}(i, j)\) is defined as a transformation of the equation system. It substitutes the right-hand-side of an equation indexed by \(i\) for a matching variable in the right-hand-side of the equation indexed by \(j\).

Definition 4.2.7. \((Substitution Operation)\)
\[\text{sub}(EQS, i, j) = \begin{cases} 
X_i = f_i(\bigcap_{k \in \text{pred}(i)} X_k)(f_j/X_j), & \text{if } i \neq j \text{ and } X_i \in EQS \\
X_i = f_i(\bigcap_{k \in \text{pred}(i)} X_k), & \text{otherwise}
\end{cases}\]

In Definition 4.2.7 we define \(i\) an index to an equation in the EQS that will be acted upon and \(j\) an index to an equation in the EQS that will be substituted into \(i\). The substitution operation syntactically substitutes all occurrences of \(X_j\) for \(f_j\) given that \(i \neq j\). We show that substitution can produce the initial or normal form of an equation.

Definition 4.2.8. \((Equation Normal Form)\) We describe the substitution as follows:
- By commutativity of meet we can describe \(X_i\) as \(\text{sub}(EQS, i, j) \leftrightarrow X_i \sqsubseteq f_i(\bigcap_{k \in \text{pred}(i)} X_k \cap X_j)\)
- If \(x_j\) is in \(\text{pred}(i)\)
  - By substitution \(\sqsubseteq f_i(\bigcap_{k \in \text{pred}(i)} X_k \cap f_j(\bigcap_{k \in \text{pred}(i)} X_k))\)
  - And once normalised we have \(\sqsubseteq f'_i(\bigcap_{k \in \text{pred}(i)} X_k)\).
We define the loop-breaking operation \( lb(i) \) as substituting a function \( F^* \) for the recurring variable at the right-hand-side of the recurrence equation. \( F^* \) will replace the reoccurring variable with a value that is also a solution. \( F^* \) is essentially widening and narrowing. We first substitute the bottom element \( (\mathbb{R}, \mathbb{R}) \) for all recurring variables in the equation and then calculating a solution that is re-inserted for the recurring variable. This process of substituting solutions is repeated until a fixpoint is reached.

**Definition 4.2.9. (Loop-Break Operation)** Let \( X_i \subseteq f'_i(\mathbb{R}) \) and \( X_i \in EQS \)

\[
 lb(EQS, i) = F^*(f'_i(\mathbb{R}), X_i) \quad \text{and} \quad F^* = \text{narrow}^+(\text{widen}(EQS, i), i) \quad \text{where} \quad + \quad \text{is an application of} \quad \text{narrowing one or more times until there is no more change}
\]

### 4.2.1 Elimination

The elimination framework will perform a sequence of transformations, with the sub and \( lb \) operations. Ultimately, any cyclic graph will result in a loop-breaking operation being called, regardless of the reducibility of the graph. In this example we follow the Gaussian elimination approach from Chapter 2. To illustrate the technique, suppose we have a CFG representation of an irreducible CFG. This situation can only be caused by a unstructured goto loop program. Suppose we start with the following data flow equations as shown in Figures 4.3 and 4.4. A sequence of sub and \( lb \) operations are performed eliminating all variables from the right-hand-sides of equations.
EQS = \[
\begin{align*}
X_0 &= f_0(\tau) \\
X_1 &= f_1(X_0, X_2) \\
X_2 &= f_2(X_0, X_1)
\end{align*}
\] =>

EQS_0 = \[
\begin{align*}
X_0 &= f_0(\tau) \\
X_1 &= f_1(f_0(\tau), X_2) \quad \text{result of } sub(0, \forall j \mid i < j \leq n) \\
X_2 &= f_2(f_0(\tau), X_1)
\end{align*}
\]

EQS_1 = \[
\begin{align*}
X_0 &= f_0(\tau) \\
X_1 &= f_1(f_0(\tau), X_2) \quad \text{result of } sub(1, \forall j \mid 0 < j \leq n) \\
X_2 &= f_2(f_0(\tau), f_1(f_0(\tau), X_2))
\end{align*}
\]

EQS_2 = \[
\begin{align*}
X_0 &= f_0(\tau) \\
X_1 &= F^+(f_1(f_0(\tau), f_2(f_0(\tau), f_1(f_0(\tau), X_2), X_2'))) \quad \text{result of } lb(2), sub(2, \forall j \mid 2 > j \leq 0) \\
X_2 &= F^+(f_2(f_0(\tau), f_1(f_0(\tau), X_2), X_2'))
\end{align*}
\]

We denote the result of a loop-breaking operation as $F^+(f_i(\ldots X_i, \ldots), X'_i)$ where $X'_i$ is the fixpoint value that shall replace $X_i$. Now that the equation is reduced, we solve the system of equations. All $f_i$ functions reflect the standard data flow equation functions. The $F^+$ function is the widening and narrowing function. We show how the fixpoint is calculated on the converging equation in Figure 4.3 and the diverging equation in Figure 4.4.

4.2.2 Solving

Figures 4.5 and 4.6 show tree function decompositions of the post-elimination $X_1$ equation from the examples in Figures 4.3 and 4.4. After solving an equation, the left-hand-side variable represents the program environment at its respective point in the program.

An equation is solved by a standard eager evaluation of the right-hand-side function. That is, the evaluation is bottom up. In Figure 4.5 the converging case from Figure 4.3 is depicted. Evaluation of the fixpoint function deserves special attention. Its evaluation will result in several iterations until a fixpoint solution is found. In the example in Figure 4.5 a first iteration calculates the value of $f_2$ with the variable $X_2$ mapped to $\top$. This results in the value $[0, 0]$. The resultant value is then substituted into $X_2$ and again the resultant value of $f_2$ is $[0, 0]$. As we have reached a fixpoint, the value of $F^+$ is $[0, 0]$ and the computation results in $f_1$ having a value of $[1, 1]$.

In the divergent case in Figure 4.6 more iterations are needed. The first iteration results in $f_2$ having post calculation value of $[2, 2]$. In the next iteration however we see that the value is $[2, 4]$. By definition of widening, if a value begins to diverge, we replace it with an approximation of a fixpoint value, namely, $[2, \infty]$. The next iteration tries to improve on the fixpoint value of $[2, \infty]$. This however, for this program, is not possible and we conclude that $f_2$ diverges, as does $f_1$.

By looking at the programs and the analysis result it is easy to see that these results are accurate. In the divergent case the variable $i$ is incremented and decremented in the infinite loop, maintaining a stable value for each basic block. In the second example, because both functions $f_1$ and $f_2$ increment, it causes a divergent value in the infinite loop.
In Figure 4.7 we solve the example from Figure 4.2. We can see that in this example we can perform a narrowing after a widening.

We can see that the result from Cousot and Cousot’s widening and narrowing is replicated, however, without the need for a while statement.

### 4.3 Correctness of Framework

In this section of the thesis we prove the correctness of the presented solution. We define correctness as maintaining the invariant that every equation transformation is also in the solution space.

Formally we say that the invariant that must be maintained is that the modified equation system defined in Definition 4.2.2 is an approximation of the previous equation system.

**Definition 4.3.1. (Correctness Invariant)**

\[ SOL(EQS') \subseteq SOL(EQS) \]

First we prove that bottom is always solution, and then define widening and narrowing in terms of solution spaces.

**Observation 4.3.1. (Bottom Solution)** A solution of bottom \( (\psi_\bot) \) is in the solution space.

\[ \{ X_i \mapsto \_ \_ \_ : i \in \{1..n\} \} \in SOL(EQS) \]

where we define \( \_ \_ \_ \) a lattice where all recurring variables are \( \bot \)

**Proof.** Trivial (By definition \( \bot \) approximates all values in the domain) \( \Box \)

**Definition 4.3.2. (Widening in Framework)** We say that \( \psi_\bot \) is the solution to any function \( f \). By widening, we mean that we have approximated the solution to a function as \( \psi_\bot \)
First iteration

widening iteration

narrowing iteration 1

narrowing iteration 2

Transfer Functions

\[ f_0 = x := 1 \]
\[ f_1 = x <= 10 \]
\[ f_2 = x := x + 1 \]
\[ f_3 = ....... \]

Figure 4.7: Widening and Narrowing on a While Program
Definition 4.3.3. (Narrowing in Framework) We define the solution to narrowing as the solution obtained by substitution \( \psi_\perp \) for some \( X_j \)
\[
\psi' = f(\bigcap_{k \in \text{pred}(i)} X_k \cap \psi_\perp)
\]

Next we proceed to prove that the algebraic substitution operation of the elimination framework is correct, i.e. that it maintains the invariant of Definition 4.3.1.

Theorem (Correctness of Substitution) 1. \( SOL(\text{sub}(EQS, i, j)) \subseteq SOL(EQS) \)

Proof. By definition 4.2.5, we assume \( \forall \psi \in SOL(EQS) : \psi(X_i) \subseteq f_i(\bigcap_{k \in \text{pred}(i)[j]} \psi(X_k) \cap \psi(X_j)) \)
- By commutativity of \( \cap \) and definition of sub
\[
f_i(\bigcap_{k \in \text{pred}(i)[j]} \psi(X_k) \cap \psi(X_j)) \subseteq f_i(\bigcap_{k \in \text{pred}(i)[j]} \psi(X_k) \cap f_i(\bigcap_{k \in \text{pred}(i)} \psi(X_k)))
\]
- By monotonicity
\[
\bigcap_{k \in \text{pred}(i)[j]} \psi(X_k) \cap \psi(X_j) \subseteq \bigcap_{k \in \text{pred}(i)[j]} \psi(X_k) \cap f_i(\bigcap_{k \in \text{pred}(i)} \psi(X_k))
\]
- By equivalence
\[
\psi(X_j) \subseteq f_i(\bigcap_{k \in \text{pred}(i)} \psi(X_k))
\]
- By Assumption T

Before we prove that the algebraic loop-breaking operation of the elimination framework is correct, we prove that performing widening and narrowing maintains a solution.

Lemma (Correctness of Widening) 2. \( SOL(\text{widen}(EQS, i)) \subseteq SOL(EQS) \)

Proof. By definition 4.2.5, we assume \( \forall \psi \in SOL(EQS) : \psi(X_i) \subseteq f_i(\bigcap_{k \in \text{pred}(i)[i]} \psi(X_k) \cap \psi(X_j)) \)
- By commutativity of \( \cap \) and definition of sub
\[
f_i(\bigcap_{k \in \text{pred}(i)[i]} \psi(X_k) \cap \psi(X_j)) \subseteq f_i(\bigcap_{k \in \text{pred}(i)[i]} \psi(X_k) \cap f_i(\bigcap_{k \in \text{pred}(i)} \psi(X_k)))
\]
- By widen
\[
f_i(\bigcap_{k \in \text{pred}(i)[i]} \psi(X_k) \cap \psi_\perp) \subseteq f_i(\bigcap_{k \in \text{pred}(i)[i]} \psi(X_k) \cap f_i(\bigcap_{k \in \text{pred}(i)} \psi(X_k)))
\]
- By monotonicity
\[
\bigcap_{k \in \text{pred}(i)[i]} \psi(X_k) \cap \psi_\perp \subseteq \bigcap_{k \in \text{pred}(i)[i]} \psi(X_k) \cap f_i(\bigcap_{k \in \text{pred}(i)} \psi(X_k))
\]
- By equivalence
\[
\psi_\perp \subseteq f_i(\bigcap_{k \in \text{pred}(i)} \psi(X_k))
\]
- By definition T

Lemma (Correctness of Narrowing) 3. \( EQS' = SOL(\text{widen}(EQS, i)) \rightarrow SOL(\text{narrow}(EQS', i)) \subseteq SOL(EQS) \)

Proof. Assume widening
Let narrowing solution be \( \psi_n(X) = f_i(\bigcap_{k \in \text{pred}(i)[i]} X_k \cap \psi_\perp) \)
We prove \( \psi_n(X) \subseteq f_i(\bigcap_{k \in \text{pred}(i)[i]} \psi(X_k)) \)
By definition of \( \psi_n(X) \)
\[
f_i(\bigcap_{k \in \text{pred}(i)[i]} X_k \cap \psi_\perp) \subseteq f_i(\bigcap_{k \in \text{pred}(i)[i]} \psi(X_k))
\]
- By monotonicity
\[
\bigcap_{k \in \text{pred}(i)[i]} X_k \cap \psi_\perp \subseteq \bigcap_{k \in \text{pred}(i)[i]} \psi(X_k)
\]
- By definition of meet \( \bigcap_{k \in \text{pred}(i)[i]} \psi_\perp \subseteq \bigcap_{k \in \text{pred}(i)[i]} \psi(X_k) \)
- By definition of \( \psi_n \)
From the proofs of widening and narrowing we can prove that the loop breaking solution is correct.

**Theorem (Correctness of Loop-Breaking) 4.** $SOL(lb(EQS, i)) \subseteq SOL(EQS)$

*Proof.* We split the process of loop-breaking into widening and narrowing. - By Lemma 2 and Lemma 3

$\square$
Chapter 5

Implementation

In this chapter of the thesis we describe the implementation of the technique of this thesis. The analysis is implemented as a function level pass in the LLVM compiler framework. We present an overview of the implementation, followed by more detailed explanations of each component and usage of external libraries.

![High-Level Implementation Diagram](image)

Figure 5.1: High-Level Implementation Diagram
5.1 Architecture

In Figure 5.1 a high-level architecture is shown. Each block represents a component of the analysis, with the range analysis representing the implementation of the presented approach. The lines indicate a usage of a component. As shown in Figure 5.1 the analysis code uses several external libraries. The LLVM library contains all specific code to parse and manipulate the LLVM source language. The GiNaC library is used to lift the interval domain to a symbolic domain. It contains rewriting capabilities of general arithmetic rules. However, we add rewrite rules for min and max operations. Finally an external implementation of an efficient elimination data flow analysis is used to correctly sequence substitution and loop-breaking operations defined in the analysis code.

Below we summarise each component within the range analysis. A detailed description is given in the next section.

- LLVM Range Analysis Pass: This component contains the main class for the analysis. The class integrates into the LLVM framework by implementing the Instruction Visitor Class and the Function Pass class.
- Function Space: The function space component contains classes that represent types of equation elements such as a variable, function, or fixpoint function.
• **Domains:** The domain component consists of the interval domain and symbolic domain classes. The symbolic domain class interfaces with GiNaC to assist in symbolic arithmetic. Additionally, we extend GiNaC to incorporate two new classes, namely, min and max. For these classes we implement rewrite equations so their representation can be reduced to a normal form.

• **Transition Functions:** Transition functions represent individual LLVM operations such as load, store, br, binary operations etc.

• **Environment and Transitions:** The environment is the mapping of instructions and values. A transition is a mapping to a basic block label to an environment as described in the abstract semantics.

In Figure 5.2 a component dependency diagram is shown. Each block represents a component of the analysis. The lines indicate a component dependency on another component. Below we present a detailed description of each component.

### 5.2 LLVM Range Analysis Pass

The FInterval class is the main class in the implementation, in the sense that it is responsible for setting up the equation system, initiating the elimination and solving the equation system. The representation of the equation system is shown in Figure 5.3. This construction is explained in more detail in the next section.

We can see in Figure 5.3 that every component of an equation system is represented by a class. The initiating function in the FInterval class is runOnFunction. This function is called for every LLVM function. Within runOnFunction the elimination framework is setup. The loop in Listing 5.1 is responsible for creating data flow equations for each basic block.

```cpp
for (Function::iterator b = F.begin(), be = F.end();
     b != be; i++, ++b) {
    // This function depends now on a state and instruction
    ffun* fun = FInterval::createFun(&(*b), i);
    fs_equ* equ = new fs_equ(&(*b), i, fun);
    fvar* var = new fvar(&(*b), i, equ);
    var_map.insert(std::make_pair(&(*b), var));   // int |-> (bb*, var*)
    fw->add_equation(equ);  // Add the equation to the framework
}
```

Listing 5.1: Creating of the Equation System

Every equation is added to the framework through the add_equation function. By inheriting from the LLVM classes FunctionPass and InstVisitor the FInterval implements a function level analysis that utilises the visitor pattern and "hooks into" the instruction class functions, as is common in the visitor pattern architecture. This enables a ffun to have associated semantics of the sequence of instructions in a basic block. When a ffun is created, visitor functions for each LLVM instruction inside a basic block are called.

```cpp
trans* visitBranchInst(BranchInst& i)
{
    // Only care about conditional
    if (i.isUnconditional()) { return NULL; }
    Value* v = i.getCondition();
}
```

Listing 5.2: Visitor Function for Branch Instruction
Equation $X_i = f_i(X_{j1}, X_{j2}, \ldots)$

Basic Blocks

Classes

Class $fs_{\_equation}$
Class $fvar$
Class $ffix$

Equation $X_i = F^*(f_i(X_{j1}, X_{j2}, \ldots), X_i)$

Classes

Class $fs_{\_equation}$
Class $fvar$
Class $ffun$

Figure 5.3: Equation System Construction
if (!isa<ICmpInst>(v) || isa<FCmpInst>(v))
{
    return NULL;
}
CmpInst* cins = static_cast<CmpInst*>(v);
return new br_trans(cins, i.getSuccessor(0), i.getSuccessor(1));

Listing 5.2: A Visitor Function for the LLVM ICmp Instruction

Listing 5.2 shows the visitor function for a branch instruction. Each of these functions returns a transition function that represents a sequential flow function, i.e. a basic block is composed of several of these functions. Next, dependencies are set up. This is shown in Listing 5.3.

fw->set_root(var_map[&(*F.begin())]->get_eq());
// Set up dependencies and function arguments
// For all basic blocks
for (Function::iterator b = F.begin(), be = F.end(); b != be; ++b)
{
    fs_equ* equ = (var_map[&(*b)])->get_eq();
    // For all Preds
    for (pred_iterator PI = pred_begin(&(*b)), E = pred_end(&(*b));
        PI != E; ++PI)
    {
        BasicBlock* pred = *PI;
        fvar* pred_var = var_map[pred];
        fvar* var = var_map[&(*b)];
        static_cast<ffun*>(equ->get_rhs())->add_dependence(pred_var->clone());
        fw->add_dependence(pred_var->get_eq(), var->get_eq()); // What dependency to use
    }
}
fw->main_elim();
fw->main_propagate();

Listing 5.3: Creating of the Equation System

First the root node is set in the framework, and for every function, predecessors are connected with their successor equation. Once the setup is complete, the elimination framework is triggered to eliminate the equation system. A final iteration is conducted where each equation system is solved and the result is printed out, as shown in Listing 5.4.

for (std::map<BasicBlock*, fvar*>::iterator it = var_map.begin(); it != var_map.end(); ++it)
{
    binding* a = (*it).second->get_eq()->get_rhs()->solve();
    std::cout << "------------------------------------------\n";
    std::cout << "for fvar " << (*it).second << "\n";
    std::cout << "-----\n";
    a->print();
    ii++;
}

Listing 5.4: Creating of the Equation System
5.3 Function Space

The function space component implements the classes that form a flow equation. A class diagram depicting the relationships is presented in Figure 5.4.

```cpp
virtual void loop_break()
{
  fspacet var = new fvar(bblock, -1);
  fspacet oldrhs = rhs;
}
```

Figure 5.4: Class Diagram of LLVM Range Analysis Implementation

The fs_equation class implements the elimination data flow frameworks substitution and loop break operations as was presented in Chapter 2 and Chapter 4. Their implementations are shown in Listing 5.5.
The substitution implementation, is straightforward. The substitution operation is called on the right-hand-side of the equation (either a function of fixpoint object). This operation will perform a substitution of a right-hand-side variable with an equation. Loop-breaking wraps a fixpoint class over the right-hand-side of the equation. That way, any call to the equation’s right-hand-side will be a call to the fixpoint operations.

Solving equations is implemented through the solve functions, that are defined in classes ffun and ffix. Solving ffun involves iterating through all predecessor functions and performing any pending condition joins. As LLVM is in SSA form and we define a \( \phi \) node as a meet of all arguments, a meet of predecessors is implicit in the \( \phi \) transformation function. Once all predecessors are solved, a final merged environment is obtained and a list of transformation functions are applied to the environment. The resultant environment is the solution for that equation. Solving ffix objects requires additional widening and narrowing passes as described in Chapter 4.

The scenario where there are several ffix objects within an equation requires special attention. For example, an equation of the following structure may exist:

\[
X_i = F^*_i(f_i(...F^*_j(f_n(X_i, X_j)..., X_j), X_i)
\]

Such a scenario requires a vector of variables to be passed through each solve function. Each ffvar in the list contains an environment. When a variable is encountered during solving, instead of solve being called, its environment will be retrieved from the list.

Another issue this scenario introduces is widening and narrowing occurrences in a fixpoint function. Before widening and narrowing can proceed in a fixpoint function, any sub-fixpoints must be solved. This requires a fixpoint to know if it is a sub fixpoint or not. This problem is overcome in the implementation by a check on the variable vector to determine if is empty or not. An empty list indicates that no fixpoints exists above the current fixpoint, or there be a reoccurring variable in the vector. The implementation is depicted in Listing 5.6.
Listing 5.6: Solving in Fixpoints

The Listing shows that two implementations of solve are required. The solve with no arguments, creates a fresh vector, the solve_aux function is called for sub composite (ffun or ffix) and passes the vector on.

Widening and narrowing is implemented by the doWiden and doNarrow functions called within the ffix’s solve_aux function. The calls to these functions is depicted in Listing 5.7.

Listing 5.7: Solving in Fixpoints

The algorithm directly follows the approach in Chapter 4. A call to doWiden occurs with an old state and new state as arguments. doWiden will modify the old state if any variable definition values have widened. Next, doNarrow is called with a narrowing state and the existing state. If a better result has been obtained, the narrowing process will repeat until a fixpoint solution is found.
5.4 Domains and Environments

Domains and environments are implemented with the abstract semantics in mind. In the environment component two types of mappings are defined. Transitions and environments. A transition is simply a mapping between a label (pointer to a basic block) and an environment object. Environments are mappings between instructions (instruction pointers) and values.

Values are implemented as intervals of `ex` (expression) objects. These objects are defined within the GiNaC library and they represent algebraic expressions. As we define in our symbolic domain in Chapter 3, algebraic expressions may consists of min and max constructs. These are not standard arithmetic operations and are implemented as extensions to GiNaC. The operations min and max are defined as classes that are registered with the GiNaC library as depicted in Listing 5.8. Each class has several member functions that further reduce the arguments of the operations. They apply rewrite rules that use GiNaC’s pattern matching ability to detect if an expression needs to be reduced. An example rewrite rule is depicted in Listing 5.9.
class maxf : public basic
{
    GINAC_DECLARE_REGISTERED_CLASS(maxf, basic)
public:
    maxf(ex left, ex right);
    ex eval(int level) const;
    ex rewrite() const;
private:
    bool apply1(ex* result) const;
    bool apply2(ex* result) const;
    bool apply3(ex* result) const;
    bool apply4(ex* result) const;
    bool apply5(ex* result) const;
    bool apply6(ex* result) const;
    bool apply7(ex* result) const;
    bool apply8(ex* result) const;
public:
    ex l;
    ex u;
protected:
    void do_print(const print_context & c, unsigned level = 0) const;
};

// min(x, x + num) = x
// min(x + num, x) = x
bool maxf::apply1(ex* result) const
{
    std::cout << "Rule 1 -- \n";
    ex arg1 = wild(1);
    ex arg2 = add(wild(2), arg1);
    bool ln = hasNumeric(l);
    bool rn = hasNumeric(u);
    bool neg = hasNegNumeric(u);
    bool p1 = ( l.match(arg1) && u.match(arg2));
    bool p2 = ( l.match(arg2) && u.match(arg1));

    std::cout << "ln : " << ln << " rn : " << rn << " p1 : " << p1 << " p2 : " << p2 << "\n";
    if ( p1 && rn ) // is right > left
    {
        bool p3 = ( l == u.op(0) ) || ( l == u.op(1));
        if (p3)
        {
            std::cout << "applied rule 1.1 to u " << u << " l " << l << "\n";
            *result = neg ? l : u;
            return true;
        }
    }
    ...
    ...
}

Listing 5.8: Max GiNaC class

Listing 5.9: A Rewrite Rule for the Min Class
5.5 Implementation Summary

The LLVM implementation of this chapter demonstrates the feasibility of using our program analysis technique as a regular compiler pass acting on low-level unstructured languages. The implementation uses several external libraries and algorithms. GiNaC was used because it is developed in C++ and was easily integrated into the LLVM code base. Additionally, it is a very flexible library that allows extensions and additions. This is a decisive factor as several non standard operations such as symbolic min and max are required in our implementation. Further, GiNaC is open source unlike potentially more powerful CAS systems such as Mathematica [82]. Additionally we use an elimination based algorithm developed by Vugranam C. Sreedhar [75]. This algorithm is much more efficient than the Gaussian algorithm presented in Chapter [2] and is more scalable to larger code bases.

This implementation may serve as a template for specific analysis such as assertion elimination, detection of array out of bounds errors and numerical instability etc. For example, this analysis could be easily modified for detecting numerical instability. A floating point number domain would be implemented that instead of tracking variable value bounds, tracks variable error bounds. An error that diverges to infinity would indicate a potential variable that is numerically unstable.

Additionally, symbolic nature of the domain adds further possibilities of improving bounds and incorporating function calls and arguments. In the future work section of the conclusion we elaborate on some further improvements to the symbolic interval domain. In the next Chapter we present several test programs using our implementation.
Chapter 6

Results

This section of the thesis summarises the results of the analysis implementation through a number of case studies. Each case study is presented as a non-SSA (non Single Static Assignment) program written in the language of Figure 3.2. Equivalent SSA LLVM programs and actual implementation analysis output can be found in Appendix A of the thesis. Case studies 1 - 4 are focused on the basic interval domain. In these studies we present programs with different control flow structures and comment on the precision of the analysis. Case studies 5 - 6 focus on the symbolic domain, for both case studies we present symbolic and non-symbolic results, highlighting the difference in precision between them.
6.1 Case Study 1: Branching

Program Description

This test demonstrates interval widening through conditional branching. The result should be sound but some loss in precision is expected as all paths are taken in the analysis. The program contains a variable \( a \) that is initially assigned the symbolic or bottom value\(^1\). Both branches are taken as they are both possible. At each branch a different constant value is assigned and the program branches to block 3. At block 3 the variable \( a \) is incremented by one.

```
1 B0:
2   int x := f();
3   if (x <= 10) goto B1; goto B2;
4 B1:
5   x := 5;
6   goto B3;
7 B2:
8   x := 15;
9   goto B3;
10 B3:
11   x := x + 1;
```

Listing 6.1: A Branching Example

Output

<table>
<thead>
<tr>
<th>Block</th>
<th>( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>([-\infty, \infty])</td>
</tr>
<tr>
<td>B1</td>
<td>[5, 5]</td>
</tr>
<tr>
<td>B2</td>
<td>[15, 15]</td>
</tr>
<tr>
<td>B3</td>
<td>[6, 16]</td>
</tr>
</tbody>
</table>

Table 6.1: Branching Analysis Results

Results Analysis

The results show that as expected there is an interval widening. The variable \( x \) has an initial value of symbol or bottom. One branch takes a path where variable \( x \) is assigned [5, 5] and the other branch takes the path where variable \( x \) is assigned [15, 15]. This ultimately results in an interval widening of [5, 15] when both paths are merged at block 3 and [6, 16] after the additional of the interval [1, 1] to the existing value.

The result is sound, covering all possible values that \( a \) could have. However, the value is imprecise as it allows \( x \) to have values (at block 3) between 6 and 16 when it is evident that \( x \) can only be 6 or 16. This is imprecision is attributed to the interval domain.

\(^1\)Depending on the domain, but for this example it doesn’t matter
6.2 Case Study 2: Irreducible Graphs

Program Description

This test demonstrate the analysis on irreducible graphs with a slightly different example from the one presented in Chapter 4. Only unstructured programs can produce irreducible graphs. This example shows a divergent program with a single variable \( x \) that is incremented in blocks 1 and 2. Block 2 always increments after block 1, so it should have a higher lower bound.

```
B0:
  int x := 1;
  if (x < 2) goto B1 : goto B2;
B1:
  x := x + 1;
  goto B2;
B2:
  x := x + 1;
  goto B1;
```

Listing 6.2: A Irreducible Graph Example

Output

<table>
<thead>
<tr>
<th>Block</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>[1, 1]</td>
</tr>
<tr>
<td>B1</td>
<td>[1, ∞]</td>
</tr>
<tr>
<td>B2</td>
<td>[2, ∞]</td>
</tr>
</tbody>
</table>

Table 6.2: Branching Analysis Results

Results Analysis

The analysis results show that variable \( x \) is divergent at both blocks, utilising widening to accelerate convergence to \( ∞ \). Additionally, the results correctly show that variable \( x \) in block 2 has a lower bound of 2. The analysis can detect that the branch from block 0 to block 2 is impossible so that path is disregarded (value form path is \( ⊤ \)) when calculating the merge at block 2. The resulting value for variable \( x \) is therefore \( [2, ∞] \). This result is precise in the sense that the final intervals contain a trace of all possible values in the loop.
6.3 Case Study 3: Widening and Narrowing

Program Description

We demonstrate widening and narrowing through a while loop program as presented in [12]. In this test we want to demonstrate that our approach can perform the same analysis on unstructured languages as existing techniques can on structured languages. This example presents a simple while loop program that increments the variable \( x \) as long as \( x \) is less than or equal to 10.

```plaintext
B0:
  x := 1;
B1:
  if (x <= 10) goto B2: goto B3;
B2:
  x := x + 1;
goto B1;
B3
return;
```

Listing 6.3: A Widening and Narrowing Example

Output

<table>
<thead>
<tr>
<th>Block</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>[1, 1]</td>
</tr>
<tr>
<td>B1</td>
<td>[1, 11]</td>
</tr>
<tr>
<td>B2</td>
<td>[2, 11]</td>
</tr>
<tr>
<td>B3</td>
<td>[11, 11]</td>
</tr>
</tbody>
</table>

Table 6.3: Widening and Narrowing Analysis Results

Results Analysis

The results match the expected results from our theoretical explanation of this problem in Chapter 4. We can see the results show that widening and narrowing results in the variable \( x \) having a fixpoint of [1, 11]. When the path where the inverse condition \( (x > 10) \) holds the value of \( x \) is constrained at the lower bound to a value greater than 10, resulting in the very accurate value of [11, 11]. In this case, narrowing is able to give excellent precision of [11, 11] at block 3, the most precise value possible.
6.4 Case Study 4: Bubble Sort

Program Description

We demonstrate the analysis results on a motivating example from Chapter 1. We present a simplified version of the program focusing on the incrementing of the index variables $i$, $j$ and $k$. We can see that the outer loop increments the variable $i$ if $i <= 4$. In inner loop increments $k$ and $j$ if $j <= 4$.

```
B0:
  k := 0;
  i := 0;
  goto B1;
B1:
  if i <= 4 goto B2 : goto B6;
B2:
  j := 0;
  goto B3;
B3:
  if (j <= 4) goto B4 : goto B5;
B4:
  j := j + 1;
  k := k + 1;
  goto B3;
B5:
  i := i + 1;
  goto B1;
B6:
return;
```

Listing 6.4: A Bubble Sort Example

Output

<table>
<thead>
<tr>
<th>Block</th>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>[0, 0]</td>
<td>$\top$</td>
<td>[0,0]</td>
</tr>
<tr>
<td>B1</td>
<td>[0, 5]</td>
<td>[0, 5]</td>
<td>[0, $\infty$]</td>
</tr>
<tr>
<td>B2</td>
<td>[0, 4]</td>
<td>[0, 0]</td>
<td>[0, $\infty$]</td>
</tr>
<tr>
<td>B3</td>
<td>[0, 4]</td>
<td>[0, 5]</td>
<td>[0, $\infty$]</td>
</tr>
<tr>
<td>B4</td>
<td>[0, 4]</td>
<td>[1, 4]</td>
<td>[1, $\infty$]</td>
</tr>
<tr>
<td>B5</td>
<td>[1, 5]</td>
<td>[5, 5]</td>
<td>[1, $\infty$]</td>
</tr>
<tr>
<td>B6</td>
<td>[5, 5]</td>
<td>[5, 5]</td>
<td>[0, $\infty$]</td>
</tr>
</tbody>
</table>

Table 6.4: Branching Analysis Results
Results Analysis

The results mimic very closely the expected results from the Chapter 1. However, the variable $k$ is over approximated with the sound value of $[0, \infty]$. This over approximation results from our analysis’s inability to infer a relationship between $k$ and $j$ and $i$. 
6.5 Case Study 5: Symbolic Elimination

Program Description

In this test we demonstrate the symbolic domain and its ability to achieve better bounds due to its immunity to absorption of values i.e. addition or subtraction of infinity is infinity. In this program two symbols are generated from function calls. Variables $a$ and $b$ have the same symbolic value of $s_0$ generated from the call $f()$. Variables $c$ and $d$ have the symbolic value of $s_1$ generated from the call $g(1)$. Variable $x$ contains the value of the subtraction of $a$ and $b$ as well as $c$ while adding 4. Variable $z$ is assigned the value if subtracting symbol $s_1 + 4$ from $z$.

```c
1 B0:
2 int a := f();
3 int b := f();
4 int c := g(1);
5 int d := g(1);
6 int x := a - b - c + 4;
7 int z := x - g(1) + 4;
```

Listing 6.5: A Symbolic Interval Example

Output

<table>
<thead>
<tr>
<th>Block</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>x</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>$[s_0, s_0]$</td>
<td>$[s_0, s_0]$</td>
<td>$[s_1, s_1]$</td>
<td>$[s_1, s_1]$</td>
<td>$[s_1 + 4, s_1 + 4]$</td>
<td>$[0, 0]$</td>
</tr>
</tbody>
</table>

Table 6.5: Symbolic Analysis Results

<table>
<thead>
<tr>
<th>Block</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>x</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>$[-\infty, \infty]$</td>
<td>$[-\infty, \infty]$</td>
<td>$[-\infty, \infty]$</td>
<td>$[-\infty, \infty]$</td>
<td>$[-\infty, \infty]$</td>
<td>$[-\infty, \infty]$</td>
</tr>
</tbody>
</table>

Table 6.6: Non Symbolic Analysis Results
Results Analysis

The results show that the symbolic values are eliminated in the computation resulting in accurate bounds for the variable $z$. The values of functions are symbolically evaluated and eventually eliminated. We can see in the assignment to $x$ the $a$ and $b$ are symbolically eliminated as they are represented by the same symbolic interval. In the assignment to $z$ a new symbolic interval is generated that eliminates the $c$ and by the discrete interval arithmetic rules $[4, 4] - [4, 4]$ reduces to $[0, 0]$, giving a very accurate result.

In the non-symbolic domain however, the variable $x$ absorbs the addition of $[4, 4]$ to $[\infty, \infty]$ and the entire expression results in $[\infty, \infty]$. The same problem is present in the assignment to $z$ again resulting in the value $[\infty, \infty]$. This is a safe bound, however, it is a very large over approximation, making the analysis unusable.
6.6 Case Study 6: Symbolic Elimination with Branching

Program Description

This test demonstrates how symbolic intervals can further improve variable bounds by generating intervals that exclude values within the interval as well as limiting bounds. In this example there is a branch that on both edges multiplies the symbolic value by a positive number.

```
1 B0:
2    x = f();
3    y := 0;
4    if x <= 1 goto B1: goto B2;
5 B1:
6    y := 2 * x;
7    goto B3;
8 B2:
9    y := 2 * x;
10   goto B2;
11 B3:
12   return;
```

Listing 6.6: A Symbolic Interval Example with Branching

Output

<table>
<thead>
<tr>
<th>Block</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>[s0, s0]</td>
<td>[0, 0]</td>
</tr>
<tr>
<td>B1</td>
<td>[s0, min(1, s0)]</td>
<td>[2 * s0, 2 * min(1, s0)]</td>
</tr>
<tr>
<td>B2</td>
<td>[max(s0, 2), s0]</td>
<td>[2 * max(s0, 2), 2 * s0]</td>
</tr>
<tr>
<td>B3</td>
<td>[min(s0, max(s0, 2)), max(min(s0, 1), s0)]</td>
<td>[min(2 * s0, 2 * max(s0, 2)), max(2 * min(s0, 1), 2 * s0)]</td>
</tr>
</tbody>
</table>

Table 6.7: Symbolic Analysis Results

<table>
<thead>
<tr>
<th>Block</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>[−∞, 0]</td>
<td>[0, 0]</td>
</tr>
<tr>
<td>B1</td>
<td>[−∞, 1]</td>
<td>[−∞, 2]</td>
</tr>
<tr>
<td>B2</td>
<td>[2, 0]</td>
<td>[4, 0]</td>
</tr>
<tr>
<td>B3</td>
<td>[−∞, 0]</td>
<td>[−∞, 0]</td>
</tr>
</tbody>
</table>

Table 6.8: Non Symbolic Analysis Results

Results Analysis

The results show improves on the non symbolic intervals not by its upper/lower bounds (as y can be any number) but by producing sound interval values where some values inside the interval are excluded. For example, in the above analysis, the discrete intervals that the symbolic interval of y generates cannot contain odd numbers. For example at block 1, y is bounded by 2 and when it is
lower than 2, it can only generate intervals with two identical even numbers. Conversely, the on-
symbolic interval cannot generate such precision as 2 * infinity = infinity and the doubling property 
that generates even numbers is lost through absorption.

6.7 Results Summary

The results show that the analysis can achieve sound results where precision is limited only by the 
domain, rather than the analysis technique itself. The results further demonstrate the improved preci-
sion using symbolic intervals. Symbolic intervals achieve better precision by allowing symbols to be 
canceled and through not absorbing arithmetic calculations, which occurs in non-symbolic intervals. 
Despite the improvement in precision with symbolic intervals, we believe that the symbolic domains 
ability to discover closed forms of loops would provide even better precision and would present an 
alternative widening and narrowing mechanism that is currently employed. In any case, the results 
demonstrate that our proposed technique is in no way inferior to any of the state-of-the-art techniques 
and can analyse a greater range of programs i.e. programs not limited by structural composition.
Chapter 7

Related Work

In this chapter of the thesis we summarise previous work relating to abstract interpretation based analyses. Abstract interpretation has existed since the mid-seventies \cite{38,12} and can be seen as one of the fundamental theories in program analysis. We begin this chapter with a brief overview of the applications of abstract interpretation and then focus on related work specifically in abstract interpretation based range analysis and detecting loops in programs.

7.1 Abstract interpretation in Literature

The uses of abstract interpretation are manifold. The theory under-pins existing program analysis techniques such as iterative data flow analysis \cite{12} and to some extent model-checking \cite{66}. Range analysis based abstract interpretation is used to perform roundoff error analysis \cite{22,45}, assertion elimination \cite{1}, numerical analysis \cite{11,35}, value range analysis \cite{23,69,35,11,12}, among others. Additionally, abstract interpretation is present in other types of analyses. These include, control flow analysis \cite{36,70}, escape analysis \cite{60,16}, heap analysis \cite{40,64}, pointer analysis \cite{15}, symbolic analysis \cite{35,7,8}, concurrency analysis \cite{9}, mobility analysis \cite{19}, probabilistic analysis \cite{52}, hardware description verification \cite{79}. Abstract interpretation, unlike data flow analysis, is not limited to imperative languages, and has been used to analyse functional \cite{30,36,70} and logic programs \cite{74,46}.

7.1.1 Range Analysis based Abstract Interpretation

A natural domain for abstract interpretation, is the range-based interval domain proposed by Cousot and Cousot in 1976 \cite{12}. Intervals have been shown to be in the class of polyhedral based domains. These domains utilise linear programming to ensure more accurate bounds by reflecting relationships between variables. By incorporate relationships between variables linear inequalities can be discovered between variables which can result in improved bounds. Polyhedra based approaches are shown to be useful for numerical analysis \cite{13}, delay analysis \cite{26}, quantitative time properties \cite{27} among others. This approach however, can result in large overheads for storing an unlimited amount of constraints. Practically, the number of these constraints is usually fixed and operations are restricted as reflected in the Octagonal domain \cite{50}. The Octagonal analysis restricts the type of relations that are observed. This domain is shown in \cite{50,49,51} to be effective and practical for use in numerical instability analysis. Symbolic-based domains are another method of providing more precision by also taking into account variable relations. Symbolic ranges are shown in \cite{35,11} to be effective in range
based analysis, particularly in the area of numerical stability. Another domain that may be classed as a range analysis domain is the congruence domain. Congruence analyses can discover variable relations based on congruence relations. In the paper [39], a SAT solver is utilised to derive invariants that are systems of congruence equations where the modulo is a power of two. Range analysis have also been shown to be able to compute symbolic computational complexity bounds. The analysis presented in [23, 69], is able to find bounds on the number of iterations a loop can take. This approach augments the program with counters to extract numeric symbolic bounds and is primarily used to prove program termination.

7.1.2 Loop Detection

When comparing the work of this thesis to such related work, although the state-of-the-art domains presented in these works achieve much better bounds, they still appear to rely on structural composition in languages when convergence acceleration is performed. None of these methods address abstract interpretation on low-level languages adequately. However, previous research has been conducted on removing goto statements from code which often required goto loop detection. In [54], regular expressions are employed to detect and remove goto statements from legacy programs. In [18] the authors take an abstract syntax tree approach to detect goto statements. The work in [59] uses formal semantics and congruent equivalence transformations to detect and remove goto statements. Additionally, several algebraic elimination-based frameworks have been proposed that could be utilised in an analysis such as the proposed thesis solution. Other than the Gaussian elimination approach and the Sreedhar-Gao-Lee DJ graph based analysis used in the thesis implementation [75], four other approaches exist that are algebraic in their nature, namely, the Allen-Cocke interval analysis [65], Hecht-Ullman T1-T2 analysis [65], Tarjan interval analysis [65], and Graham-Wegman analysis [65].

7.2 Abstract Interpretation-Based Tools

Several abstract domain libraries have been developed for use in static analysis. These libraries implement a representation of abstract properties such as logical operations, properties transformers, widening etc. The NewPolka [33] is a convex polyhedral based library whose constraints have rational coefficients. The octagon abstract domain library is a limited polyhedral based library that limits the domain to an octagon abstract which only allows addition, negation, division in inequalities. The Parma Polyhedral library handles convex polyhedra that can defined as intersection of a finite number of hyper-spaces with rational coefficients.

Additionally, several tools have been implemented for a variety of application domains. The Airac5 is a static analyser developed by Seoul National University [73] for automatic verification of buffer overruns by C programs. Astree is a static analyser from Ecole normale superierure. It can prove the absence of runtime errors, specialising in control/command programs written in C [11]. C Global Surveyor [55], developed by NASA, is a tool that finds runtime errors in C programs. TVLA [80], developed by Tel Aviv University, is a tool for experimenting with abstract interpretation using logical descriptions of programs. Terminator [48] is a tool developed by Microsoft Research Cambridge for proving program termination and general liveness properties of programs written in C.
Chapter 8

Conclusion

In this thesis we have presented a novel program analysis technique that performs an unbounded analysis on unstructured imperative low-level languages. We have shown that by combining the theories of abstract interpretation with elimination-based data flow analysis we are able to present a flexible solution that can perform range analysis for program variables on unstructured languages. Unbounded abstract domains such as the interval domain of the range analysis pose difficulties for state-of-the-art techniques, which limit input programs to programs that are structurally composed. We have implemented our novel program analysis technique in the LLVM compiler framework. To show the feasibility of our technique, experiments were conducted in the LLVM compiler framework with a suite of input programs. Our program analysis framework has been designed such that it can be deployed in other compiler frameworks as well and it is not limited to the LLVM compiler framework.

8.1 Summary

In Chapter 3 of this thesis we presented two unbounded domains, namely, intervals and symbolic intervals. We defined a canonical low-level language and specified its concrete and abstract semantics for both domains. Using the theory of abstract interpretation we proved a Galois connection between the concrete and abstract semantics as well as semantic soundness. In Chapter 4 we presented our proposed technique and showed how it performs program analysis on our abstract semantics from Chapter 3. We proved that our approach always produces a solution that is within the solution space. In Chapter 5 we described in detail how our approach was implemented in the LLVM compiler framework. In Chapter 6 we presented the feasibility of the implementation by discussing several input programs that typically pose difficulties for previous abstract interpretation techniques.

8.2 Future work

The work in this thesis has raised new research possibilities and questions. We propose several future investigations and highlight areas where the approach and implementation could be improved.

- **Improve Precision:** The description and implementation of the symbolic interval domain in this thesis is elementary. It relies on widening and narrowing for program cycles. A more powerful approach would be to implement symbolic recurrences to discover closed form representations of loops. Additionally, extending the domain to include variable relationships would be a very powerful addition. Such a feature would result in more accurate variable bounds and...
also allow the analysis to determine unreachable code. The flexible nature of our approach allows techniques other than widening and narrowing to be implemented within the algebraic operation of our presented elimination framework. The framework could further be used as a test bed for new techniques for finding accurate bounds in programs.

- **Extend the Implementation:** The implementation of the analysis in this thesis is a prototype, and not all LLVM instructions and constructs are handled. This prototype may be extended and integrated into the LLVM core project in the future. This will require that all instructions are implemented as well as more domains including floating point numbers. Additionally, the implementation of our program analysis technique will enable new analyses including buffer-overflow detection, integer overflow detection, and detecting numerical instability in programs. From our conversations with LLVM developers, this contribution will be greatly appreciated by the LLVM community.

- **Efficiency:** In this thesis we have not attempted to investigate the efficiency of our approach in any detail. While we believe our analysis performs well for small input programs, further investigations are necessary to ensure that the analysis will scale for large code bases.

- **Analysis Output:** The implementation has very coarse analysis output, that requires a user to inspect every variable value at every program point to discover bugs. Further work on pretty printing of analysis output and more automated detection of errors is desirable.

- **Explore the Algebraic Aspects of the Technique:** The approach in this thesis appears to have delegated algebraic properties from a structured programming language to the analysis itself. The algebraic aspects of the technique would be interesting to investigate in the future. Additionally, it would be interesting to formalise the analysis approach using an interactive theorem provers such as Coq [32] or PVS [76].

### 8.3 Reflection

In this thesis we have learned that the traditional understanding of data flow analysis and abstract interpretation is not all together accurate. Previous discussions [66, 12] have often dismissed or ignored the possibility of performing data flow analysis on unbounded domains. Further, the possibility of performing abstract interpretation on unstructured imperative programs to our knowledge has not been investigated. The technique in this thesis extends the discussion of the relationship between data flow analysis and abstract interpretation. We have shown that by combining elimination-based data flow analysis and abstract interpretation unbounded program analyses can be performed on unstructured imperative programs. Additionally, in this thesis we have learned that a symbolic approach to range analysis can result in a more accurate and powerful outcomes. Further work in this area may lead to more accurate symbolic techniques to achieve more precise variable bounds.
Appendix A

LLVM Analysis Output

1. Branching Program:

```assembly
; ModuleID = './t5phi.bc'
   :16:32"
target triple = "i386-pc-linux-gnu"
define i32 @main () nounwind {
   ; <label>:0
   %1 = add nsw i32 0, 15
   %2 = icmp sle i32 %1, 10
   br il %2, label %3, label %5
   
   ; <label>:3
   %4 = add nsw i32 0, 5
   br label %5
   
   ; <label>:5
   %a.0 = phi i32 [ %4, %3 ], [ %1, %0 ]
   %6 = add nsw i32 %a.0, 1
   ret i32 0
}
```

Listing A.1: A Branching Example

```
Output of Range Analysis
-----------------------------------------------
for fvar X0:0
  0x98a7c70---

Transition Start
Label -> 0x98a0850
Environment contains the following mappings(Size:1)
I -> 0x98a07e8, V -> T 1

-----------------------------------------------
Transition End
```

71
Transition Start
Label -> 0x98a0880
Environment contains the following mappings (Size: 1)
I -> 0x98a07c8, V -> [15,15] 1
---
Transition End
---
for fvar X1:1
0x98a7d40---
---
Transition Start
for fvar X2:2
0x98a7c30---
---
Transition Start
Label -> 0x98a0880
Environment contains the following mappings (Size: 2)
I -> 0x98a07c8, V -> T 1
I -> 0x98a0950, V -> [5,5] 1
---
Transition End
---
for fvar X1:1
0x98a7d40---
---
Transition Start
---
Label -> 0x98a0880
Environment contains the following mappings (Size: 6)
I -> 0x98a0658, V -> T 0
I -> 0x98a07c8, V -> [15,15] 1
I -> 0x98a0850, V -> T 0
I -> 0x98a0950, V -> [5,5] 1
I -> 0x98a09c8, V -> [5,15] 1
I -> 0x98a0b10, V -> [6,16] 1
---
Transition End
---
End of Range Analysis

Listing A.2: Results
2. Irreducible Program:

```c
; ModuleID = "/irrphi.bc"
:16:32"
target triple = "i386-pc-linux-gnu"
define i32 @main() nounwind {
    ; <label>:0
    %1 = add nsw i32 0, 1
    %2 = icmp sle i32 %1, 3
    br il %2, label %3, label %5

    ; <label>:3 ; preds = %0, %5
    %a.0 = phi i32 [ %1, 0 ], [ %6, 5 ]
    %4 = add nsw i32 %a.0, 1
    br label %5

    ; <label>:5 ; preds = %0, %3
    %a.1 = phi i32 [ %1, 0 ], [ %4, 3 ]
    %6 = add nsw i32 %a.1 , 1
    br label %3
}
```

Output of Range Analysis

```
---
for fvar X0:0 0xa1a2a90---

Transition Start
Label -> 0xa19b848
Environment contains the following mappings(Size:1)
I => 0xa19b7c0 , V => [1,1] 1
---
Transition End

Transition Start
Label -> 0xa19b878
Environment contains the following mappings(Size:1)
I => 0xa19b7c0 , V => T 1
---
Transition End
---
for fvar X1:1 0xa1a2b98---

Transition Start
Label -> 0xa19b878
Environment contains the following mappings(Size:8)
```
31 I \rightarrow 0xa19b680, V \rightarrow T 0
32 I \rightarrow 0xa19b7c0, V \rightarrow [1,1] 1
33 I \rightarrow 0xa19b848, V \rightarrow T 0
34 I \rightarrow 0xa19b878, V \rightarrow T 0
35 I \rightarrow 0xa19b920, V \rightarrow [1.2147483649] 1
36 I \rightarrow 0xa19ba38, V \rightarrow [2.2147483650] 1
37 I \rightarrow 0xa19bad8, V \rightarrow [1.2147483648] 1
38 I \rightarrow 0xa19bb70, V \rightarrow [2.2147483649] 0

---

Transition End

for fvar X2:2
0xala2cc0

Transition Start

Label \rightarrow 0xa19b848

Environment contains the following mappings (Size : 8)
30 I \rightarrow 0xa19b680, V \rightarrow T 0
31 I \rightarrow 0xa19b7c0, V \rightarrow [1,1] 1
32 I \rightarrow 0xa19b848, V \rightarrow T 0
33 I \rightarrow 0xa19b878, V \rightarrow T 0
34 I \rightarrow 0xa19b920, V \rightarrow [1.2147483647] 1
35 I \rightarrow 0xa19ba38, V \rightarrow [2.2147483648] 1
36 I \rightarrow 0xa19bad8, V \rightarrow [1.2147483648] 1
37 I \rightarrow 0xa19bb70, V \rightarrow [2.2147483649] 0

---

Transition End

End of Range Analysis

Listing A.4: Results
3. Widening and Narrowing Program:

```c
; ModuleID = "/whilephi.bc"

:16:32"

target triple = "i386-pc-linux-gnu"

define i32 @main() nounwind {
    ; <label>:0
    %1 = add nsw i32 0, 1
    br label %2

    ; <label>:2 ; preds = %4, %0
    %3 = phi i32 [ %1, %0 ], [ %5, %4 ]
    br i1 %3, label %4, label %6

    ; <label>:4 ; preds = %2
    %5 = add nsw i32 %a.0, 1
    br label %2

    ; <label>:6 ; preds = %2
    %7 = add nsw i32 1, %a.0
    ret i32 0
}"
```

Listing A.5: A Widening/Narrowing Example

```
Output of Range Analysis

---

for fvar X0:0
0x99f7a88-----

Transition Start

Label -> 0x99f0800
Environment contains the following mappings(Size:1)
I -> 0x99f07c8 , V -> [1,1] 1

---

Transition End

---

for fvar X1:1
0x99f7b78-----

Transition Start

Label -> 0x99f0870
Environment contains the following mappings(Size:5)
I -> 0x99f06a8 , V -> T 0
I -> 0x99f07c8 , V -> [1,1] 1
I -> 0x99f0870 , V -> T 0
I -> 0x99f08c0 , V -> [1,10] 0
I -> 0x99f0b40 , V -> [2,11] 1

---
```
Environment contains the following mappings (Size: 5)
I $\rightarrow$ 0x99f06a8, V $\rightarrow$ T 0
I $\rightarrow$ 0x99f07c8, V $\rightarrow$ [1, 1] 1
I $\rightarrow$ 0x99f0870, V $\rightarrow$ T 0
I $\rightarrow$ 0x99f08c0, V $\rightarrow$ [11, 11] 0
I $\rightarrow$ 0x99f0b40, V $\rightarrow$ [2, 11] 1

Environment contains the following mappings (Size: 5)
I $\rightarrow$ 0x99f06a8, V $\rightarrow$ T 0
I $\rightarrow$ 0x99f07c8, V $\rightarrow$ [1, 1] 1
I $\rightarrow$ 0x99f0870, V $\rightarrow$ T 0
I $\rightarrow$ 0x99f08c0, V $\rightarrow$ [1, 10] 0
I $\rightarrow$ 0x99f0b40, V $\rightarrow$ [2, 11] 1

Environment contains the following mappings (Size: 7)
I $\rightarrow$ 0x99efb28, V $\rightarrow$ T 0
I $\rightarrow$ 0x99f06a8, V $\rightarrow$ T 0
I $\rightarrow$ 0x99f07c8, V $\rightarrow$ [1, 1] 1
I $\rightarrow$ 0x99f0870, V $\rightarrow$ T 0
I $\rightarrow$ 0x99f08c0, V $\rightarrow$ [11, 11] 0
I $\rightarrow$ 0x99f0b40, V $\rightarrow$ [2, 11] 1
I $\rightarrow$ 0x99f0bd0, V $\rightarrow$ T 1

End of Range Analysis
4. Bubble Sort Program:

```c
; ModuleID = "/motivatingphi.bc"
 :16:32"
target triple = "i386-pc-linux-gnu"
define i32 @main() nounwind {
  ; <label>:0
  br label %1
  ; <label>:1
  %k.0 = phi i32 [ 0, %0 ], [ %k.1, %9 ]
  %i.0 = phi i32 [ 0, %0 ], [ %10, %9 ]
  %2 = icmp sle i32 %i.0, 4
  br il %2, label %3, label %11
  ; <label>:3
  br label %4
  ; <label>:4
  %k.1 = phi i32 [ %k.0, %3 ], [ %8, %6 ]
  %j.0 = phi i32 [ 0, %3 ], [ %7, %6 ]
  %5 = icmp sle i32 %j.0, 4
  br il %5, label %6, label %9
  ; <label>:6
  %7 = add nsw i32 %j.0, 1
  %8 = add nsw i32 %k.1, 1
  br label %4
  ; <label>:9
  %10 = add nsw i32 %i.0, 1
  br label %1
  ; <label>:11
  ret i32 0
}
```

Listing A.7: The Motivating Example

---

Output of Range Analysis

---

for fvar X0:0
0x950cdc8----

Transition Start

Label -> 0x95058d0
Environment contains the following mappings (Size:0)

---

Transition End

---

for fvar X1:1
0x950cf10----
Transition Start

Label -> 0x9505bf0

Environment contains the following mappings (Size: 11)
I -> 0x9504a70, V -> [0.4] 0
I -> 0x9505890, V -> T 0
I -> 0x95059b8, V -> T 0
I -> 0x9505a20, V -> [0.0] 0
I -> 0x9505bf0, V -> T 0
I -> 0x9505da8, V -> T 0
I -> 0x9505df8, V -> [0.0] 0
I -> 0x9505e68, V -> T 0
I -> 0x9505fc8, V -> T 0
I -> 0x9506018, V -> [1.5] 1
I -> 0x95060f0, V -> [1.5] 1

Transition End

Transition Start

Label -> 0x9505c40

Environment contains the following mappings (Size: 11)
I -> 0x9504a70, V -> [5.5] 0
I -> 0x9505890, V -> T 0
I -> 0x95059b8, V -> T 0
I -> 0x9505a20, V -> [0.0] 0
I -> 0x9505bf0, V -> T 0
I -> 0x9505da8, V -> T 0
I -> 0x9505df8, V -> [0.0] 0
I -> 0x9505e68, V -> T 0
I -> 0x9505fc8, V -> T 0
I -> 0x9506018, V -> T 0
I -> 0x95060f0, V -> [1.5] 1
I -> 0x95060f0, V -> [1.5] 1

Transition End

for fvar X5:5
0x950d350 ----

Transition Start

Label -> 0x95055d0

Environment contains the following mappings (Size: 11)
I -> 0x9504a70, V -> [0.2147483647] 0
I -> 0x9505890, V -> T 0
I -> 0x95059b8, V -> T 0
I -> 0x9505a20, V -> [0.2147483648] 0
I -> 0x9505bf0, V -> T 0
I -> 0x9505da8, V -> T 0
I -> 0x9505df8, V -> [0.2147483648] 0
I -> 0x9505e68, V -> [5.5] 0
I -> 0x9505fc8, V -> [1.5] 1
I -> 0x9506018, V -> [1.2147483648] 1
I -> 0x95060f0, V -> [1.2147483648] 1
for fvar X2:2
0x950cf98

Transition Start

Label -> 0x9505ce8
Environment contains the following mappings (Size: 11)

I -> 0x950470, V -> [0, 4] 0
I -> 0x9505890, V -> T 0
I -> 0x9509b8, V -> T 0
I -> 0x9505a20, V -> [0, 0] 0
I -> 0x9503bf0, V -> T 0
I -> 0x9505da8, V -> T 0
I -> 0x9505df8, V -> [0, 0] 0
I -> 0x9505e68, V -> T 0
I -> 0x9505fc8, V -> T 1
I -> 0x9506018, V -> T 1
I -> 0x95060f0, V -> [1, 5] 1

Transition End

for fvar X6:6
0x950d418

Transition Start

Label -> 0x9505c40
Environment contains the following mappings (Size: 11)

I -> 0x950470, V -> [5, 5] 0
I -> 0x9505890, V -> T 0
I -> 0x95059b8, V -> T 0
I -> 0x9505a20, V -> [0, 0] 0
I -> 0x9503bf0, V -> T 0
I -> 0x9505da8, V -> T 0
I -> 0x9505df8, V -> [0, 0] 0
I -> 0x9505e68, V -> T 0
I -> 0x9505fc8, V -> T 1
I -> 0x9506018, V -> T 1
I -> 0x95060f0, V -> [1, 5] 1

Transition End

for fvar X3:3
0x950d130

Transition Start

Label -> 0x95059b8
Environment contains the following mappings (Size: 11)

I -> 0x950470, V -> [0.2147483647] 0
I -> 0x9505890, V -> T 0
Listing A.8: Results

I $\to$ 0x95059b8, V $\to$ T 0
I $\to$ 0x9505a20, V $\to$ [0.2147483648] 0
I $\to$ 0x9505bf0, V $\to$ T 0
I $\to$ 0x9505da8, V $\to$ T 0
I $\to$ 0x9505df8, V $\to$ [0.2147483648] 0
I $\to$ 0x9505e68, V $\to$ [5.5] 0
I $\to$ 0x9505fc8, V $\to$ [1.5] 1
I $\to$ 0x9506018, V $\to$ [1.2147483648] 1
I $\to$ 0x95060f0, V $\to$ [1.2147483648] 1

Environment contains the following mappings (Size: 11)
I $\to$ 0x9504a70, V $\to$ [0.2147483647] 0
I $\to$ 0x9505890, V $\to$ T 0
I $\to$ 0x95059b8, V $\to$ T 0
I $\to$ 0x9505a20, V $\to$ [0.2147483648] 0
I $\to$ 0x9505bf0, V $\to$ [0.4] 0
I $\to$ 0x9505fc8, V $\to$ [1.5] 1
I $\to$ 0x9506018, V $\to$ [1.2147483648] 1
I $\to$ 0x95060f0, V $\to$ [1.2147483648] 1

for fvar X4:4

0x950d240 ----

Environment contains the following mappings (Size: 11)
I $\to$ 0x9504a70, V $\to$ [0.2147483647] 0
I $\to$ 0x9505890, V $\to$ T 0
I $\to$ 0x95059b8, V $\to$ T 0
I $\to$ 0x9505a20, V $\to$ [0.2147483648] 0
I $\to$ 0x9505bf0, V $\to$ T 0
I $\to$ 0x9505da8, V $\to$ T 0
I $\to$ 0x9505df8, V $\to$ [0.2147483648] 0
I $\to$ 0x9505e68, V $\to$ [0.4] 0
I $\to$ 0x9505fc8, V $\to$ [1.5] 1
I $\to$ 0x9506018, V $\to$ [1.2147483649] 1
I $\to$ 0x95060f0, V $\to$ [1.2147483648] 1

End of Range Analysis
5. Symbolic Interval Program:

```
; ModuleID = '/symb1phi.bc'
:16:32"
target triple = "i386-pc-linux-gnu"
define i32 @f() nounwind {
  ret i32 1
}
define i32 @g(i32 %i) nounwind {
  %1 = add nsw i32 %i, 1
  ret i32 %i
}
define i32 @fun() nounwind {
  %1 = call i32 @f()
  %2 = call i32 @f()
  %3 = call i32 @g(i32 1)
  %4 = call i32 @g(i32 1)
  %5 = sub nsw i32 %1, %2
  %6 = add nsw i32 %5, 4
  %7 = add nsw i32 %5, %4
  %8 = call i32 @g(i32 1)
  %9 = mul nsw i32 2, %8
  %10 = sub nsw i32 %7, %9
  ret i32 %3
}

Listing A.9: Symbolic Interval Example

Output of Range Analysis
-------------------------------------
for fvar X0:0  
0xa3fbd90----

Transition Start

Label --> 0xa3f4658
Environment contains the following mappings(Size:0)
-------------------------------------

Transition End

-------------------------------------

for fvar X0:1  
0xa3fbc00----

Transition Start

Label --> 0xa3f3a28
Environment contains the following mappings(Size:2)
1 --> 0xa3f3bd8 , V --> T 0
1 --> 0xa3f4990 , V --> T 1
-------------------------------------
```
for fvar X0:2
0xa3fc228

Transition Start

Label -> 0xa3f4a18
Environment contains the following mappings (Size: 11)
I -> 0xa3f4a54, V -> [var64, var64] 1
I -> 0xa3f4a9c, V -> [var64, var64] 1
I -> 0xa3f4b08, V -> [var81, var81] 1
I -> 0xa3f4b58, V -> [var81, var81] 1
I -> 0xa3f4bd0, V -> [0, 0] 1
I -> 0xa3f4c50, V -> [4, 4] 1
I -> 0xa3f4cae0, V -> [2*var81, 2*var81] 1
I -> 0xa3f4cf0, V -> [var81, var81] 1
I -> 0xa3f4d70, V -> T 0
I -> 0xa3f4db8, V -> T 1
I -> 0xa3f4e08, V -> [0, 0] 1

Transition End

End of Range Analysis

Listing A.10: Results
6. Symbolic Interval Branch Program:

```c
ModuleID = ".t7phi.bc"

:16:32"

target triple = "i386-pc-linux-gnu"

define i32 @f() nounwind {
%1 = add nsw i32 undef, 4
%2 = add nsw i32 8, %1
ret i32 %2
}

define i32 @main() nounwind {
%1 = call i32 @f()
%2 = icmp sle i32 %1, 1
br il %2, label %3, label %5
%4 = mul nsw i32 %1, 2
br label %7
%6 = mul nsw i32 %1, 2
br label %7
ret i32 0
}
```

Listing A.11: Symbolic Interval Example

```
Output of Range Analysis

---
for fvar X0:0
0xa0dcb60---
--- Transitions Start ---
Label \( \rightarrow \) 0xa0d56a0
Environment contains the following mappings (Size: 3)
I \( \rightarrow \) 0xa0d56d0, V \( \rightarrow \) T 0
I \( \rightarrow \) 0xa0d57a8, V \( \rightarrow \) T 1
I \( \rightarrow \) 0xa0d57f8, V \( \rightarrow \) T 1
---
--- Transitions End ---
---
for fvar X0:1
0xa0dceda0---
--- Transitions Start ---
Label \( \rightarrow \) 0xa0d5a10
Environment contains the following mappings (Size: 1)
I \( \rightarrow \) 0xa0d593c, V \( \rightarrow \) [var32, min(var32, 1)] 1
```
For fvar X1:2
0xa0dce70----

Transition Start

Label -> 0xa0d5b38
Environment contains the following mappings (Size: 2)
I -> 0xa0d593c, V -> [max(var32, 2), var32] 1
I -> 0xa0d5bc0, V -> [2*max(var32, 2), var32] 1

Transition End

for fvar X2:3
0xa0dcf40----

Transition Start

Label -> 0xa0d5b38
Environment contains the following mappings (Size: 2)
I -> 0xa0d593c, V -> [min(var32, max(var32, 2)), max(min(var32, 1), var32)] 1
I -> 0xa0d5bc0, V -> [2*var32, 2*min(var32, 1)] 1

Transition End

for fvar X3:4
0xa0dd008----

Transition Start

Label -> 0xa0d5b38
Environment contains the following mappings (Size: 3)
I -> 0xa0d593c, V -> [min(var32, max(var32, 2)), max(min(var32, 1), var32)] 1
I -> 0xa0d5bc0, V -> [2*var32, 2*min(var32, 1)] 1
I -> 0xa0d5bc0, V -> [2*max(var32, 2), 2*var32] 1

Transition End

End of Range Analysis
Appendix B

Abstract Interpretation Proofs

This appendix presents the proofs of the Galois connection and soundness proofs for the concrete and abstract semantics from Chapter 3. Both the Galois connection and soundness is implicitly proven for the symbolic interval domain through the discrete interval domain. From here on, we will refer to the discrete interval domain simply as the interval domain.

B.1 Monotonicity

We first prove the property of monotonicity of each semantic function. For each semantic function we prove that it preserves the partial ordering of the domain.

B.1.1 Monotonicity of statements

Theorem 1. For all $\sigma_1, \sigma_2 \in Env^d$, $\sigma_1 \subseteq \sigma_2 \rightarrow f(\sigma_1) \subseteq f(\sigma_2)$ where $f = S^d[Stmt]$.

Proof: By Lemma 2 and 3 all cases of Stmt are proven. □

Lemma 2. $\sigma_1 \subseteq \sigma_2 \rightarrow f(\sigma_1) \subseteq f(\sigma_2)$ where $f = S^d[id:=\phi(varlist)]$

Proof: Because phi is essentially a meet of all vars this prove is trivial as we can assume the meet of variables is monotonic. □

Lemma 3. $\sigma_1 \subseteq \sigma_2 \rightarrow f(\sigma_1) \subseteq f(\sigma_2)$ where $f = S^d[id:=Expr]$

Proof: By Lemma 4 Expr is monotonic, so substituting expr id in any two environments will produce new environments such that they are monotonic as well.

so, $[id \mapsto f(\sigma_1)]/\sigma_1 = \sigma'_1$

and $[id \mapsto f(\sigma_2)]/\sigma_2 = \sigma'_2$

$\therefore \sigma'_1 \subseteq \sigma'_2$ □

Lemma 4. $\sigma_1 \subseteq \sigma_2 \rightarrow f(\sigma_1) \subseteq f(\sigma_2)$ where $f = E^d[Expr]$

Proof: We prove this lemma by structural induction.

IH: $\forall \sigma_1, \sigma_2 : Env^d. \sigma_1 \subseteq \sigma_2 \rightarrow f(\sigma_1) \subseteq f(\sigma_2)$ where $f = E^d[Expr]$

Base case:

1. Const: $\sigma_1 \subseteq \sigma_2 \rightarrow f(\sigma_1) \subseteq f(\sigma_2)$ where $f = E^d[Const]$
Proof: - Assume $\sigma_1 \subseteq \sigma_2$
- We apply $f_{const}$ to both environments - $f_{const}(\sigma_1) \subseteq f_{const}(\sigma_2)$
- By definition of $f_{const}$
- $[Const, Const] \subseteq [Const, Const]$ □

2. var: $\sigma_1 \subseteq \sigma_2 \rightarrow f(\sigma_1) \subseteq f(\sigma_2)$ where $f = E_d[\var]

Proof: - Assume $\sigma_1 \subseteq \sigma_2$
- We apply $f_{ar}$ to both environments - $f_{ar}(\sigma_1) \subseteq f_{ar}(\sigma_2)$
- By definition of $f_{ar}$
- $[\sigma_1(var), \sigma_1(var)] \subseteq [\sigma_2(var), \sigma_2(var)]$
- By assumption var in $\sigma_1 \subseteq$ var in $\sigma_2$ □

Inductive step: $\sigma_1 \subseteq \sigma_2 \rightarrow f(\sigma_1) \subseteq f(\sigma_2)$ where $f = E_d[Expr BinOp Expr]$

Proof: - By IH and Lemma 5 $\sigma_1 \subseteq \sigma_2 \rightarrow f(\sigma_1) \subseteq f(\sigma_2)$ where $f = E_d[Expr BinOp Expr]$ □

Lemma 5. $\sigma_1 \subseteq \sigma_2 \rightarrow f(\sigma_1) \subseteq f(\sigma_2)$ where $f = R_d[binop]$

- Case addition: $[a, b] \subseteq [c, d] \rightarrow [a, b] + [x, y] \subseteq [c, d] + [x, y]$

Proof: - Add $[x, y]$ to both intervals
$[a, b] + [x, y]$
$[c, d] + [x, y]$
- By interval arithmetic
$[a + x, b + y]$
$[c + x, d + y]$
- As $a \leq c$ and $b \geq d$ then
$a + x \leq c + x$ and $b + y \geq d + y$
- By the rules of intervals
$[a, b] + [x, y] \subseteq [c, d] + [x, y]$ □

- Case subtraction: Same as addition

- Case multiplication: $[a, b] \subseteq [c, d] \rightarrow [a, b] \ast [x, y] \subseteq [c, d] \ast [x, y]$

Proof: - Add $[x, y]$ to both intervals
$[a, b] \ast [x, y]$
$[c, d] \ast [x, y]$
- By interval arithmetic
$[\min(a \ast x, a \ast y, b \ast x, b \ast y), \max(a \ast x, a \ast y, b \ast x, b \ast y)]$
$[\min(c \ast x, c \ast y, d \ast x, d \ast y), \max(c \ast x, c \ast y, d \ast x, d \ast y)]$
- As $a \leq c$ and $b \geq d$ then
$\min(a \ast x, a \ast y, b \ast x, b \ast y) \leq \min(c \ast x, c \ast y, d \ast x, d \ast y)$ and $\max(a \ast x, a \ast y, b \ast x, b \ast y) \geq$
$\max(c \ast x, c \ast y, d \ast x, d \ast y)$
- By the rules of intervals
$[a, b] \ast [x, y] \subseteq [c, d] \ast [x, y]$ □

- Case division: same as multiplication
B.2 Galois Connection

We define the alpha function that maps the concrete to the abstract domain. See Chapter 2 for a discussion on its significance.

Definition (Interval α) B.2.1. Let
\[ \alpha(\emptyset) = \bot \]
\[ \alpha(X) = [\min(X), \max(X)] \]

Additionally we define the gamma function that maps the abstract to the concrete domain.

Definition (Interval γ) B.2.1. Let
\[ \gamma(\bot) = \emptyset \]
\[ \gamma(Y) = \{ y | \min(Y) \leq y \leq \max(Y) \} \]

We define the alpha function for environments as follows:

Definition (Interval Environment α) B.2.1. \[ \alpha(\sigma) = \{ (v, [x, x]) : (v, x) \in \sigma \} \]

Additionally, we define the gamma function for environments as follows:

Definition (Interval Environment γ) B.2.1. \[ \gamma(\sigma^\flat) = \{ (v, [x, x]) : (v, y) \in \sigma^\flat \} \]

B.2.1 Monotonicity of γ and α

The following proofs, prove that the gamma and alpha functions are monotone.

- Monotonicity of α

Theorem 1. \( \forall c_1, c_2 \in \varphi(\mathbb{Z}) : c_1 \subseteq c_2 \rightarrow \alpha(c_1) \subseteq \alpha(c_2) \)

Proof: Assume \( c_1 \subseteq c_2 \)
- Assumption
  \( c_1 \subseteq c_2 \)
- Definition of \( \subseteq \)
  \( \min(c_1) \leq \min(c_2) \) and \( \max(c_1) \geq \max(c_2) \)
- Definition of \( \subseteq \)
  \( [\min(c_1), \max(c_1)] \subseteq [\min(c_2), \max(c_2)] \)
- Definition of \( \alpha \)
  \( \alpha(c_1) \subseteq \alpha(c_2) \)

\[ \therefore \forall c_1, c_2 \in \varphi(\mathbb{Z}) : c_1 \subseteq c_2 \rightarrow \alpha(c_1) \subseteq \alpha(c_2) \]

- Monotonicity of γ

Theorem 2. \( \forall [l_1, u_1], [l_2, u_2] \in \mathbf{I}(\mathbb{Z}) : [l_1, u_1] \subseteq [l_2, u_2] \rightarrow \alpha([l_1, u_1]) \subseteq \alpha([l_2, u_2]) \)

Proof: Assume \( [l_1, u_1] \subseteq [l_2, u_2] \)
- Definition of \( \subseteq \)
  \( l_1 \leq l_2 \) and \( u_1 \geq u_2 \)
- Definition of \( \varphi(\mathbb{Z}) \)
\[ \{ x | l_1 \leq x \leq u_1 \} \subseteq \{ x | l_2 \leq x \leq u_2 \} \]
- Definition of \( \gamma \)
\( \gamma([l_1, u_1]) \subseteq \gamma([l_2, u_2]) \)
\[ \therefore \forall [l_1, u_1], [l_2, u_2] \in \varphi(\mathbb{Z}) : [l_1, u_1] \subseteq [l_2, u_2] \rightarrow \gamma([l_1, u_1]) \subseteq \gamma([l_2, u_2]) \]

**B.2.2 Galois connection for intervals**

Through the definitions of alpha and gamma we prove a Galois connection between the domain.

**Theorem 1.** \( \alpha(c) \sqsubseteq c' \leftrightarrow c \subseteq \gamma(c') \)

*Proof.* By 2 and 4 \( \square \)

**Lemma 2.** \( \alpha(c) \sqsubseteq c' \leftarrow c \subseteq \gamma(c') \)

*Proof.* Assume \( \alpha(c) \sqsubseteq c' \)
\( c \subseteq \gamma(c') \)
- By monotonicity if \( \alpha \)
\( \alpha(c) \sqsubseteq \alpha(\gamma(c')) \)
- By Lemma 3
\( \alpha(c) \sqsubseteq c' \)
- Assumption \( \square \)

**Lemma 3.** \( \alpha(\gamma([l, u]))) \leftrightarrow [l, u] \)

*Proof.* \( \alpha(\gamma([l, u]))) \)
- By definition of \( \gamma \)
\( \alpha([x | l \leq x \leq u]) \)
- By definition of \( \alpha \)
\([l, u] \)
\( \square \)

**Lemma 4.** \( c \subseteq \gamma(c') \leftarrow \alpha(c) \sqsubseteq c' \)

*Proof.* Assume \( c \subseteq \gamma(c') \)
\( \alpha(c) \sqsubseteq c' \)
- By monotonicity if \( \gamma \)
\( \gamma(\alpha(c)) \subseteq \gamma(c') \)
- By definition of \( \alpha \)
\( \gamma([c, c]) \subseteq \gamma(c') \)
- By definition of \( \gamma \)
\( c \subseteq \gamma(c') \)
- Assumption \( \square \)

**B.3 Soundness proof**

We prove the soundness of the Galois connection between the abstract and concrete semantics

1. \( \forall \text{ initial states } I. \alpha(I) \sqsubseteq I^\sharp \)
2. ∀ valuation functions \( Y.a(Y) \sqsubseteq Y^\# \)

Because of the monotonicity of \( \gamma \) we can prove from \( 1 \) and \( 2 \)

1. ∀ initial states \( I.\gamma(I) \sqsubseteq I^\# \)

2. ∀ semantic functions \( Y.\gamma(Y \sigma^\#) \sqsubseteq Y \gamma(\sigma^\#) \)

### B.3.1 Initial State

**Lemma 1.** Galois connection for Initial states, ∀ initial states \( I \). \( \alpha(I) \sqsubseteq I^\# \)

**Proof.** Let \( I \) be the environment where all variables map to the empty set \( \emptyset \).
Let \( I^\# \) be the environment where all variables map to \( \top \).
By definition of \( \alpha \) on environments it is true that ∀ \((\text{var}, i) \in I\).
\( \text{var} \mapsto \top \)

\[ I^\# \sqsubseteq I^\# \]

\[ \square \]

### B.3.2 Soundness of Semantic Functions

**Lemma 1.** Galois connection between \( E[\text{expr}] \) and \( E^\#[\text{expr}] \)

**Proof.** Proof: By structural induction.

IH: \( \forall \sigma \in \text{Env}. \forall \sigma \in \text{Expr}, \alpha(E[\text{expr}]\sigma) \sqsubseteq E^\#[\text{expr}]\alpha(\sigma) \)

Base Case:

1. Constant Expressions
   \( \forall \sigma \in \text{Env}. \forall \sigma \in \text{Const} \sqsubseteq E^\#[\text{Const}]\alpha(\sigma) \)
   - By definition of \( E[\text{Const}] \)
   \( \forall \sigma \in \text{Env}. \forall \sigma \in \text{Const} \sqsubseteq E^\#[\text{Const}]\alpha(\sigma) \)
   - By definition of \( \alpha \)
   \( \forall \sigma \in \text{Env}. \forall \sigma \in \text{Const} \sqsubseteq E^\#[\text{Const}]\alpha(\sigma) \)
   - By definition of \( E^\#[\text{Const}] \)
   \( [\min(\text{Const}), \max(\text{Const})] \sqsubseteq [\min(\text{Const}), \max(\text{Const})] \)
   \[ \square \]

2. Variable Expressions
   \( \alpha(E[\text{var}]\sigma) \sqsubseteq E^\#[\text{var}]\alpha(\sigma) \)
   - By definition of \( E[\text{var}] \)
   \( \alpha(\sigma(\text{var})) \sqsubseteq E^\#[\text{var}]\alpha(\sigma) \)
   - By definition of \( \alpha \)
   \( \sigma(\text{var}), \sigma(\text{var}) \sqsubseteq E^\#[\text{var}]\sigma^\# \)
   - By definition of \( E^\#[\text{var}] \)
   \( \sigma(\text{var}), \sigma(\text{var}) \sqsubseteq [\sigma(\text{var}), \sigma(\text{var})] \)
   \[ \square \]

Inductive Step:

1. Binop Expressions \( \alpha(E[\text{expr binop expr}]\sigma) \sqsubseteq E^\#[\text{expr binop expr}]\sigma \)
   - Assume IH.
- By arithmetic rules
  \[ E[expr] \text{binop} E[expr] \] can be reduced to \[ E[expr] \] and
  \[ E[expr] \text{binop} E[expr] \] can be reduced to \[ E[expr] \].
- By assumption of IH T

2. Unop Expressions \[ a(E[\text{binop expr}](\sigma)) \subseteq E[\text{unop expr}]a(\sigma) \]
   Proof:
   - Assume IH.
   - By arithmetic rules
     \[ E[expr] \text{unop} \] can be reduced to \[ E[expr] \]. and
     \[ E[expr] \text{unop} \] can be reduced to \[ E[expr] \].
   - By assumption IH T

\[ \therefore \forall \sigma \in \text{Env}. \forall \sigma \in \text{Expr}. a(E[expr]a(\sigma) = T \]

**B.3.3 Statements**

**Lemma 1.** Galois connection between \( S[Stmt] \) and \( S[Stmt] \)

**Proof.**

Case Assignment:
\[ \forall \sigma \in \text{Env}. a(S[\text{id:= expr}](\sigma)) \subseteq S[\text{id:= expr}]a(\sigma) \]
- By definition of \( S[\text{id:= expr}] \)
\[ a((id \mapsto E[expr](\sigma)/\sigma)) \subseteq E[\text{id:= expr}]a(\sigma) \]
- By substitution
\[ a(\sigma') \subseteq E[\text{id:= expr}]a(\sigma) \]
- By definition of \( a \)
\[ \sigma'' \subseteq E[\text{id:= expr}]\sigma' \]
- By definition of \( E[\text{id:= expr}] \)
\[ \sigma'' \subseteq (id \mapsto E[\text{expr}]\sigma in (\sigma')) \]
- By substitution
\[ \sigma'' \subseteq \sigma' \]

**B.3.4 Statement list**

**Lemma 1.** Galois connection between \( S'[Stmtlist] \) and \( S'[Stmtlist] \)

**Proof.** We prove the Galois connection on lists of statements by structural induction.
IH: \( \forall \sigma \in \text{Env}. S'[Stmtlist](\sigma) \)

- Base Case
  \[ a(S'[Stmt](\sigma)) \subseteq S'[Stmt]a(\sigma) \]
  - By lemma [1]

- Inductive Case
  \[ a(S'[Stmt; Stmtlist](\sigma)) \subseteq S'[Stmt; Stmtlist]a(\sigma) \]
- By composition elimination $\alpha(\text{S} \llbracket \text{stmtlist} \rrbracket)(\text{S} \llbracket \text{stmt} \rrbracket(\sigma)) \subseteq (\text{S} \llbracket \text{stmtlist} \rrbracket)\text{S} \llbracket \text{stmt} \rrbracket \alpha(\sigma)$
- By lemma $\alpha(\text{S} \llbracket \text{stmtlist} \rrbracket) \subseteq (\text{S} \llbracket \text{stmtlist} \rrbracket)\alpha(\sigma)$
- By IH
## Notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>16</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>16</td>
</tr>
<tr>
<td>$\cup$</td>
<td>21</td>
</tr>
<tr>
<td>$L$</td>
<td>14</td>
</tr>
<tr>
<td>$\cap$</td>
<td>34</td>
</tr>
<tr>
<td>$\sqcap$</td>
<td>15</td>
</tr>
<tr>
<td>$\sqcup$</td>
<td>15</td>
</tr>
<tr>
<td>$\vee$</td>
<td>17</td>
</tr>
<tr>
<td>$\Delta$</td>
<td>17</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>15</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>$\bot$</td>
<td>15</td>
</tr>
<tr>
<td>$\top$</td>
<td>15</td>
</tr>
<tr>
<td>$\sqsubseteq$</td>
<td>14</td>
</tr>
<tr>
<td>$\cdot$</td>
<td>19</td>
</tr>
<tr>
<td>$\psi$</td>
<td>39</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>38</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>38</td>
</tr>
<tr>
<td>$::= bnf$</td>
<td>27</td>
</tr>
<tr>
<td>$\phi$</td>
<td>27</td>
</tr>
<tr>
<td>$\rho$</td>
<td>33</td>
</tr>
<tr>
<td>$\star$</td>
<td>35</td>
</tr>
<tr>
<td>$N$</td>
<td>34</td>
</tr>
<tr>
<td>$\zeta$</td>
<td>29</td>
</tr>
<tr>
<td>$\xi$</td>
<td>29</td>
</tr>
</tbody>
</table>
Bibliography


[12] Patrick Cousot and Radhina Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the*


