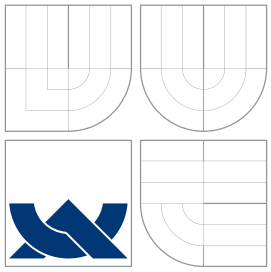
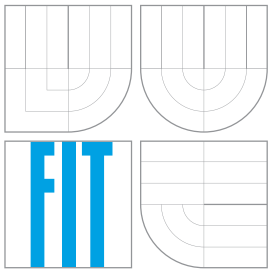


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

STATICKÁ ANALÝZA MOŽNÝCH HODNOT PROMĚNNÝCH V PROGRAMECH V C

STATIC VALUE ANALYSIS OVER C PROGRAMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. DANIELA ĎURIČEKOVÁ

VEDOUCÍ PRÁCE
SUPERVISOR

prof. Ing. TOMÁŠ VOJNAR, Ph.D.

BRNO 2013

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2012/2013

Zadání diplomové práce

Řešitel: **Ďuričeková Daniela, Bc.**
Obor: Inteligentní systémy
Téma: **Statická analýza možných hodnot proměnných v programech v C**
Static Value Analysis over C Programs
Kategorie: Formální verifikace

Pokyny:

1. Nastudujte teorii analýzy toku dat (data flow analysis), zejména pak analýzu možných hodnot proměnných (value analysis).
2. Seznamte se s prostředím Code Listener vyvíjeným na FIT VUT jako nadstavba gcc usnadňující tvorbu statických analýz.
3. Navrhněte analýzu možných hodnot proměnných v programech v C s využitím prostředí Code Listener.
4. Navrženou analýzu implementujte a otestujte na vhodných programech.
5. Shrňte získané zkušenosti a možnosti dalšího rozvoje vytvořeného nástroje.

Literatura:

- Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, Springer-Verlag, 2005.
- Aho, A.V., Lam, S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, Addison Wesley, 2006.
- Khedker, U.P., Sanyal, A., Karkare, B.: Data Flow Analysis: Theory and Practice, CRC Press, 2009.
- Cuoq, P., Prevosto, V.: Frama-C's Value Analysis Plug-in, CEA LIST, Software Reliability Laboratory, Saclay, France, 2011. http://frama-c.com/download/frama-c-value-analysis.pdf
- Dudka, K., Peringer, P., Vojnar, T.: An Easy to Use Infrastructure for Building Static Analysis Tools, In: Proc. of EUROCAST'11, Las Palmas, Spain, 2011. http://www.fit.vutbr.cz/research/groups/verifit/tools/code-listener/

Při obhajobě semestrální části diplomového projektu je požadováno:

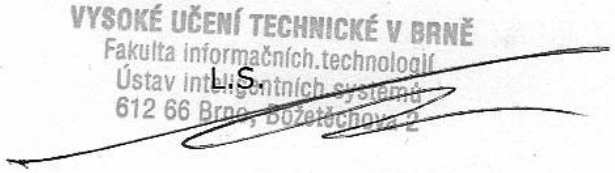
- Bez požadavků.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Vojnar Tomáš, prof. Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 17. září 2012
Datum odevzdání: 22. května 2013



doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Static Value Analysis over C Programs

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně pod vedením prof. Tomáše Vojnara. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Daniela Ďuričeková
May 18, 2013

Poděkování

Na tomto místě bych ráda poděkovala mému vedoucímu prof. Tomáši Vojnarovi za odborné vedení, za poskytnutou literaturu a za ochotu a čas, který mi při tvorbě práce věnoval. Také bych chtěla poděkovat Ing. Kamilu Dudkovi za jeho rady týkající se prostředí Code Listener, které bylo využito k implementaci analyzátoru. V neposlední řadě bych ráda poděkovala svému příteli za podporu během práce a závěrečnou kontrolu angličtiny.

Abstrakt

Analýza rozsahu hodnot (anglicky value-range analysis) je metoda statické analýzy založená na zjišťování hodnot, kterých může daná proměnná nabývat v určitém místě v programu. Tato technika může být použita k dokázání, že se v programu nevyskytují chyby za běhu, jako například přístup za hranici pole. Jelikož analýza rozsahu hodnot získává informace o každém místě v programu, lze k její implementaci využít analýzu toku dat (anglicky data-flow analysis). Cílem této diplomové práce je návrh a implementace funkčního nástroje provádějícího analýzu rozsahu hodnot. Práce začíná úvodem do problematiky, vysvětlením analýz toku dat a hodnot proměnných a popisem abstraktní interpretace, která tvoří formální základ analyzátoru. Následuje seznámení s prostředím Code Listener, které bylo využito k implementaci analyzátoru. Jádro práce tvoří návrh, implementace a otestování analyzátoru. V závěru jsou shrnuty nabyté zkušenosti a diskutovány možnosti budoucího vývoje vytvořeného nástroje.

Abstract

Value-range analysis is a static analysis technique based on arguing about the values that a variable may take on a given program point. It can be used to prove absence of run-time errors such as out-of-bound array accesses. Since value-range analysis collects information on each program point, data-flow analysis can be used in association with it. The main goal of this work is designing and implementing such a value-range analysis tool. The work begins with an introduction into the topic, an explanation of data-flow and value-range analyses and a description of abstract interpretation, which provides the formal basis of the analyser. The core of this work is the design, implementation, testing and evaluation of the analyser. In the conclusion, our personal experience obtained in the area of the thesis is mentioned, along with a discussion of a possible future development of the designed tool.

Klíčová slova

analýza toku dat, analýza rozsahu hodnot, abstraktní interpretace, Code Listener, intraprocedurální analýza, graf toku řízení, částečně uspořádaná množina, svaz

Keywords

data-flow analysis, value-range analysis, abstract interpretation, intraprocedural analysis, Code Listener, control-flow graph, partially ordered set, lattice

Citace

Daniela Ďuričeková: Static Value Analysis over C Programs, diplomová práce, Brno, FIT VUT v Brně, 2013

Contents

Chapter 1

Introduction

Arguably, one of the most well-known type of software vulnerabilities is a situation called buffer overflow. A buffer overflow occurs when data are written into a memory buffer that is not large enough to store these data. Buffer overflows may be exploited by a malicious person to gain control over a computer system. For example, in November 1988, an infamous Morris worm infected approximately 6000 network-connected hosts which represented 5–10 % of the Internet at that time [7]. One of the primary replication mechanisms of the Morris worm was based on exploiting a buffer overflow in the `fingerd` daemon. However, in many cases, buffer-overflow vulnerabilities do not need to be exploited by malicious persons to have disastrous consequences. Indeed, probably the best-known case of a buffer overflow is the Ariane 5 failure from 1996 where the catastrophe was caused by a program trying to store a 64-bit number into a 16-bit space [35].

Since both mentioned cases took place more than a decade ago, one might think that at present, buffer overflows are no longer an issue because programmers are well aware of them. However, the opposite is true. In the graph from Figure 1.1, the number of buffer-overflow-related errors in recent years is shown. Data for this graph are obtained from [40]. Of course, only reported errors are included. From this graph, it is obvious that the number of buffer overflow vulnerabilities is increasing.

Buffer overflows and other run-time errors, especially in software-critical systems, may cause not only a loss of huge amounts of money but even worse—a loss of human lives. So, a need for a precise verification of the systems before their usage is consistently increasing. This need caused an emergence of formal verification methods. In summary, formal verification is the use of rigorous methods to ensure that a system conforms to some precisely expressed notion of functional correctness. There are several methods of formal verification and each of them is based on a different mathematical apparatus. Model checking, abstract interpretation, theorem proving and static analysis belong to the best known representatives of these methods [34].

To prove the absence of buffer overflows and other run-time errors, an analysis called *value-range analysis* in collaboration with data-flow analysis can be used. Both of them belong to static analysis techniques. Value-range analysis is based on arguing about the values that a variable may take on a given program point. For example, it may tell us that a variable `i` in the statement `a[i] = x;` can be from the interval $\langle 0, 10 \rangle$.

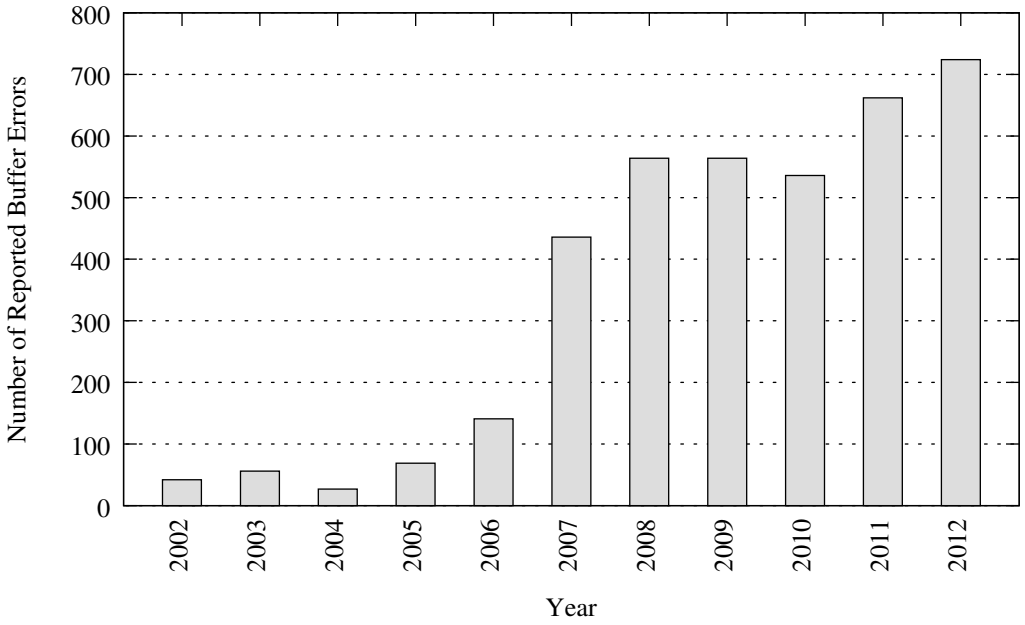


Figure 1.1: Number of reported buffer-overflow-related errors in recent years

The goal of this work is to design and implement an intraprocedural value-range analyser built on top of the Code Listener infrastructure (see [17]). Code Listener is a completely open-source infrastructure intended to simplify construction of tools for static analysis of C programs. At present, Code Listener provides a unified, object-oriented and easy to use Application Programming Interface (API) over the GCC front-end (see [51]) as well as over the Sparse front-end (see [53]). Our value-range analyser receives the intermediate source code representation from Code Listener and computes ranges of variables used in the analysed source code. Presently, it is able to handle all variables of primitive data types in the C language, namely `char`, `short`, `int`, `long`, `float`, `double` and `long double`. Moreover, our analyser is able to calculate ranges for items in structures and statically allocated arrays. However, at present, it does not manage anything stored on the heap (dynamically allocated by the `malloc()` function or other related functions, such as `calloc()`). If the analyser is used on programs that fulfill the above conditions, it provides a sound analysis, meaning that all computed ranges are safe over-approximation of the real ones.

There exist several approaches to value-range analysis. From all of them, abstract interpretation was chosen for our analyser because probably the most successful value-range analyser deployed in industry, namely the Value Analysis plug-in [15] from the Frama-C platform, is based on this approach. Informally, abstract interpretation is a static analysis technique that executes analysed programs in an abstract way related to the concrete semantics via links based on the theory of Galois connections [13]. During this execution, abstraction is used to preserve only important program properties and abstracts away all irrelevant details. This is done to speedup the execution and make it converge on infinite data domains, thus making the analysis computationally feasible.

We believe that our analyser will be useful for detecting buffer overflows and related errors in critical software systems.

Chapter Survey

This work is divided into eight chapters. In Chapter 2, data-flow analysis is introduced, a mathematical background needed for constructing a general data-flow framework is studied, intraprocedural data-flow analysis is explained, and the so-called work list algorithm is presented. Chapter 3 deals with value-range analysis. A motivating example for using a value-range analysis is presented in there. Moreover, different approaches to value-range analysis with the emphasize on abstract interpretation and tools implementing value-range analysis are also introduced. The Code Listener infrastructure is described in Chapter 4. It presents the intermediate source code representation used by Code Listener, its architecture and the provided API. Chapter 5 explains the proposed design of our value-range analyser. This is followed by a description of its implementation in Chapter 6. Then, in Chapter 7, the testing process is described and the tool is evaluated. Finally, Chapter 8 summarizes the work and outlines its possible further development. In Appendix A, a complete example of an analysis by using the developed tool is given. Appendix B lists the contents of the enclosed CD.

Chapter 2

Data-Flow Analysis

This chapter provides a brief introduction to data-flow analysis. The fundamentals of data-flow analysis are provided in Section 2.1. In Section 2.2, two simplified examples of data-flow analyses are given, namely live variables analysis and available expressions analysis. Section 2.3 deals with the mathematical background needed for developing a general data-flow framework. For example, partially ordered sets and different kinds of lattices are studied there. Finally, Section 2.4 describes intraprocedural data-flow analysis. This chapter is based on [3, 29, 33].

2.1 Introduction to Data-Flow Analysis

Data-flow analysis is a static analysis technique that is used to discover relevant properties for each program point in the program being analysed. Unlike dynamic analyses, static analyses, including data-flow analysis, are performed without direct program execution. Therefore, static analysis techniques are appropriate if the run-time overhead is a matter of concern. Moreover, when directly executing a program, there is no guarantee of covering all its behaviour. Information discovered during data-flow analysis represents the run-time behaviour of a program.

Data-flow analysis has found many useful applications. Originally, it was introduced in the context of transformations performed by compilers that are aimed at optimizing the programs for space, time or power consumption. Usage in optimizations still remains its most dominant application. Another example of application of data-flow analysis is software verification used for determining the validity of programs with respect to some desired properties of interest. Information discovered during analysis is also helpful for understanding the behaviour of a program that can be used for debugging, testing and maintenance. Finally, reverse engineering also utilizes a data-flow analysis approach.

The following part briefly describes the most relevant properties of data-flow analysis, like flow sensitivity, context sensitivity, program representation and the like.

Types of Data-Flow Analysis

Data-flow analysis can be performed on different types of program representations, such as control-flow graphs (CFGs), abstract syntax trees (ASTs), program-flow graphs (PFGs), call graphs (CGs), program dependence graphs (PDGs) and static single assignment (SSA) forms. CFGs represent a typical internal program representation used in textbooks on data-flow analysis for the purposes of clarification. Since CFGs are also utilized in our examples, a more detailed explanation is provided later.

The most common representations of data-flow information are sets whose elements can be program entities that satisfy the given constraints (e.g. variables whose values are within certain ranges), or program states that satisfy the given formulae or facts that hold at a given program point. The sets of variables or expressions are typical for most familiar analyses and these sets tend to be implemented by bit vectors. Another approach to represent data-flow information is to use access paths.

With respect to granularity, there are two versions of data-flow analysis, namely exhaustive analysis and incremental analysis. An exhaustive analysis starts discovering information from scratch, whereas incremental analysis must be preceded by the corresponding exhaustive analysis in the initialization part, and after that, it updates the information derived in the previous step whenever there is a change in the code.

Another property of data-flow analysis is its sensitivity. It comes in two different flavors, specifically flow sensitivity and context sensitivity. The analysis is flow-sensitive if it takes into account the control flow of the program. Otherwise, it is a flow-insensitive analysis. A context-sensitive analysis distinguishes between different calling contexts of a function. So, context-sensitive analysis ensures that information discovered by the analysis could vary from one calling context of the function to another. Otherwise, it is a context-insensitive analysis.

With regard to the scope of data-flow analysis, three kinds of data-flow analyses are recognized:

- local data-flow analysis,
- global (intraprocedural) data-flow analysis,
- interprocedural data-flow analysis.

Local data-flow analysis is performed within a basic block. Basic blocks are covered in the following part, but for now, we only need to know that a basic block is a maximal group of consecutive statements that are always executed together with a rigorously sequential control flow between them. Intraprocedural analysis is performed across basic blocks restricted to a function or procedure and interprocedural analysis is accomplished across functions and procedures. A more detailed description of intraprocedural analysis is provided in Section 2.4.

Basic Blocks and Control-Flow Graphs

As mentioned in the previous part, a control-flow graph (CFG) is a frequently used program representation on which data-flow analysis is performed. It is a graph representation that

consists of basic blocks and edges between these blocks. A basic block is a maximal sequence of consecutive statements[1]. The first instruction in the basic block serves as the only entry point into that block. Similarly, the last instruction of the basic block is the only exit point from the block. Thus, there are no jumps into the middle of the block or from the middle of the block.

Given an input program, its CFG can be constructed in two steps. Firstly, the source code is partitioned into basic blocks. This partitioning process starts with the first statement of the given source code and consecutive statements are added until a jump or label, designating the end of a jump, is reached. Thus, the first basic block is created. In a similar way, the remaining basic blocks are generated. Secondly, the set of basic blocks are connected by edges that represent the control flow. It means that there is an edge from a basic block B1 to a block B2 if the first statement from block B2 can immediately follow the last statement in block B1. This is true if there is a jump from the last statement of block B1 to the first statement of block B2 or if the first statement of block B2 follows the last statement of block B1 in the given source code. In simple terms, the B1 basic block is a predecessor of the B2 block, and the B2 block is a successor of the B1 block. In Figure 2.1, a fragment of a source code with its corresponding CFG can be seen.

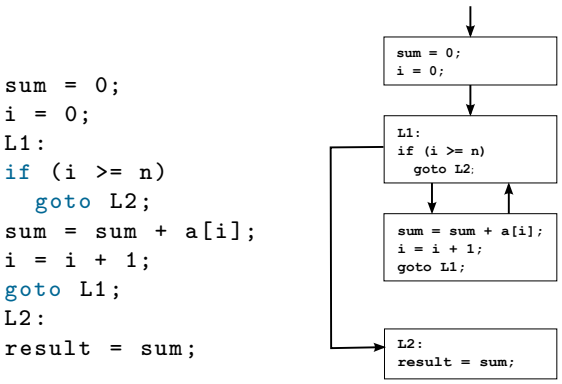


Figure 2.1: A fragment of a source code and its corresponding CFG

Therefore, a CFG is an oriented graph where nodes are basic blocks, and edges follow the transfer of control. For the purposes of data-flow analysis, in this text, it is assumed that two unique blocks, called *Start* and *End*, are in every CFG. The Start block is the first block of the CFG and it has no predecessors whereas the End block is the last block of a CFG and has no successors. Without any loss of generality, these blocks can be added if they do not exist.

Data-Flow Analysis Scheme

A data-flow analysis scheme is constructed in two granularity levels, from a simpler level to a more complex one but also a more effective one. In the first level, two data-flow variables are associated to each statement whereas in the second level, two data-flow variables are associated to a block of statements. In the following text, a more precise description of both granularity levels is provided.

[1]In compiler terminology, three-address instructions are used instead of statements. For simplicity, statements are used in this text.

Let us start with a clarification of the term data-flow variable. A data-flow variable represents an abstraction of the set of all possible program states that are observed at the given program point. For the reasons of efficiency, only relevant track of information is kept. In the first level of granularity, there are two data-flow variables for each statement s in the given source code. The first data-flow variable is associated with the point before statement s and it is denoted by In_s. The second data-flow variable is associated with the point immediately after statement s and it is labeled as Out_s. The goal of data-flow analysis is to find a solution to a set of constraints that satisfies In_s and Out_s for all statements s. For the subsequent explanation, consider an arbitrary statement s. In the first place, data-flow variables In_s and Out_s belonging to statement s are influenced by the semantics of s. The term transfer function, denoted by f_s, is often used to represent relationship between variables In_s and Out_s based on the semantics of the statement s. Since in data-flow analysis, two types of direction with respect to control flow are recognized (forward and backward analysis), the transfer function f_s can take two different forms. If the information is propagated forward along execution paths, the function f_s takes the data-flow variable before statement s as a parameter, and the output is a new data-flow variable immediately after the following statement:

$$Out_s = f_s(In_s).$$

In case that information is propagated backwards up the execution paths, the f_s function takes a data-flow variable after statement s as a parameter, and the output is a new data-flow variable before s:

$$In_s = f_s(Out_s).$$

In the second place, data-flow variables are influenced by the control-flow constraints. This makes sense, for example, when dealing with the second level of granularity whose explanation is provided in a few lines further. As shown in this section, by data-flow analysis, a constraint resolution system based on equalities is used to find a solution to a set of constraints on In_s and Out_s. This system consists of a constraints storage and a logic for solving constraints. These constraints are also known as data-flow equations.

Let us continue with the second granularity level that is aimed to save space and time by using basic blocks. In this case, the first data-flow variable is associated with the point before a basic block b and it is denoted by In_b. The second data-flow variable is associated with the point immediately after basic block b and it is denoted by Out_b. Therefore, it is necessary to reformulate the previous schema in terms of data-flow variables entering and leaving basic blocks. The data-flow equations of block b are derived from data-flow equations involving variables In_s and Out_s for all statements s in the given basic block b. Without any loss of generality, we can suppose that b consists of n statements $s_1, s_2, \ldots, s_n$ in that order, where s_1 is the first statement in block b and s_n is the last statement in block b. And so, it must hold that $In_b = In_{s_1}$ and $Out_b = Out_{s_n}$. Let f_{s_i} be the transfer function for statement s_i, where $1 \leq i \leq n$. The transfer function for block b is constructed by composing the transfer functions of the individual statements:

$$f_b = f_{s_n} \circ f_{s_{n-1}} \circ \cdots \circ f_{s_1}.$$

Taking into account a CFG, its basic blocks influence each other according to the edges in the CFG. This influence depends on the direction of the control flow. The forward-flow problem can be expressed by the following data-flow equations:

$$Out_b = f_b(In_b),$$

$$In_b = \bigcup_{p \in P} Out_p \quad \text{or} \quad In_b = \bigcap_{p \in P} Out_p,$$

where P is the set of all predecessors of block b. As can be seen from the previous equations, there are two alternatives for computing In_b, one based on using the union of predecessors' output and the second based on using the intersection of the predecessors' output. The choice between these two possibilities depends on the given problem. A better understanding of the differences between these two alternatives is demonstrated in Section 2.2.

The backward data-flow equations can be defined as follows:

$$In_b = f_b(Out_b),$$

$$Out_b = \bigcup_{s \in S} In_s \quad \text{or} \quad Out_b = \bigcap_{s \in S} In_s,$$

where S is the set of all successors of the block b.

Since there is usually no unique solution to data-flow equations, the goal of data-flow analysis is to find the most precise solution that satisfies transfer constraints and control-flow constraints.

In textbooks dealing with data-flow problems, the usage of data-flow information can be often seen expressed in terms of the Gen_b and $Kill_b$ sets for each basic block b. The variable Gen_b represents the data-flow information which is generated in basic block b and the variable $Kill_b$ denotes the data-flow information which becomes invalid in basic block b. For computing data-flow variables Gen_b and $Kill_b$, it is necessary to identify operations that are exposed in the direction of the performed analysis. An operation is exposed if it is not followed by an inverse operation in the direction of the performed analysis. For forward data-flow problems, downwards exposed operations are interesting. In contrast, for backward problems, upwards exposed operations are the center of interest.

2.2 Two Examples

In this section, two data-flow problems are presented. The first introduced problem is live variables analysis and the second one is available expressions analysis. Live variables analysis is a representative of backward data-flow problems whereas available expressions analysis belongs to forward problems. Both problems are illustrated on a sample program that uses a set of variables Var and a set of expressions $Expr$:

$$Var = \{a, b, c, d\}$$

$$Expr = \{a * b, a + b, a - b, a - c, b + c\}$$

The structure of a sample program represented as a CFG that is used for illustrating data-flow problems is shown in Figure 2.2. Both presented examples are from [33].

At the end of this section, a comparison of the considered two data-flow problems and their solutions is presented. The similarities between provided solutions are also pointed out there.

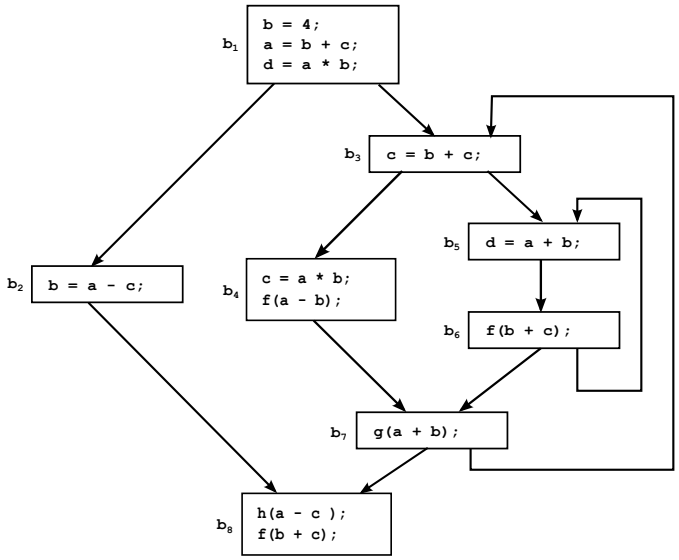


Figure 2.2: The CFG of the sample program

Live Variables Analysis

A simplified version of live variables analysis is presented here. This simplification lies in using only scalar variables. Pointer variables are not considered. Firstly, the definitions of a live variable and general data-flow equations are provided. Secondly, an application of data-flow equations on the given sample program is performed.

Definition 2.1. [33] A variable $x \in Var$ is *live* at a program point u if some path from u to End contains a use of x which is not preceded by its definition.

Data-flow equations for live variables analysis are defined in the following way [33]:

$$In_b = (Out_b - Kill_b) \cup Gen_b \tag{2.1}$$

$$Out_b = \begin{cases} BI & b \text{ is End} \\ \bigcup\limits_{s \in succ(b)} In_s & \text{otherwise,} \end{cases} \tag{2.2}$$

where

In_b is the set of live variables on the input of block b,

Out_b is the set of live variables on the output of block b,

$Kill_b$ is the set of variables whose liveness is killed within b,

Gen_b is the set of variables whose liveness is generated within b and

BI is the empty set $\emptyset$.

The use of $\cup$ in Equation 2.2 is worth noting. It points out that using a variable along a single path is sufficient to make a variable live and it is consistent with Definition 2.1. It can also be seen that data-flow information at a block b is dependent on the successor blocks. Therefore, live variables analysis is an example of a backward data-flow problem. In Table 2.1, a trace of liveness analysis for the sample program presented in Figure 2.2 is shown. In the first phase of the analysis, local information represented by variables Gen_b and $Kill_b$ is computed. Then, in the second phase, global information is obtained by traversing the CFG of the sample program.

Table 2.1: The values computed by live variables analysis

Block	Gen_b	$Kill_b$	1st Iteration		2nd Iteration	
			Out_b	In_b	Out_b	In_b
b_8	$\{a, b, c\}$	$\emptyset$	$\emptyset$	$\{a, b, c\}$	$\emptyset$	$\{a, b, c\}$
b_7	$\{a, b\}$	$\emptyset$	$\{a, b, c\}$	$\{a, b, c\}$	$\{a, b, c\}$	$\{a, b, c\}$
b_6	$\{b, c\}$	$\emptyset$	$\{a, b, c\}$	$\{a, b, c\}$	$\{a, b, c\}$	$\{a, b, c\}$
b_5	$\{a, b\}$	$\{d\}$	$\{a, b, c\}$	$\{a, b, c\}$	$\{a, b, c\}$	$\{a, b, c\}$
b_4	$\{a, b\}$	$\{c\}$	$\{a, b, c\}$	$\{a, b\}$	$\{a, b, c\}$	$\{a, b\}$
b_3	$\{b, c\}$	$\{c\}$	$\{a, b, c\}$	$\{a, b, c\}$	$\{a, b, c\}$	$\{a, b, c\}$
b_2	$\{a, c\}$	$\{b\}$	$\{a, b, c\}$	$\{a, c\}$	$\{a, b, c\}$	$\{a, c\}$
b_1	$\{c\}$	$\{a, b, d\}$	$\{a, b, c\}$	$\{c\}$	$\{a, b, c\}$	$\{c\}$

From Table 2.1, it is apparent that the values computed in the first iteration are identical to the values computed in the second iteration. This identity indicates convergence that means the end point of the analysis. We can see that after the second iteration, variables from the set $\{a, b, c\}$ are live on the output of block b_1. However, only the c variable is live on the input of block b_1 because in this block, assignments to variables a and b are performed.

The primary use of live variables analysis lies in dead code elimination and register allocation. In the first one, if a variable x is not live immediately after the assignment to x, it implies that it is not used after the assignment. Hence, it can be safely deleted. It is important to note that the removal of an unused assignment may result in a fact that variables used on the right hand side of this assignment cease to be live. A simple but ineffective solution is based on repeating the live variables analysis followed by dead code elimination until there is no change. A more reasonable solution is to design an analysis which takes this transitive effect into account. Such analysis is *faint variables analysis* that is not described here. More information and an example of faint variables analysis is given in [33].

Available Expressions Analysis

In this part, available expression analysis is explained. Firstly, definition of an available expression and general data-flow equations are provided. Secondly, an application of these equations on a given sample program from Figure 2.1 is performed.

Definition 2.2. [33] An expression $e \in Exp$ is *available* at a program point u if all paths from Start to u contain a computation of e which is not followed by an assignment to any of its operands.

Data-flow equations for available expressions analysis are defined in the following way [33]:

$$In_b = \begin{cases} BI & \text{b Start} \\ \bigcap\limits_{p \in pred(b)} Out_p & \text{otherwise} \end{cases} \tag{2.3}$$

$$Out_b = (In_b - Kill_b) \cup Gen_b \tag{2.4}$$

where

In_b is the set of expressions that are available on the input of block b,

Out_b is the set of expressions that are available on the output of block b,

$Kill_b$ is the set of expressions whose operands are modified in block b,

Gen_b is the set containing all expressions in block b which are not followed by modifications of their operands in the forward direction (downwards exposed expressions),

BI is the set of all expressions.

Unlike the live variables analysis, in available expressions analysis, $\cap$ is used in Equation 2.3. The usage of $\cap$ expresses that an expression e is available at the given point u only if all paths from Start to u contain a still valid computation of e. Consequently, it is consistent with Definition 2.2. Data-flow information in a block b is dependent on the predecessor blocks. Thus, available expressions analysis is a representative of a forward data-flow problem.

For the purpose of saving space needed for writing a set of all expressions used in a program, this set is represented by a bit vector in the following description. Figure 2.3 shows the representation. For the sake of clarity, the bit string 11111 represents the set $\{a * b, a + b, a - b, a - c, b + c\}$ whereas the bit string 00000 represents $\emptyset$.

a * b	a + b	a - b	a - c	b * c

Figure 2.3: The representation for a set of expressions

In Table 2.2, the values computed by available expressions analysis are shown. As in the case of live variables analysis, local information represented by variables Gen_b and $Kill_b$ is computed in the first iteration. Then, in the second phase, global information is obtained by traversing the CFG of the sample program. The values computed in the second iteration are identical to the values computed in the third iteration. This identity indicate convergence. For example, we can see that expressions $a * b$ and $a + b$ are available on the input of block b_6 after the third iteration.

A widespread optimization known as common subexpression elimination is based on the availability information. The task of this optimization is finding redundant expressions. An expression is marked as redundant at point u if the expression is available at that point. For the purposes of computing redundant expressions, two new variables are established, namely $AntGen_b$ and $Redundant_b$. The set $AntGen_b$ contains all expressions in the block b which are not preceded by modifications of their operands in the backward direction in

Table 2.2: The values computed by available expressions analysis

Block	Gen_b	$Kill_b$	1st Iteration		2nd Iteration		3rd Iteration	
			In_b	Out_b	In_b	Out_b	In_b	Out_b
b_1	10001	11111	00000	10001	00000	10001	00000	10001
b_2	00010	11101	10001	00010	10001	00010	10001	00010
b_3	00000	00011	10001	10000	10000	10000	10000	10000
b_4	10100	00011	10000	10100	10000	10100	10000	10100
b_5	01000	00000	10000	11000	10000	11000	10000	11000
b_6	00001	00000	11000	11001	11000	11001	11000	11001
b_7	01000	00000	10000	11000	10000	11000	10000	11000
b_8	00011	00000	00000	00011	00000	00011	00000	00011

block b. They are also called downwards exposed expressions. The variable $Redundant_b$ denotes expressions that can be eliminated in block b. Thus, the following equation holds:

$$Redundant_b = AntGen_b \cap In_b \tag{2.5}$$

In Table 2.3, the values computed for expression elimination are provided. Note that expression $a * b$ is redundant in block b_4. Therefore, it can be eliminated. In practice, a value of the previous expression computation is stored in a temporary variable and that temporary variable is used instead of redundant expression computation. For example, this form of optimization can be found in the `gcc` compiler.

Table 2.3: The values computed for expression elimination

$Block$	$AntGen_b$	$Redundant_b$
b_1	00000	00000
b_2	00010	00000
b_3	00001	00000
b_4	10100	10000
b_5	01000	00000
b_6	00001	00000
b_7	01000	00000
b_8	00011	00000

Concluding Remarks

The final part of this section summarizes the information gained in the previous two examples. From these examples, it is obvious that data-flow frameworks used to solve the presented data-flow problems have a common form that can be customized for each analysis. Generally, a data-flow problem is specified in terms of the domain, the direction of flow (forward or backward direction), the confluence operator ($\cap$ or $\cup$) and data-flow functions defined on the Gen_b and $Kill_b$ components. In Table 2.4, the differences and similarities between the two presented data-flow problems are summarized.

From Table 2.4, there is a striking resemblance between both of the presented analyses. The domain of live variables analysis is defined over sets of variables from Var whereas the

Table 2.4: Comparison of live variables analysis and available expressions analysis

	Live variables analysis	*Available expressions analysis*
Domain	2^{Var}	2^{Expr}
Confluence operator	$\cup$	$\cap$
Control-flow direction	backward	forward
Data-flow equations	$In_b = (Out_b - Kill_b) \cup Gen_b$ $Out_b = \begin{cases} BI & b \text{ is End} \\ \bigcup_{s \in succ(b)} In_s & \text{otherwise} \end{cases}$	$In_b = \begin{cases} BI & b \text{ is Start} \\ \bigcap_{p \in pred(b)} Out_p & \text{otherwise} \end{cases}$ $Out_b = (In_b - Kill_b) \cup Gen_b$
Initial value	$\emptyset$	$Expr$
Start block	b_8	b_1

domain of available expressions analysis is specified over sets of expressions from $Expr$. The confluence operator is used for collecting values from predecessor or successor blocks. In the live variables analysis, operation $\cup$ is used as the confluence operator because according to Definition 2.1, this analysis has the "any path" nature. On the other hand, operation $\cap$ is used in the available expressions analysis because Definition 2.2 requires the "all path" nature. The control-flow direction of live variables analysis is backward that is consistent with the start block b_8 and also with the definition of the data-flow equations because the Out set of the blocks depends on the successor block. On the contrary, available expressions analysis is based on forward direction of control flow. Therefore, the In set of the blocks depends on predecessor blocks and the start block is b_1.

The study of two examples of data-flow problems suggests that both analyses have similar features in terms of their specifications, formulation of data-flow equations and solution methods. These similarities can be used to design a general framework appropriate for both presented data-flow problems and many other ones. Some of them can be found in [33]. In the next section, a mathematical background is provided for constructing such a general framework.

2.3 Mathematical Background

As follows from the previous section, there are many similarities between specifications and solution methods of data-flow problems. These similarities can be used to design a general framework and specific data-flow problems can be viewed as instances of this framework. For this generalization, some underlying mathematical background is required, namely partially ordered sets and latices. This part deals with lattice-theoretic modeling of data-flow frameworks. The generalization is a mathematical abstraction that includes data-flow values, data-flow functions, confluence operators and so on. Firstly, partially ordered sets and lattices are introduced. Then, modeling data-flow values using lattices

is presented. Subsequently, modeling data-flow functions is explained. After that, the definitions of data-flow frameworks and data-flow assignments are given. The important mathematical terms are presented on the examples provided in Section 2.2.

Modeling Data-Flow Values

From the previous part, it follows that approximations amongst data-flow values and merging these values is the basis for data-flow analysis. Since many similarities in the specifications and solution methods between data-flow problems have been found, it would be appropriate to try to wrap data-flow problems in mathematics to simplify and automate the analysis. In this part, mathematical structures capable to capture the relationships between the data-flow values are defined.

Definition 2.3. [33] A *partial order* $\sqsubseteq$ on a set S is a relation over $S \times S$ that is

- reflexive: for all elements $x \in S : x \sqsubseteq x$;
- transitive: for all elements $x, y, z \in S : x \sqsubseteq y$ and $y \sqsubseteq z$ implies that $x \sqsubseteq z$;
- anti-symmetric: for all elements $x, y \in S : x \sqsubseteq y$ and $y \sqsubseteq x$ implies that $x = y$.

A *partially ordered set* (abbreviated as *poset*), denoted by $(S, \sqsubseteq)$, is a set S with a partial order $\sqsubseteq$.

In Table 2.5, the interpretation of the relation $\sqsubseteq$ is shown. The partial order $\sqsubseteq$ is interpreted as a "conservative or safe approximation of" in terms of data-flow analysis. Thus, notation $x \sqsubseteq y$ means that data-flow value x can be used instead of data-flow value y without influencing the correctness of computation. The use of data-flow value y instead of x is more exhaustive but it could lead to a loss of safety. For the purposes of data-flow analysis in this text, it is given preference to safety from exhaustiveness as is usual in the literature.

Table 2.5: The interpretation of relation $\sqsubseteq$

Notation	Interpretation
$x \sqsubseteq y$	x is weaker than y
$x \sqsubset y$	x is strictly weaker that y
$y \sqsupseteq x$	y is stronger than x
$y \sqsupset x$	y is strictly stronger than x

In posets representing data-flow values, there are often two unique elements. The first one is weaker than any other element in the poset. It is called the *least element* and it is denoted as $\bot$. The second element is stronger than any other element in the poset. It is known as the *greatest element* and it is denoted as $\top$. For defining the confluence operator, it is necessary to understand the terms from the two following definitions.

Definition 2.4. [33] Let $(L, \sqsubseteq)$ be a poset and let $S \subseteq L$. An element $x \in L$ is an *upper bound* of S iff for all $y \in S$, $y \sqsubseteq x$. Similarly, an element $x \in L$ is a *lower bound* of S iff for all $y \in S$, $x \sqsubseteq y$.

Definition 2.5. [33] The *least upper bound (lub)* of a set S is an element x such that

(i) x is an upper bound of S and

(ii) for all other upper bounds y of S, $x \sqsubseteq y$.

The *greatest lower bound (glb)* of a set S is an element x such that

(i) x is a lower bound of S and

(ii) for all other lower bound y of S, $y \sqsubseteq x$.

In Table 2.6, a different widely used notation for *lub* and *glb* is presented. Both elements, *lub* and *glb*, satisfy the idempotence, commutativity and associativity properties. The *lub* (*glb*) of a set, if it exists, is unique. From our point of view, the meet operator is more interesting because in data-flow analysis, it is used to merge data-flow values along different paths and it provides the most exhaustive safe approximation of data-flow values.

Table 2.6: Different widely used notation for *lub* and *glb*

lub	*glb*
join	meet
$\sqcup S$	$\sqcap S$

From the two examples presented in Section 2.2, it may not be obvious that posets representing data-flow values may be countably infinite. However, there are cases of data-flow problems defined on countably infinite posets. Thus, it seems reasonable to impose supplementary condition on these posets to ensure termination of the algorithms. Firstly, the definition of a chain is given. Secondly, the condition of a descending chain is defined and applied to posets. The descending chain condition is a guarantee that the meets of countably infinite sets exist.

Definition 2.6. [33] A *chain* S is a subset of a poset which is totally ordered, i.e., $\forall x, y \in S : x \sqsubseteq y$ or $y \sqsubseteq x$. A *descending chain* is a sequence of elements $\{x_1, x_2, \dots\}$ from a poset such that $i \leq j$ implies that $x_i \sqsupseteq x_j$.

Definition 2.7. [33] A descending chain $\{x_1, x_2, \dots\}$ *eventually stabilizes* iff $\exists n, \forall m > n : x_m = x_n$.

In [33], it is proved that a poset satisfies the descending chain condition iff every descending chain in the poset eventually stabilizes. However, this condition does not guarantee that the meet of each subset in a poset exists in this poset. Since this is required for analysis, there is a need to use a more restricted mathematical structure, namely a lattice.

Definition 2.8. [33] A poset $(L, \sqsubseteq)$ is a *lattice* iff for each non-empty finite subset S of L, both $\sqcup S$ and $\sqcap S$ are in L. L is a *complete lattice* iff for each subset S of L, both $\sqcup S$ and $\sqcap S$ are in L.

The top element of the lattice, $\sqcup L$, is in a complete lattice denoted as $\top$ whereas the bottom element of the lattice, $\sqcap L$, is denoted as $\bot$. Even though the guarantee of the existence

of the meet for each subset in a poset is solved by using the lattice structure, there is also a problem with some data-flow analyses whose data-flow values cannot be modeled as a lattice or complete lattice. This problem can be solved by using a less restrictive kind of lattice, specifically a *meet semilattice*. The meet semilattice is a poset in which subsets have a *glb* but do not need to have a *lub*. It means that the $\bot$ element is present in these lattices whereas the $\top$ element may not be present.

Definition 2.9. [33] A poset $(L, \sqsubseteq)$ is a *meet semilattice* iff for each non-empty finite subset S of L, $\sqcap S$ is in L.

Now, we need to move to a higher level needed for developing a general data-flow framework and it is modeling data-flow functions.

Modeling Data-Flow Functions

Data-flow equations from Section 2.2 are modified in this section. First, we recall these equations:

Live variables analysis	*Available expressions analysis*
$In_b = (Out_b - Kill_b) \cup Gen_b$	$In_b = \begin{cases} BI & b \text{ is Start} \\ \bigcap\limits_{p \in pred(b)} Out_p & \text{otherwise} \end{cases}$
$Out_b = \begin{cases} BI & b \text{ is End} \\ \bigcup\limits_{s \in succ(b)} In_s & \text{otherwise} \end{cases}$	$Out_b = (In_b - Kill_b) \cup Gen_b$

Note that both analyses are unidirectional and hence, it is not essential to have two sets of variables. Thus, in the live variables analysis equations, In_b is substituted in the equation of Out_b by the right-hand side of equation of In_b whereas in the available expressions analysis equations, Out_b is substituted in the equation of In_b by the right-hand side of equation of Out_b. For brevity, new functions f_s and f_p are introduced.

Live variables analysis	*Available expressions analysis*
$f_s(Out_s) = (Out_s - Kill_s) \cup Gen_s$	$f_p(In_p) = (In_p - Kill_p) \cup Gen_p$
$Out_b = \begin{cases} BI & b \text{ is End} \\ \bigcup\limits_{s \in succ(b)} f_s(Out_s) & \text{otherwise} \end{cases}$	$In_b = \begin{cases} BI & b \text{ is Start} \\ \bigcap\limits_{p \in pred(b)} f_p(In_p) & \text{otherwise} \end{cases}$

Instead of $\cup$ and $\cap$, the merge operator $\sqcap$ is used to merge data-flow information along different paths:

Suppose that the set of data-flow values is L. A function $f_b : L \mapsto L$ is a transformation of the data-flow values that reach the basic block b performed by the statements in block b. In the case of live variables analysis, block b is reached through the paths outgoing from

Live variables analysis	*Available expressions analysis*
$Out_b = \begin{cases} BI & b \text{ is End} \\ \bigsqcap_{s \in succ(b)} f_s(Out_s) & \text{otherwise} \end{cases}$	$In_b = \begin{cases} BI & b \text{ is Start} \\ \bigsqcap_{p \in pred(b)} f_p(In_p) & \text{otherwise} \end{cases}$

block b (backward analysis) whereas in the case of available expressions analysis, block b is reached through the paths ingoing to b (forward analysis). These functions are known as *flow functions* and they must satisfies two important properties:

- monotonicity and
- distributivity.

The monotonicity of the flow function ensures that the order of approximations is preserved. In addition to the preservation of this order, the distributivity of the flow function guarantees that merging information before applications of the functions does not cause any loss of precision. The definitions of the aforementioned properties of the flow functions are provided in the following text.

Definition 2.10. [33] A function $f : L \mapsto L$ is called *monotonic* iff

$$\forall x, y \in L : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y).$$

Definition 2.11. [33] A function $f : L \mapsto L$ is called *distributive* iff

$$\forall x, y \in L : f(x \sqcap y) = f(x) \sqcap f(y).$$

A set of flow functions F is admissible in the context of data-flow analysis if the following four properties are satisfied [33]:

- $id \in F$,
- if $f \in F$ and $g \in F$, then $f \circ g \in F$,
- the functions in F are monotonic and
- for every $x \in L$, there is a finite set of flow functions $\{f_1, f_2, \ldots, f_m\}$ such that

$$x = \bigsqcap_{1 \leq i \leq m} f_i(BI).$$

Data-Flow Frameworks

Until now, the fundamental steps needed for developing a general data-flow framework have been discussed. Now, we have enough knowledge to model data-flow values and data-flow functions. Therefore, the general data-flow framework can be introduced.

Definition 2.12. [33] A *data-flow framework* is a tuple $(L_{\mathcal{G}}, \sqcap_{\mathcal{G}}, F_{\mathcal{G}})$, where $\mathcal{G}$ is a symbol standing for an unspecified CFG and:

- $L_{\mathcal{G}}$ is a description of a meet semilattice that represents the data-flow values relevant to the problem. $L_{\mathcal{G}}$ must satisfy the descending chain condition.
- $\sqcap_{\mathcal{G}}$ is a description of the meet operator of the semilattice. $\sqcap_{\mathcal{G}}$ is, of course, derivable from $L_{\mathcal{G}}$.
- $F_{\mathcal{G}}$ is a description of the set of admissible flow functions from $L_{\mathcal{G}}$ to $L_{\mathcal{G}}$. Each flow function has an associated direction which can be along the control flow in the unspecified CFG $\mathcal{G}$ or against it.

Data-flow frameworks defined in this way are also known as *monotone data-flow frameworks*, which follow from the property of monotonicity required for admissible functions. If the property of distributivity is satisfied, a framework is called *distributive data-flow framework*. For solving a real data-flow problem, an instance of a general data-flow framework must be defined, which we do next.

Definition 2.13. [33] An *instance of a data-flow framework* is an instantiation of a framework to a particular CFG. It is a pair $(\mathbb{G}, M_{\mathbb{G}})$, where

- $\mathbb{G} = (\mathbb{N}odes, \mathbb{E}dges)$ is an instance of $\mathcal{G}$. This yields concrete values $L_{\mathbb{G}}$, $\sqcap_{\mathbb{G}}$ and $F_{\mathbb{G}}$ for $L_{\mathcal{G}}$, $\sqcap_{\mathcal{G}}$ and $F_{\mathcal{G}}$.
- $M_{\mathbb{G}}$ is a mapping from blocks in $\mathbb{G}$ to $F_{\mathbb{G}}$.

Table 2.7 shows two instances of a general data-flow framework for our presented data-flow problems. For completeness, the direction of the analysis, $\top$ and $\bot$, is also presented.

Table 2.7: Two instances of a data-flow framework

	Live variables analysis	*Available expressions analysis*
$\mathbb{G}$	CFG presented in Figure 2.2	
$L_{\mathbb{G}}$	2^{Var}	2^{Expr}
$\sqsubseteq_{L_{\mathbb{G}}}$	$\supseteq$	$\subseteq$
$\sqcap_{\mathbb{G}}$	$\cup$	$\cap$
direction	backward	forward
$\bot$	Var	$\emptyset$
$\top$	$\emptyset$	$Expr$

Data-Flow Assignments

A data-flow assignment is a mapping from each data-flow variable In_b to a data-flow value. In the context of data-flow analysis, a safe (also called conservative) data-flow assignment is required in most cases. Before the definition of a safe assignment, a set $paths(p)$ and a function f_ρ are necessary to be defined. The set $paths(p)$ represents all paths from the Start node to the program point p. The function f_ρ denotes the composition of functions belonging to the blocks $\{b_1, b_2, \ldots, b_i\}$ in a path ρ. Thus, $f_\rho = f_{n_{i-1}} \circ \cdots \circ f_2 \circ f_1$. Now, we can proceed to the definition of a safe assignment.

Definition 2.14. [33] An assignment represented by the values of data-flow variables In_b is *safe* iff

$$\forall b \in \mathbb{N}odes : In_b \sqsubseteq \bigsqcap_{\rho \in paths(b)} f_\rho(BI).$$

Two data-flow assignments are briefly covered in the following part, namely a meet over paths assignment and a maximum fixed point assignment.

Definition 2.15. [33] A *meet over paths assignment*, denoted as MOP, is the maximum safe assignment:

$$\forall b \in \mathbb{N}odes : MOP_b = \bigsqcap_{\rho \in paths(b)} f_\rho(BI).$$

As can be seen from the definition of MOP, it is a path-based assignment. However, many data-flow problems are edge-based (like those presented in Section 2.2). For some problems, the MOP assignment is not algorithmically computable. Thus, another approach is required. The solution of this problem is called a maximum fixed point assignment (also denoted as MFP). The MFP assignment is based on computing a fixed point whose definition follows.

Definition 2.16. [33] A *fixed point* of a function $f : L \mapsto L$ is a value $v \in L$ that satisfies $f(v) = v$.

For the fixed point assignment, the value of variable In_b is denoted by FP_b. The maximum point assignment is denoted by MFP_b, and it must hold that

$$\forall b \in \mathbb{N}odes : FP_b \sqsubseteq MFP_b.$$

To illustrate, consider a simple example of a CFG in Figure 2.4. The first tree next to the CFG depicts an expression tree for MFP whereas the second tree displays an expression tree for MOP. From this figure, it is obvious that MOP does not merge values at intermediate points. In contrast, MFP performs the merging of information at all the intermediate points.

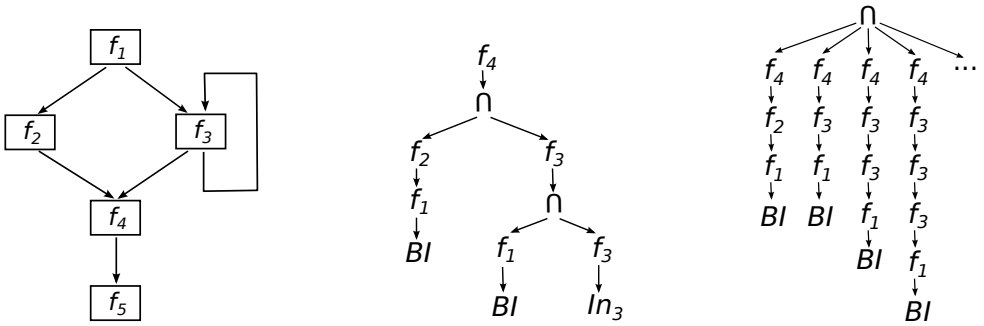


Figure 2.4: The illustration of the MOP and MFP assignment [33]

2.4 Intraprocedural Analysis

Everything that was said in the previous parts can be applied to intraprocedural analysis. Hence, in this part, only an explanation of the term intraprocedural analysis is provided.

Also, an algorithm for computing intraprocedural data-flow analysis is given, namely the round-robin algorithm. However, before the introduction of the algorithm, we define the scope for intraprocedural analysis. In particular, intraprocedural analysis is restricted to a single function and ignores function calls. However, there are several improvements of intraprocedural analysis that approximate the results from the called functions. Now, we can proceed to the round-robin algorithm.

Round-Robin Algorithm

In this part, the round-robin algorithm for performing intraprocedural data-flow analysis is introduced. From the following pseudocode of this algorithm, it is obvious that it is determined for forward data-flow problems because it starts computation from the Start block and the data-flow information depends on the predecessor blocks. However, it is not hard to modify this source code to perform backward analysis. Since the merging of the data-flow information is computed at the intermediate points, the algorithm computes the MFP assignment. It is intended for frameworks with a complete lattice.

Algorithm 1: Round-Robin Algorithm For Intraprocedural Data-Flow Analysis

Input: An instance $(\mathbb{G}, M_{\mathbb{G}})$ of a monotone data-flow framework $(L_{\mathcal{G}}, \sqsubseteq_{\mathcal{G}}, F_{\mathcal{G}})$. The function to which $M_{\mathbb{G}}$ maps a node b is denoted as f_b. The Start node is numbered by 0. The rest of the nodes are ordered from 1 to $N-1$.

Output: Data-flow values In_b for each block b.

```
// Initialization part.
1  In_0 = BI;
2  foreach j ∈ {1, 2, ..., N − 1} do
3  |  In_j = ⊤;
4  end
5  change = true;
   // Iterative part
6  while change do
7  |  change = false;
8  |  foreach j ∈ {1, 2, ..., N − 1} do
9  |  |  temp = ⊓_{p∈pred(j)} f_p(In_p);
10 |  |  if temp ≠ In_j then
11 |  |  |  In_j = temp;
12 |  |  |  change = true;
13 |  |  end
14 |  end
15 end
```

In the initialization part, we initialize the value of all input blocks to $\top$, except for the Start node, which we initialize to the empty set. Then, in the iterative part, we keep iterating over the nodes. In every iteration, we re-compute the input value of all blocks based on the input value of all the predecessors. We repeat this process until we reach the fixed point—that is, the values do not change anymore.

Improving the Algorithm By Using a Work List

A classical way of improving the above algorithm is to use a so-called *work list* of nodes which need to be processed. This approach is based on the observation that the input of a block will not change if the input blocks of its predecessors do not change.

The modified algorithm works as follows. Instead of a boolean flag *change*, we use a list of nodes, *todo*. We initialize it by putting In_0 into it. Then, we keep iterating until *todo* is empty—that is, everything that needed to be processed has been processed. During a single iteration, we remove the first node from *todo*, and re-compute its input. If the input has changed, we put all of its successors into *todo*. Such a modified version of the Round-Robin algorithm is called a *work list algorithm*.

From a practical viewpoint, it may be desirable to use a queue or a set instead of a list. Moreover, for efficiency, a block should not be in the work list more than once.

Chapter 3

Value-Range Analysis

As mentioned in Chapter 1, value-range analysis is a static analysis technique based on arguing about the values that a variable may take at a given program point. In other words, it tries to deduce bounds on the ranges of values that a variable may have at various program points during the program execution. Since it belongs to static analyses like data-flow analysis, it operates solely on the source code and neither modifies nor executes the program directly. However, there are approaches used for implementing value-range analysis that can execute the program in an abstract way. Among these approaches belongs abstract interpretation, which is discussed later in this chapter.

Value-range analysis found extensive applications in many compiler optimizations. In [42], it is utilized to predict the likelihood of a particular branch being taken at compile time. The result provides useful information for several optimizations. Among the most interesting optimizations that benefit from branch predictions belong global instruction scheduling, code layout, instruction cache optimizations, function inlining and interprocedural register allocation. In addition, [31] and [42] employ value-range analysis to remove unnecessary bound checks. In [47], value-range analysis is used to show that some operations cannot produce overflows. This knowledge is subsequently utilized for removing redundant overflow tests from the program text resulting in decreasing code size and making code motion optimization easier. According to [31] and [37], value-range algorithms can be used to derive an upper bound on the number of iterations made when a loop is entered. In [48], bitwidth analysis is formulated as a value-range analysis problem. Since bitwidth analysis is essential to the bitwidth aware register allocator (see [50]) and also useful to synthesize hardware (see [11, 38]), value-range analysis plays an important role in both these areas.

Additionally, results gained from value-range analysis provide valuable information for the purposes of proving programs to be correct. According to [45, 55, 56], information inferred by a value-range analyser can be used to detect buffer overflow vulnerabilities. Besides the aforementioned usages of value-range analysis, [31] mentions that value-range analysis can also be used to provide diagnostic information and choose a suitable data representation.

This chapter is organized as follows. The first part provides a motivating example that shows the usefulness of value-range analysis. In the second part, different ways of implementing value-range analysis are presented. Then, the third part introduces existing tools that perform value-range analysis, especially the value-range analysis plug-in from the Frama-C platform [25].

3.1 A Motivating Example

Although buffer overflow problems in C programs have been recognized since the early '70s, we still encounter these problems in many today's applications. Indeed, as demonstrated in Chapter 1, buffer overflows still belong to active attack methods. Since value-range analysis can be successfully used to detect buffer overflows in programs, an example of a source code with a simple buffer overflow is presented in the following text. Subsequently, we run a hypothetical value-range analyser over the given source code to see what actions it does when analysing the code.

In Figure 3.1, a simple C program with a buffer overflow is shown. The `importantData` variable is initialized with `1` on line 5. On line 6, a static array named `buffer` with ten elements is declared. In the `for` loop on lines 9 through 11, `9999` is assigned to each element of this array. Subsequently, the value of the `importantData` variable is printed on line 13. Since the `importantData` variable is apparently not changed in the program, the expected output is `importantData = 1`. However, after compiling the source code and running the resulting program, we see that `importantData = 9999` is printed.

```
 1 #include <stdio.h>
 2
 3 int main(int argc, const char *argv[])
 4 {
 5   int importantData = 1;
 6   int buffer[10];
 7
 8   int i;
 9   for (i = 0; i <= 10; i++) {
10     buffer[i] = 9999;
11   }
12
13   printf("importantData = %d\n", importantData);
14
15   return 0;
16 }
```

Figure 3.1: A simple C program containing a buffer overflow problem

Now, suppose that we have a hypothetical value-range analyser that provides a safe analysis of the value ranges the variables may have. After running the analyser over the source code from Figure 3.1, the results could look as shown in Figure 3.2. As we can see from this output, the variable `i` can hold an arbitrary value from the interval $\langle 0, 10 \rangle$. Since the size of the array `buffer` is 10 and the size of the interval is 11, the results for variable `i` are suspicious. However, for a value-range analyser, it is typical to over-approximate the results. Thus, we can suppose that an over-approximation happened in this case and continue to the next line of the output. All elements of the `buffer` array can hold only 9999 which agrees with our assumptions. According to the last line of the output, the variable `importantData` can hold 1 or 9999. However, we do not see any assignments to the variable `importantData` except the one on the line 5 which assigns 1 to it. Since there are no other visible modifications of the variable `importantData` and the program actually prints `importantData = 9999`, our value-range analyser cannot over-approximate in this case. Hence, it provides precise results. Thus, the value 9999 must be assigned

to the variable `importantData` indirectly. When we look carefully at the output of the analyser, we can see that the value 9999 is used only when assigning to `buffer`. This knowledge together with the suspicious result interval for the variable `i` indicates an off-by-one error and a buffer overflow. Because this is a very simple example, even less experienced programmers may notice that the buffer is accessed outside its bounds and do not need a value-range analyser. However, in more complex programs, the problem may be hidden from the eyes of programmers. In these cases, value-range analysers can be very useful.

```
i = {<0, 10>}
buffer = {<9999, 9999>}
importantData = {<1, 1>, <9999, 9999>}
```

Figure 3.2: Output of the hypothetical value-range analyser

3.2 Approaches

In this section, different ways of implementing value-range analysis are introduced. Attention is devoted primarily to abstract interpretation because this approach was chosen for our value-range analyser. In the second part, other possible approaches are presented.

3.2.1 Abstract Interpretation

Abstract interpretation was formalized by Patrick Cousot and Rhadia Cousot in 1977 in [13]. It is a theory of approximating the semantics of programming languages. It belongs to static analysis techniques that execute analysed programs in an abstract way. Informally, the central idea of abstract interpretation lies in interpreting an abstract version of the analysed program. The abstraction preserves only important program properties and abstracts away from irrelevant details. It must be guaranteed that the abstraction is sound so that all possible errors are found. It is also required for the abstraction to be precise enough to avoid false alarms. However, the abstraction should also be as simple as possible to evade a combinatorial explosion in the number of reachable abstract program configurations. The last two requirements on abstraction go significantly against each other. In practice, it is necessary to find a reasonable compromise between these two requirements. The most prevalent application for abstract interpretation lies in formal verification. Currently, it is used in many automated verification tools. Some of them are introduced in Section 3.3. The information about abstract interpretation in this part is drawn from [36, 39].

Since abstract interpretation can be formally defined by Galois connections, we try to informally show why Galois connections are useful for abstract interpretation and then we define them formally. Let $\mathcal{P}$ be a poset that represents program properties such as shapes of data structures, lower and upper bounds for real numbers and the like. Then, let p_1 and p_2 be two program properties from $\mathcal{P}$. Assume that *prog* represents a program that transforms the property p_1 to another property p_2 that can be written as $prog \vdash p_1 \rhd p_2$. In many cases, it is too costly or even uncomputable to perform the analysis $prog \vdash p_1 \rhd p_2$ in $\mathcal{P}$. Thus, it may be suitable to replace $\mathcal{P}$ by a simpler poset $\mathcal{Q}$. For this purpose, we have to find a representation of the elements of $\mathcal{P}$ in $\mathcal{Q}$. This can be done by an *abstraction function* that is in the following definition labeled as α. The abstraction function represents elements

of $\mathcal{P}$ as elements of $\mathcal{Q}$ by leaving irrelevant details out of consideration. On the other hand, it is also necessary to express the meaning of elements of $\mathcal{Q}$ in terms of elements of $\mathcal{P}$. This service is provided by a *concretization function* that is labeled as γ in the following definition. Since it is possible to safely move from one poset to another one and vice versa, we can perform the analysis $prog \vdash q_1 \rhd q_2$ in $\mathcal{Q}$. If the abstract and concretization functions are properly designed, then the results of the analysis computed in $\mathcal{Q}$ are sound and the analysis is less demanding. Next, we give the mathematical definition of a Galois connection.

Definition 3.1. A *Galois connection* is a quadruple $\pi = (\mathcal{P}, \alpha, \gamma, \mathcal{Q})$, where

- $\mathcal{P} = (P, \leq)$ and $\mathcal{Q} = (Q, \sqsubseteq)$ are posets,
- $\alpha : P \to Q$ and $\gamma : Q \to P$ are functions such that $\forall p \in P$ and $\forall q \in Q$:

$$p \leq \gamma(q) \iff \alpha(p) \sqsubseteq q.$$

In Figure 3.3, elements of a Galois connection are shown. As is mentioned in the previous text, α is an abstraction function and γ is a concretization function. P is considered to be a *concrete domain* whereas Q is an *abstract domain*. The elements of P are known as *concrete contexts* and the elements of Q are called *abstract contexts*. In side boxes, we can see examples of concrete and abstract contexts used in the following illustration of constructing a Galois connection. Suppose that we have a programming language that can use only one `unsigned int` variable and two instructions, specifically incrementing and decrementing a variable by one. Our task is to design a value-range analyser that displays which values can our variable gain during a program execution at different program points. Let the range of `unsigned int` be denoted by $\mathbb{I}$ (for example, on 32 bits, it can be $\langle 0, 4294967295 \rangle$). Since during the value-range analysis values obtained for the variable at a certain program location can be represented by an arbitrary subset of $\mathbb{I}$, the concrete domain P is $2^{\mathbb{I}}$. The order of $\mathcal{P}$ coincides with the standard set inclusion. However, it can be unnecessarily complicated to perform the analysis in $\mathcal{P}$. Thus, instead of using a set of numbers to represent values acquired by the variable during the analysis, we may use an interval. Then, the abstract domain Q is a set of intervals. Suppose that q is an element of Q, and that q_{min} and q_{max} denote the minimal and maximal number of q, respectively. Then, the order of $\mathcal{Q}$ can be defined as

$$\forall x, y \in Q \colon x \sqsubseteq y \iff \{i \mid x_{min} \leq i \leq x_{max}\} \subseteq \{j \mid y_{min} \leq j \leq y_{max}\}.$$

The abstraction function α takes responsibility for converting elements of P to elements of Q. Assume that p is an element of P. Since p is a set of numbers, let $\min(p)$ denote the minimal number in p and $\max(p)$ denote the maximal number in p. With these assumptions,

$$\alpha(p) = \langle \min(p), \max(p) \rangle.$$

Suppose that q is an element of Q, and recall that q_{min} and q_{max} denote the minimal number and maximal number of q, respectively. The concretization function γ is defined as

$$\gamma(q) = \{x \mid q_{min} \leq x \leq q_{max}\}.$$

Now, we have completely defined a Galois connection for our example. Next, we provide a mathematical structure for defining abstract interpretation and then we continue with constructing this structure for this illustration.

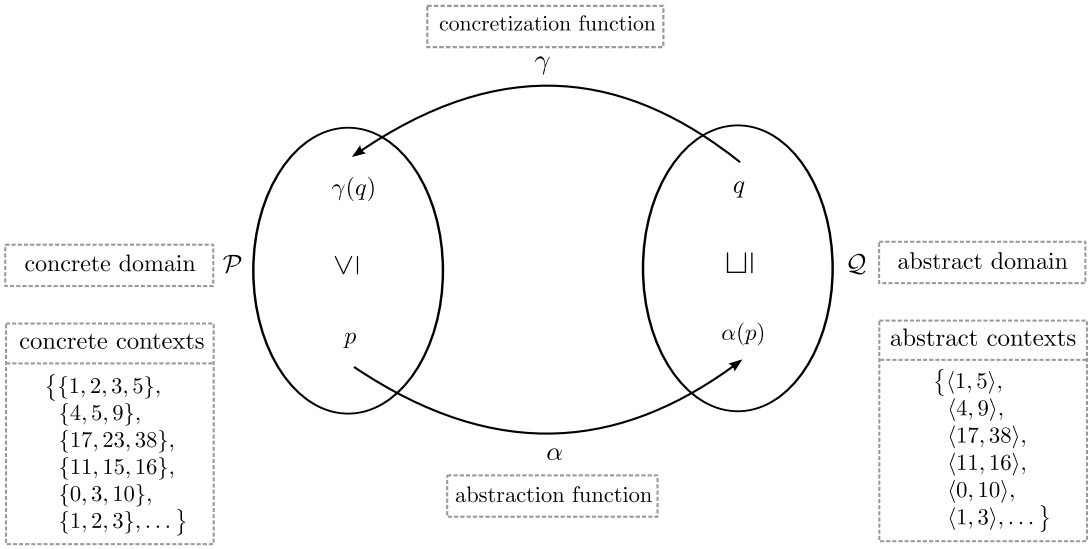


Figure 3.3: Galois connection $\pi = (\mathcal{P}, \alpha, \gamma, \mathcal{Q})$

Definition 3.2. An *abstract interpretation* $\mathcal{I}$ of a program *prog* with the instruction set `Instr` is a tuple $\mathcal{I} = (Q, \circ, \sqsubseteq, \top, \bot, \tau)$, where

- Q is a set of abstract contexts,
- $\top \in Q$ is the supremum of Q,
- $\bot \in Q$ is the infimum of Q,
- $\circ : Q \times Q \to Q$ is an operation for accumulation of abstract contexts, together with Q and $\top$ yielding the complete semilattice $(Q, \circ, \top)$,
- $(\sqsubseteq) \subseteq Q \times Q$ is an ordering defined as $x \sqsubseteq y \Leftrightarrow x \circ y = y$ in $(Q, \circ, \top)$,
- $\tau : \texttt{Instr} \times Q \to Q$ defines an interpretation of instructions, τ is monotonic.

Now, we can continue with our illustration. Since abstract interpretation is performed in $\mathcal{Q}$, we need to represent $(Q, \sqsubseteq)$ in the abstract interpretation structure. Both Q and $\sqsubseteq$ are defined in the first part of this example. The supremum is represented by $\langle 0, \texttt{UINT\_MAX}\rangle$, where `UINT_MAX` denotes the maximal number of our `unsigned int` data type. The infimum is represented by the empty interval, denoted by $\langle\rangle$. To accumulate abstract contexts on branch and loop junctions, the operation $\circ$ is utilized. For our purposes, we can define $\circ$ as

$$x \circ y = \langle \min(x_{min}, y_{min}), \max(x_{max}, y_{max})\rangle.$$

Since our instruction set `Instr` consists of two instructions, incrementation and decrementation, we need to specify their meaning in $\mathcal{Q}$. For each instruction, there exists an abstract transformer in the form of the function τ. Assume that our variable is called a and the interval on which the instruction is performed is x. Then, abstract transformers can be defined by

$$\tau(a + 1, x) = \langle x_{min} + 1, x_{max} + 1\rangle$$

and

$$\tau(a-1,x) = \langle x_{min} - 1, x_{max} - 1 \rangle.$$

It is important to note that if abstraction and concretization functions form a Galois connection which is used by an abstract interpretation, the abstract interpretation may only over-approximate. The proof of this statement can be found in [36].

Abstract interpretation is based on an fixed point computation. However, computation of the most precise fixed point can be time consuming, and its termination is not even guaranteed (loops in which variables from infinite domains are manipulated). For these reasons, widening operators can be used. They belong to a special class of upper bound operators used for over-approximation of a fixed point. The mathematical definition follows.

Definition 3.3. Let $\mathcal{I} = (Q, \circ, \sqsubseteq, \top, \bot, \tau)$ be an abstract interpretation of a program. The binary *widening operation* $\triangledown$ is defined as:

- $\triangledown : Q \times Q \to Q$,
- $\forall C, D \in Q : (C \circ D) \sqsubseteq (C \triangledown D)$,
- for all infinite sequences $(C_0, \ldots, C_n, \ldots) \in Q^\omega$, it holds that the infinite sequence $(s_0, \ldots, s_n, \ldots)$, defined recursively as

  $$s_0 = C_0,$$

  $$s_n = s_{n-1} \triangledown C_n,$$

  is not strictly increasing and because the result of $\triangledown$ is an upper bound, the sequence eventually stabilizes.

By using the technique of widening we get too rough results. So, in the following text, another technique that copes with this problem is introduced. This technique is called narrowing and it can be used only after performing widening. The purpose of narrowing operators usage lies in refining the approximation of a fixed point gained from the widening technique. For completeness, the mathematical definition of narrowing operator is provided below.

Definition 3.4. Let $\mathcal{I} = (Q, \circ, \sqsubseteq, \top, \bot, \tau)$ be an abstract interpretation of a program. The binary *narrowing operation* $\triangle$ is defined as:

- $\triangle$: $Q \times Q \to Q$,
- $\forall C, D \in Q : C \sqsubseteq D \to (C \sqsupseteq (C \triangle D) \sqsupseteq D)$,
- for all infinite sequences $(C_0, \ldots, C_n, \ldots) \in Q^\omega$, it holds that the infinite sequence $(s_0, \ldots, s_n, \ldots)$, defined recursively as

  $$s_0 = C_0,$$

  $$s_n = s_{n-1} \triangle C_n,$$

  is not strictly decreasing and because the result of $C \triangle D$ is a lower bound of C, the sequence eventually stabilizes provided that the input sequence is not strictly increasing.

In our example, widening may be used when analysing infinite loops, like `while(true) i++;`. In such a case, after a certain prescribed amount of iterations is performed, we over-approximate the value of `i` to the maximal interval. A more complex example that uses both operators is presented in [36].

Usage of abstract interpretation for the purposes of value-range analysis can be found in several successful tools used in industry, like Frama-C, AbsInt's ValueAnalyser and Polyspace products. These tools are introduced in Section 3.3. Moreover, in [45], Simon introduces an algorithm based on abstract interpretation. The main goal of this algorithm is to detect buffer overflows. According to Simon, the proposed algorithm is sound. In [55], Venet and Brat use abstract interpretation to implement an array-bound checker specialized for NASA software. There are also other examples of using abstract interpretation for the purposes of value-range analysis. However, the aim of this work is not a summary of all of them. Thus, we can move to the next section, which presents other approaches to value-range analysis.

3.2.2 Other Approaches

In addition to abstract interpretation, there exist also other approaches to value-range analysis. An overview of some of them, together with references, is provided in this section.

In 1977, Harrison in [31] proposed an algorithm for value-range analysis. The main idea of this algorithm lies in decomposing the problem into two mechanisms, specifically the *value-range propagation* and *value-range analysis*. The value-range propagation mechanism is based on a simple algorithm that uses data and the conditional structure of an analysed program to derive and propagate refinements from the point of an assignment to a variable to the points where this variable is used. The value-range analysis tracks the changes applied to a variable at each program point in a loop. However, unlike the value-range propagation, the value-range analysis ignores the conditional structure of the loop which leads to limited accuracy. However, when results of the value-range analysis are intersected with those obtained by value-range propagation, we get more precise results. Subsequently, the value-range propagation continues with these results. Despite the fact that this value-range algorithm was designed for a compiler optimization, nowadays different modifications of this algorithm are used not only in compiler optimizations but also in a formal verification. For example, in [42], Patterson introduced a static branch prediction by using Harrison's value-range propagation mechanism. Also, value-range analysis, introduced in [56], used to detect buffer overflows is built on Harrison's work. A value-range propagation technique similar to Harrison's is presented in [48], where it is utilized for minimizing the bitwidth of operands. In other words, it is employed to minimize the number of bits used to represent every integer and pointer in a program. Dynamic elimination of overflow tests in a trace compiler, described in [47], also uses a modified version of Harrison's value-range algorithm.

Another approach for implementing value-range analysis of integer variables is studied in [10]. Firstly, the analysed source codes are converted to a suitable intermediate representation. From this intermediate representation, it must be easy to extract equational constraints imposed on the variables that are used for building a constraints graph in the next step. This graph captures the dependencies between the constraints. In the last step, different fixed-point iterators are applied on the constraints graph and the constraints are solved. See [10] for more details.

3.3 Existing Tools

In this section, several value-range analysis tools are introduced. Attention is paid mainly to Frama-C, which is an extensible and collaborative platform dedicated to static analysis of source code written in the C programming language. The description of this platform is based on [9, 12, 14, 15, 25]. In the conclusion of this section, which is based on [1, 9, 43], a brief introduction of AbsInt's ValueAnalyser and Polyspace is provided. All of the aforementioned static analysers are widely used in industry.

3.3.1 The Frama-C Platform

Frama-C is an open-source platform written in the OCaml language. It has been developed since 2004 by the CEA LIST and the INRIA labs, and its development is still ongoing. Frama-C is distributed in two forms: source code and binary. Binaries are available for all popular architectures. In all distributions, the Frama-C kernel and several ready-to-use analyses are included. Its modular architecture makes Frama-C an easy-to-extend tool. Since Frama-C is a plug-in-based platform, each analyser comes in the form of a plug-in. Plug-ins can collaborate with Frama-C's kernel and other plug-ins in order to utilize the results computed by the kernel or other analyses. Moreover, Frama-C has been designed to provide the capability of developing new specific analyses in the form of plug-ins through its extensible API. In addition, Frama-C's plug-ins can be used in a batch or interactive mode under Windows, UNIX and Mac OS X.

One of the means that allow different analyses to collaborate in Frama-C the is ANSI/ISO-C Specification Language (ACSL). This collaboration can be achieved by exchanging information through ACSL annotations. An ACSL annotation is written as a particular comment in the C source code and its purpose is to formally specify source code properties. It can originate from three different sources: the user, Frama-C's kernel and various plug-ins. ACSL uses Hoare-like expressions of properties such as preconditions, postconditions, invariants and so on. More details about ACSL are provided in [5].

In Figure 3.4, the architecture of the Frama-C platform is presented. The repository is used to store project management information, states and results of analyses, dependencies between analyses and so on. Therefore, data can be shared by different plug-ins through this repository. The Frama-C kernel helps to centralize information, guides the analysis and provides common functionalities. Remaining boxes represent some of the plug-ins distributed with the Frama-C kernel. Their functionality is described in the following text. The Aoraï plug-in [49] offers a method to automatically annotate C source codes. Also, the RTE plug-in [32] can be used for automatic generation of annotations, particularly annotations for common run-time errors, unsigned integer overflows, precondition and postcondition checking and so on. The Metrics plug-in [8] computes complexity measures of the C source code. The WP plug-in [4] uses weakest precondition calculus to prove ACSL annotations of C functions, and the Mthread plugin [57] is dedicated for analysing concurrent C programs. The Slicing plug-in [26] is used to convert the original program to a simplified one.

The last plug-in, which is the center of our interest, is called Value Analysis [15]. As its name suggests, it performs value-range analysis. Thus, it computes sets of possible values for each variable of the program at each point of the execution. The theoretical framework on which this plug-in is based is abstract interpretation, introduced in the previous section.

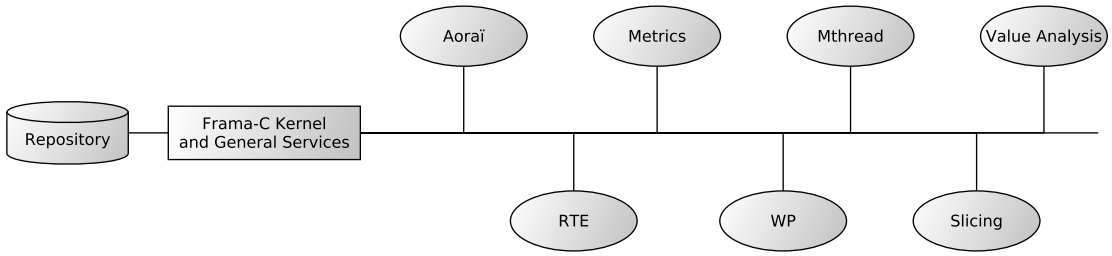


Figure 3.4: Architecture of the Frama-C platform [9]

For each variable, abstract interpretation computes over-approximated sets of all possible values acquired by the variable and labels all instructions that are identified as dangerous from the viewpoint of safety. Subsequently, it warns about all possible run-time errors in the analysed program. As for other plug-ins, the results of value-range analysis is stored in the repository. Variables are divided into several variation domains by the plug-in. The first variation domain is a set of integers and it covers all variables of an integral type. This domain can be represented as an enumeration, interval without periodicity information or interval with periodicity information. The second domain contains a floating-point value or an interval of floating-point numbers. The third variation domain represents a set of addresses.

The Value-Range plug-in can handle analysis of complete and incomplete applications. For the purposes of incomplete applications and libraries, an option to specify the entry point for the analysis can be used. Moreover, it is important to realize that the analysis of the source code containing an infinite loop can never force the analyser itself to loop forever. Thus, the analyser needs to use over-approximations when analysing a loop. This can lead to broadly over-approximating the sets of possible values of the variables in the analysed program. So, three options that can be used to refine the analysis of the loops are introduced in the following text. The first one is placing an ACSL annotation before a loop to suggest the analyser which values should be preferably used for which variables when widening the sets of values attached to variables. The second option when taking care of a loop is using a loop invariant that describes some properties holding at the beginning of each execution of the loop. The last option is loop unrolling which means that the body of the loop is unrolled as many times as specified. Similarly, without any special requirements set by the user, function calls are taken care of by expanding the bodies of the functions at the call sites.

The user can interact with the Value Analysis plug-in by inserting calls to pre-defined functions into the analysed source codes. These functions are in the terminology of Frama-C called primitive functions or primitives. They can be used for different reasons. Firstly, since many applications use source codes that are part of the system rather than applications themselves and these source codes are not necessarily available, primitives can be used for emulating the standard C library functions, like `malloc()`, `strncpy()`, `atan()` and so on. Primitive functions try to model the behaviour of the standard C library function as precisely as possible. Secondly, primitives declared in `share/builtin.c` allow us to introduce some non-determinism in the analysis. For example, the `Frama_C_nondet()` function randomly selects one of two given integer variables, the `Frama_C_nondet_ptr()` function randomly chooses one of two given pointer variables, and functions `Frama_C_interval()`, `Frama_C_float_interval()`, `Frama_C_double_interval()` return an `int`, `float` or `double`

number from the given interval. Thirdly, primitives can be also used for observing the results of the analysis that would not be observed otherwise. For displaying the entire memory state, the function `Frama_C_dump_each()` is used. Moreover, the function `Frama_C_show_each_name()` displays the values of a given expression.

Since nothing is perfect, the Value Analysis plug-in has also some limitations. The correctness of results provided by this plug-in is under assumption that the false-positive alarms emitted during the analysis have been verified by the user. If only one assert emitted by the plug-in is violated, it is also not possible to rely on the results of the analysis. Moreover, the current version of the plug-in is not capable to precisely analyse programs that use variadic functions. Besides, at present, it is not able to handle recursive functions at all. Programs using interrupts are another problem causing that the result of the analysis do not reflect reality and cannot be considered as sound. Proper alignment of memory accesses in memory is not checked by the plug-in at present. The Value Analysis plug-in supposes that pointer casts are utilized in ways that are compatible with strict aliasing rules. Otherwise, the result produced by the analyser need not be correct.

Although the ISO standard leaves the behaviour of many constructs of the C language unspecified, these constructs are compiled in the same way by almost all compilers for almost all architectures. Thus, the Value-Range plug-in assumes a reasonable compiler and target architecture for analysing some of these constructs. In this way, more usable information can be obtained than by strictly conforming to the ISO C standard. However, the correctness of the analysis does not depend on the standard itself, but rather on the standard and the way compilers implement unspecified constructs. For instance, the ISO C standard does not specify the result of incrementing a signed variable holding the maximum integer value. However, most compilers make the variable simply overflow.

Frama-C never remains silent for a location in the source code where an error can happen at run-time. Since the users are informed about all potential run-time flaws detected in the source code by Frama-C, a guarantee that there are no bugs in a program is provided. Achieving the goal of correctness brings also one inconvenience: the Frama-C platform emits warnings for constructs that do not cause any run-time errors. Thus, it produces false alarms. However, Frama-C is still quite precise despite the pitfalls of the C language.

3.3.2 Other Tools

AbsInt's ValueAnalyzer [2] is another successful static program analyser. It performs a value-range analysis that is fully automatic and conservative. It is primarily focused on embedded systems. It allows collaboration with other analysis tools from AbsInt. For example, it interacts with AbsInt's StackAnalyzer [1] to provide an analysis of stack usage. In collaboration with other AbsInt's analysers, it can detect unreachable code, validate user-defined assertions and provide reports on all memory accesses and function calls.

Other static analysis tools that have found application in industry are Polyspace products [43]. The aim of Polyspace tools is to verify source code of embedded systems, especially to detect run-time errors and to formally prove their absence in C, C++ or ADA source codes. Moreover, these tools can be exploited to compute a subset of dead code, check compliance to coding standards, measure software quality, review code complexity metrics and last but not least to determine ranges of variables and function return values. Similarly to Frama-C, Polyspace products are based on abstract interpretation.

Chapter 4

Code Listener Infrastructure

Code Listener [17] is a completely open-source infrastructure intended to simplify construction of tools for static analysis of C programs. The development of Code Listener has started at Brno University of Technology in 2010. Its long-term goal is wrapping the interfaces of existing code parsers and providing a unified, well-documented, object-oriented and easy to use API over these source code parsers. Simply speaking, the Code Listener infrastructure transforms the interfaces provided by parsers into its own simplified interface. So, the Code Listener API exposed to analysers is completely independent of these parsers. Thus, it is possible to run the same analyser on top of different code parsers without touching the code of this analyser.

The aforementioned approach of using the existing code parsers to process source codes instead of leaving the job on static analysers brings several pros and cons. Since code parsing is performed only once, we spare time as well as energy. It is worth noting that every source code the parser is able to parse the analyser is able to use as its input. Therefore, static analysers cannot fail due to problems with source code parsing. In addition, the Code Listener infrastructure provides a uniform interface for reporting errors. This means that errors are reported in the way programmers of static analysers are accustomed from using the code parser. The main downside of this approach lies in the fact that Code Listener is not a standalone tool and it is completely dependent on code parsers. Hence, it is necessary to devote a considerable effort to establish support for different code parsers and maintain the interface in response to changes in these parsers.

Code Listener is distributed as a C++ library. Since developed industrial compilers allow to insert extra static-analysis passes at run-time in the form of compiler plug-ins, the Code Listener infrastructure takes advantage of this fact. Hence, a static analyser is built as a compiler plug-in that takes the internal representation of the source code from a compiler via Code Listener. Nowadays, it allows building analysers that are able to handle everything that GCC is able to compile in the form of GCC plug-ins. Thanks to the Code Listener infrastructure, when dealing with software natively compiled by the GCC compiler, there is no need for pre-processing the sources or changing the way they are built. There is also no need to alter source codes or `Makefile`s. Indeed, adding the `-fplugin` compiler flag into `CFLAGS` is sufficient. For more details about using plug-ins with GCC, please refer to its manual pages.

Currently, the Code Listener infrastructure is used in two prototypical analysers: Predator [20] and Forester [30]. Both of them are aimed at analysing programs with complex dynamically linked data structures. In order to illustrate the usage of the Code Listener infrastructure, its distribution comes with a simple analyser called `fwnull` that looks for null-pointer dereferences.

This chapter is organized as follows. The first part deals with the intermediate representation of a source program used by Code Listener. In the second part, the architecture of Code Listener is studied. Then, the third part covers the Code Listener API. The present chapter is based on [18, 19, 21] and our own experiences with this tool.

4.1 Intermediate Source Code Representation

This section presents a brief overview of the intermediate source code representation that Code Listener obtains by transforming the code produced by the GCC compiler. We have chosen this compiler because it is currently the only supported one.

The representation of a source code in GCC varies along the chain of compiler passes. The Code Listener infrastructure takes one of these representations called *GIMPLE*, modifies it into a more concise and easier to understand form and provides it to static analysers. GIMPLE is a machine-independent intermediate representation of source code that is based on the structure of a parse tree. It comes in two flavours, namely low-level GIMPLE and high-level GIMPLE. Code Listener uses the former type. A typical feature of this representation is using a three-address form for most operations except function calls. GIMPLE uses short circuiting for logical *and* and *or* operations and has no high-level control-flow structures, such as `for` and `while` loops. By using the `-fdump-tree-all` parameter, the `gcc` compiler generates files containing different program representations used by `gcc` during compilation. One of these files includes the GIMPLE representation of the given program. More information related to this representation can be found in [44] and [28].

Before we start describing the Code Listener's representation of a source program, it is important to realize what is necessary to represent. Since in the C language, programs are organized in functions, a CFG representation can be used to describe each function. In this description, nodes are basic blocks in the sense in which they are presented in Section 2.1, and edges represent transitions between them during code execution. Each basic block consists of several instructions. Code Listener divides these instructions into two groups, specifically terminal and non-terminal instructions. As the names suggest, a terminal instruction appears only as the last instruction in a basic block whereas a non-terminal instruction cannot appear as the last instruction of a block. The targets of terminal instructions determine the edges of a CFG. Both types of instructions utilize their *operands* to access literals, program variables or the contents of memory pointed by a program variable. The semantics of an operand can be changed via an *accessor*, which means, for example, using the address of a variable instead of the variable itself. Since a function may call other functions, it is vital to capture calling relations between functions. For this purpose, a call graph (CG) is used.

In Table 4.1, non-terminal instructions are presented. The first column shows specific instructions. Then, in the second column, a description of these instructions is provided. The third column gives the form of instructions where *dst* is the destination operand (in

Table 4.1: Non-terminal instructions

Instruction	Meaning	Form	Supported operators
`UNOP`	unary operation	$dst = \circ src$	logical not (!), bitwise not ($\sim$), unary minus ($-$), assignment ($=$)
`BINOP`	binary operation	$dst = src1 \circ src2$	comparison binary operators, arithmetic binary operators, arithmetic unary operators, logical binary operators, bitwise binary operators
`CALL`	function call	$dst = fnc\ arg1\ \ldots$	function in analysed program, external function, indirect function call

the case of the `CALL` instruction, it can be void if the function's return value is not used); $src, src1$ and $src2$ are the source operands; $\circ$ is a unary operator (in the case of the `UNOP` instruction) or it is a binary operator (in the case of the `BINOP` instruction). Note that if $\circ$ is identity in the case of the `UNOP` instruction, the instruction becomes an assignment. It is also worth to note that fnc is an operand that specifies a function to be called, and $arg1, \ldots$ are optional arguments passed to it. Furthermore, since `CALL` instructions are classified as non-terminal instructions, they are always followed by another instruction within a basic block.

Table 4.2 presents terminal instructions. The `JMP` instruction has exactly one target and no operands which means that it connects the end of one basic block to the entry of another basic block. In contrast, the `COND` instruction has one operand treated as Boolean and two targets, one denoted as the *then target* and the other denoted as the *else target*. If the operand evaluates as true, the *then target* is taken; otherwise, the *else target* is taken. The `SWITCH` instruction is a generalisation of the `COND` instruction. In contrast to the `COND` instruction, the `SWITCH` instruction has an arbitrarily long list of value-target pairs and its semantics is similar to the `switch` statement in the C language. The `RET` instruction has no targets and only a single operand that specifies the return value of a function. This operand can be void if the function has the `void` type. The `ABORT` instruction has no targets and no operands and it means termination of execution for the whole program.

Table 4.2: Terminal instructions

Instruction	Meaning
`JMP`	unconditional jump
`COND`	conditional jump
`SWITCH`	switch instruction
`RET`	return from a function
`ABORT`	abort instruction

4.2 Architecture

In this section, different parts of the Code Listener infrastructure and their functions are covered. Then, a brief overview of communication between these parts is explained and a procedure of intermediate source code construction is provided.

Let us start with describing Figure 4.1 from the left side to the right. The small boxes on the left represent source code parsers with which Code Listener communicates. As mentioned earlier, currently only support for the GCC compiler and Sparse is provided. However, the Code Listener infrastructure is designed to be extensible in the sense that other parsers may be used to obtain intermediate source code representation. The small boxes, embedded into each source code parser, are called *adapters* and are responsible for translating the intermediate code representation of the given parser into a parser-independent code representation that is unified for all parsers. The *code parser interface* represents a channel used for communication with code parsers. Behind this communication channel, *filters* and *listeners* are located. Filters perform various transformations of the intermediate source code representation gained from the code parser interface. For example, the *switch to if* block translates every `SWITCH` instruction into a sequence of `COND` instructions. On the other hand, listeners process the incoming streams of intermediate code obtained from the code parser interface and provide some diagnostic tools. For example, a CFG plotter and an intermediate-code printer are implemented as listeners.

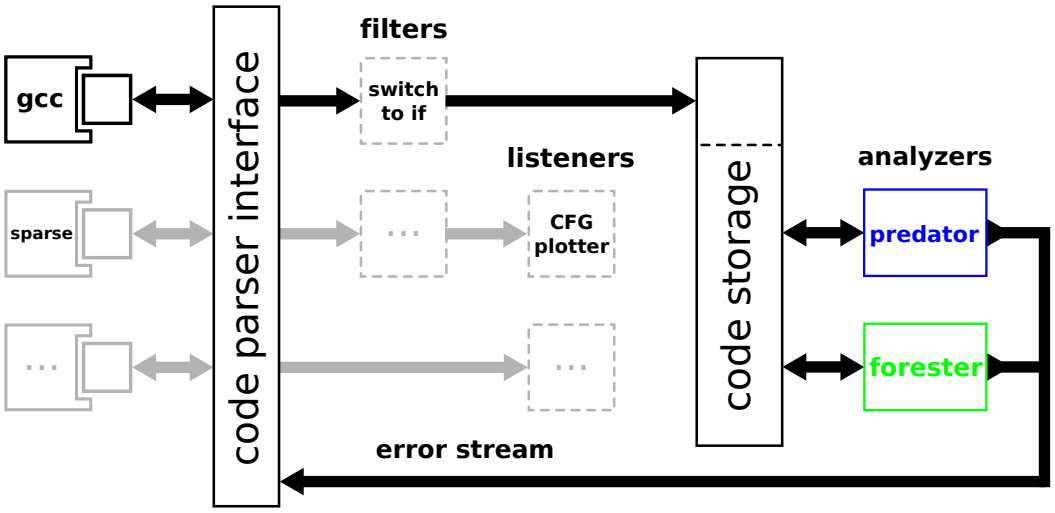


Figure 4.1: Architecture of the Code Listener infrastructure [17]

Communication between code parsers and the Code Listener infrastructure is based on callbacks. For this purpose, the address of the `handle_fnc_cfg()` function is registered by the `plugin_init()` function. Subsequently, `handle_fnc_cfg()` is called immediately as the GIMPLE code, gained from a code parser, is prepared for transformation. After that, Code Listener starts to create its own representation of the source code. For communication between Code Listener and data-flow analysers, a different approach must be introduced since a callback-based interface is not appropriate. Hence, an interface accepting callbacks and building a model of the intermediate code from them must be included. This function is performed by the *code storage* block that is in charge of creating an API and providing

it to static analysers. After code storage builds the whole object model, a static analysis can be started on this model.

4.3 Code Listener API

Since the Code Listener infrastructure serves as a communication channel between source code parsers and static analysers, two APIs are necessary to be defined. The first API is defined for communication between code parsers and Code Listener whereas the second API is defined for communication between Code Listener and static analysers. The first one is written in pure C to ensure compatibility with some parsers and the second API is written in the C++ language. Since this work focuses on developing a static analyser, the following text deals only with the API between Code Listener and static analysers.

In this section, the API provided by the code storage block, shown in Figure 4.2, is described. The functionality supplied by this API is placed in the `CodeStorage` namespace. In this namespace, the top-level data type named `Storage` is located. It gathers all available information about the analysed source code. Thus, `Storage` represents a serialised model of the source program. It contains a look-up container for types called `TypeDb`, a look-up container for variables called `VarDb` and a look-up container for functions called `FncDb`. Besides the mentioned look-up containers, `Storage` contains one look-up container for variable names, one look-up container for function names and a structure for CG used for representing the calling relationship between functions. The `TypeDb` container represents the database of types. Its items contain extensive information about types in the analysed program, such as the type's kind, location, scope, name, size, number of nested types and so on. The `VarDb` container contains all variables used in the analysed source code and information related to these variables. Every variable is represented by the `Var` class that stores, for example, the kind of a variable, location of the variable's declaration, name of the variable and so on. Since the name of the variable needs not be unique in the processed program, a unique integer identifier is assigned to each variable. The last mentioned container, `FncDb`, is used for storing information about functions. Every function is represented by the `Fnc` class that among other things provides two STL containers, the first one filled with arguments of the function and the second one filled with the variables used in the function's code. The `Fnc` class also stores a CFG of the given function. The CFG is represented by the `ControlFlow` class that allows us to iterate over its basic blocks. For the representation of a basic block, the `Block` class is utilized.

An instruction in a basic block is represented by the `Insn` class. In Section 4.1, terminal and non-terminal instructions were introduced. The former one uses a single operand and the latter uses two operands. Since operands are key elements for an instruction, it is necessary to represent these operands. For this purpose, a structure called `cl_operand` is utilized. This structure is shown in Figure 4.3. The item `code` holds the kind of an operand that can represent `void`, `unknown`, `pointer`, `structure`, `union`, `array`, `function`, `integer`, `character`, `boolean`, `enum`, `real` and `string` types. The second item `scope` is assigned to each function and variable and can hold one of three scopes, namely `local`, `global` or `static`. Since operands are statically typed, the item named `type` can be used to hold a type. Each atomic type can be defined by its kind (integer, function and so on). However, in the case of a non-atomic type, a list containing references to other types on which the non-atomic type depends must be defined. The fourth item named `accessor` represents

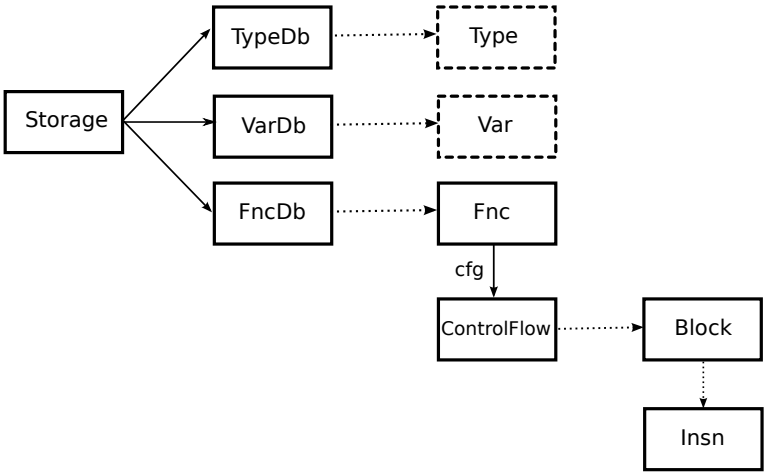


Figure 4.2: The *code storage* block [18]

a list of accessors or `NULL` if there is no accessor. This list contains the ways a variable may be used—for example, taking a reference to a variable, dereferencing a variable and so on. Note that an operand must refer to the structure `cl_var` or to the structure `cl_cst`. The first one is used if an operand refers to a variable whereas the second one is used if the operand refers to a constant, string literal or function.

```
1  struct cl_operand
2  {
3      enum cl_operand_e code;        // kind of operand
4      enum cl_scope_e scope;         // scope of the operand's validity
5      struct cl_type *type;          // type of operand
6      struct cl_accessor *accessor; // chain of accessors
7
8      union
9      {                              // per operand type specific data
10         struct cl_var *var;        // valid only for CL_OPERAND_VAR
11         struct cl_cst  cst;        // valid only for CL_OPERAND_CST
12     } data;
13 };
```

Figure 4.3: Structure for representing an operand [17]

Chapter 5

Design of the Analyser

In this chapter, we present the design of our analyser. The implementation is given in the subsequent Chapter 6. This chapter is organized as follows. First, Section 5.1 gives a high-level overview of the analyser to ease the understanding of the sections that follow it. Then, Sections 5.3 through 5.7 discuss the design of the analyser's components in detail. Finally, Section 5.8 closes the chapter by summarizing limitations of the designed analyser, mainly from the view of safety.

5.1 High-Level View

The analyser is based on the Code Listener architecture and is developed as a GCC plugin (see Chapter 4). The input of the analyser is a C source code conforming to the ISO C99 standard [24]. The output is a mapping of basic blocks of functions in the input source code into two pieces of information—the value ranges of variables on the input to the block and on the output of the block. The line numbers in the original input files are also provided so the analyst can orientate more quickly. To make the text of this and the next chapter more readable, in Appendix A, we provide a complete example of running the analyser and analysing its output.

The primary design criterion is that the analyser should be safe. This means that when it emits output, this output is an over-approximation of the actual values the variables may have. In other words, when a range for a variable `j` computed by the analysis is $\langle 1, 64 \rangle$, then in every run of the program, `j` has never a value not belonging to the interval $\langle 1, 64 \rangle$.

The supported types of variables whose range the designed analysis can compute are all integral and floating-point types from ISO C99, arrays, and structures. Other types, such pointers, unions or complex numbers are not supported (see Section 5.8).

As outlined in Chapter 3, our analysis is based on abstract interpretation. Our concrete domain is represented by numbers, and the abstract domain is that of interval ranges. As there are several integral and floating-point types in C, we need a proper representation that can capture all of these types in a unified way. Moreover, to preserve safety, we need to be able to correctly handle situations like overflows and underflows. To this end, Section 5.2 deals with the design of a class for such a unified representation of numbers. The representation of interval ranges of such numbers is the topic of Section 5.3.

In Code Listener, operands of instructions are represented by the `cl_operand` class. For example, in the instruction `a = b + c`, there are three operands: `a`, `b`, and `c`. However, apart from accessing variables, we may also access fields in structures or dereference arrays. To this end, we need a uniform representation of such accesses. We call this representation a *memory place* and discuss its design in Section 5.4. That section also covers the conversion of `cl_operand`s into these memory places.

The design of the value-range analysis itself is the subject of Section 5.5. In there, we provide details on how the analysis works and how the underlying classes, designed in Sections 5.2 through 5.4, are utilized.

Finally, the analyser uses two sub-analyses. In the first sub-analysis, discussed in Section 5.6, we address correct handling of global variables. We need to make sure that when a global variable is changed in one function, other functions that use this global variable properly reflect this fact. In the second sub-analysis, we focus on detecting loops in the analysed program. Such a detection results into an increase of both speed of the analysis and precision of the obtained results. The design of this loop analysis is addressed in Section 5.7.

5.2 Unified Representation of Numbers

All numbers are represented by the `Number` class. It can hold a value of any integral type (`char`, `short`, `int`, `long`) in both signed and unsigned versions, and of any floating-point type (`float`, `double`, `long double`). When constructing an integral number, we need to know its value, width (number of bytes), and whether it is signed or unsigned. On the other hand, when creating a floating-point number, only its value and width suffice. Indeed, floating-point types are not divided into signed and unsigned, so this piece of information is irrelevant for them.

The `Number` class handles overflows and underflows of numbers. For example, if we have `unsigned int i = UINT_MAX`, and we do `++i`, we obtain `0`. Floating-point numbers do not underflow or overflow—instead, they are set to the negative or positive infinity, respectively [24]. All these cases are handled by the class. It should be noted that the C99 standard does not say what is the result of `i++` when `i` is a signed integer [24]. However, as the common behaviour in the existing C implementations is for `i` to overflow, the class performs such an overflow instead of producing an undefined result. The same approach is done for underflows in terms of signed integers.

Another concept that is handled by the `Number` class is *integer promotion* (see Section 6.3.1.1 of [24]). The C standard assigns *conversion ranks* to integral types, and when performing an operation over an integral number whose rank is lower than the rank of `int`, it is promoted to `int`. For example, if we have `signed char c = 5` and perform `--c` over it, it is first promoted to `int`, then the operation is done, and then it is converted back to `signed char`. The `Number` class supports all the promotions, whose detailed description is omitted due to space requirements (see Section 6.3.1.1 of [24] for a full description).

Apart from integer promotions, discussed in the previous paragraph, another concept handled by the `Number` class is called *usual arithmetic conversions* (see Section 6.3.1.8 of [24]). When performing an operation over two different types, such as `signed int` and `unsigned`

`int`, their type has to be first unified so the operation can be done. To see why this is important, consider the following example:

```
1 int i = -1;
2 unsigned int j = 0;
3 if (i > j) {
4     printf("i > j");
5 }
```

When you compile and run the example, it outputs "`i > j`" even though one would expect that `-1` is lower than `0`. The reason is that before comparing `i` and `j`, the mentioned arithmetic conversions are done. First, as `i` is signed and `j` is unsigned, `i` is converted to `unsigned int`, so its value becomes `UINT_MAX` (recall that this is the unsigned value of `-1`). Then, the expression `UINT_MAX > 0` evaluates to true, so the `if` statement's body is entered.

These conversions are described in detail in Section 6.3.1.8 of [24] and apply to both integral as well as floating-point types. Once again, the `Number` class supports all of the conversions described in the referenced section of the standard. Without them, the analysis might not be safe.

Last, but certainly not least, the `Number` class supports the unary and binary operations that can be done over numbers in the Code Listener's intermediate representation. These operations are summarized in Table 5.1. First, the operation's code is given, as defined by Code Listener, then its arity (unary or binary), followed by a brief description of what the operation does. Note that the `I` in `CL_BINOP_BIT_IOR` is not a typo.

5.3 Interval Ranges and Their Representation

In the previous section, we have described the class representing standalone numbers. In this section, we move to the design of interval ranges, which form the abstract domain in terms of the abstract interpretation framework that we use.

An *interval* is a set of numbers with the property that any number that lies between two numbers in the set is also included in the set [39]. We write intervals in the form $\langle x, y \rangle$, where x is the minimal number of the set and y is the maximal number of the set. In the case of floating-point intervals, x or y may also be the negative infinity, positive infinity, or `NaN`, which is an artificial number used in the C arithmetic and defined in the IEEE floating-point standard [46]. If x or y is `NaN`, the other one has to be also `NaN`. For example, the integral interval $\langle -1, 2 \rangle$ denotes the set $\{-1, 0, 1, 2\}$, the floating-point interval $\langle -\infty, \infty \rangle$ denotes the interval of all floating-point numbers representable on the given type (excluding `NaN`), and the floating-point interval $\langle \texttt{NaN}, \texttt{NaN} \rangle$ represents the singleton set $\{\texttt{NaN}\}$.

A *range*, represented by the `Range` class, is a union of intervals. For example, the range

$$\langle -10, -8 \rangle \cup \langle 1, 4 \rangle \cup \langle 255, 255 \rangle$$

represents the set

$$\{-10, -9, -8, 1, 2, 3, 4, 255\}.$$

All intervals in a range have to be of the same type. This corresponds to the fact that every variable is of a single type so it makes no sense to consider ranges composed of intervals having different types.

Table 5.1: Operation codes in Code Listener and their meaning

Code in Code Listener	Arity	Description
CL_UNOP_ASSIGN	unary	assignment
CL_UNOP_TRUTH_NOT	unary	logical negation
CL_UNOP_BIT_NOT	unary	bit negation
CL_UNOP_MINUS	unary	minus
CL_UNOP_ABS	unary	absolute value
CL_UNOP_FLOAT	unary	cast to a floating-point type
CL_BINOP_EQ	binary	equality comparison
CL_BINOP_NE	binary	non-equality comparison
CL_BINOP_LT	binary	less-than comparison
CL_BINOP_GT	binary	greater-than comparsion
CL_BINOP_LE	binary	less-than-or-equal comparison
CL_BINOP_GE	binary	greater-than-or-equal comparison
CL_BINOP_TRUTH_AND	binary	logical and
CL_BINOP_TRUTH_OR	binary	logical or
CL_BINOP_TRUTH_XOR	binary	logical exclusive or
CL_BINOP_PLUS	binary	addition
CL_BINOP_MINUS	binary	subtraction
CL_BINOP_MULT	binary	multiplication
CL_BINOP_EXACT_DIV	binary	integral division without a remainder
CL_BINOP_TRUNC_DIV	binary	integral division with truncation
CL_BINOP_TRUNC_MOD	binary	integral modulus with truncation
CL_BINOP_RDIV	binary	floating-point division
CL_BINOP_MIN	binary	minimum
CL_BINOP_MAX	binary	maximum
CL_BINOP_BIT_AND	binary	bit and
CL_BINOP_BIT_IOR	binary	bit or
CL_BINOP_BIT_XOR	binary	bit exclusive or
CL_BINOP_LSHIFT	binary	bit left shift
CL_BINOP_RSHIFT	binary	bit right shift
CL_BINOP_LROTATE	binary	bit left rotate
CL_BINOP_RROTATE	binary	bit right rotate

Ranges support all the operations that are possible to perform over numbers (see Table 5.1). However, as we are computing in the abstract domain, instead of the domain of concrete numbers, the inputs to these operations are ranges. For example, consider the following line of C code:

```
a = b + c;
```

Let `a`, `b` and `c` be three integral variables having the ranges of possible values r_a, r_b and r_c. Then, we compute the result of $r_b + r_c$ and assign it to a new range r'_a, which denotes the range of a after the statement is performed. For example, if $r_b = \langle 1, 5\rangle$ and $r_c = \langle 6, 8\rangle \cup \langle 10, 10\rangle$, then $r'_a = \langle 7, 15\rangle$. As we do not know the exact values of b and c, to preserve safety, we have to consider all the possibilities, thus obtaining $\langle 7, 15\rangle$ instead of, for example, 11.

Apart from performing operations over ranges, the `Range` class supports the so-called *trimming* of ranges. This is best to be explained on an example. Consider two ranges, $r_x = \langle 1, 10\rangle \cup \langle 20, 30\rangle$ and $r_y = \langle 15, 25\rangle$, and the following conditional statement:

```
1 if (x < y) {
2     // ...
3 }
```

Assume that on line `1`, `x` has assigned the range r_x and `y` has assigned the range r_y. Then, during the value-range analysis, when computing the input to the statement on line `2`, we know that `x < y` has to hold. Therefore, we may appropriately trim the ranges r_x and r_y. In this case, on line `2`, the range for x would be $\langle 1, 10\rangle \cup \langle 20, 25\rangle$, and the range for y would be $\langle 15, 25\rangle$, i.e. unchanged. As we do not know the exact values of x and y, to preserve safety, we have to assume that they can be anything from their respective ranges. Moreover, the trimming also works correctly if `x` and `y` have different types—in such a case, usual arithmetics conversions (see Section 5.2) are performed, the ranges are trimmed, and the resulting ranges are converted back to the types of `x` and `y`, respectively.

When an operation over ranges either cannot be computed precisely or it would include an operation that may result into an undefined behaviour, such as division of zero in terms of integral ranges, we over-approximate the resulting range. This ensures that the result of the operation is an over-approximation, thus preserving safety. For example, when dividing $\langle 1, 10\rangle$ by $\langle -1, 5\rangle$, the resulting range is $\langle \text{MIN}, \text{MAX}\rangle$. Otherwise, we would not know what the resulting range should be. For example, on GCC 4.8 and Clang 3.3, `x / 0` for an integer variable `x` terminates the program with an exception.

Last, the `Range` class supports interval normalization. If we have an integral range formed by the intervals $\langle 1, 4\rangle$, $\langle 3, 7\rangle$ and $\langle 8, 10\rangle$, it is normalized into $\langle 1, 10\rangle$. The reason of joining $\langle 3, 7\rangle$ with $\langle 8, 10\rangle$ is that there is no integral number between 7 and 8.

5.4 Memory Places and Conversion of Operands To Them

Consider the statement `a = b;`. This statement is represented in Code Listener as an instruction formed by an assign operation having two operands—the source operand (`b`) and the destination operand (`a`). In Code Listener, every operand is represented as an instance of `cl_operand`. If the operand is just a simple variable, like in the case above, then the operand simply stores its type and UID—a unique identifier. However, operands of the form `x.y.z`, `s[10]`, or even `x.y[5].t` are more complex. What we need is a uniform form of representation of these operands so that even if they appear on several places in the program, we will get the same representation. This brings us to the idea of memory places.

A *memory place* is a uniform representation of locations in memory into which something can be written or from which something can be read. It is represented by the class `MemoryPlace` and supports the following three operations:

- Getting a string representation of the memory place. This is used when the output from the analyser is printed. For example, if there is an instance of a structure called `d` with a field named `i`, then its string representation is `d.i`.

- Distinguishing between variables that are actually in the input source code from artificial variables introduced by GCC. When emitting the output of the analysis, the ranges of artificial variables may be discarded because they are not present in the original source code.
- Checking if the memory place represents an array. Due to some limitations imposed on the analyser (see Section 5.8), when assigning something into an array, we treat the array as a single variable. Therefore, for an array `s`, `s[10]` and `s[1]` should be represented by a single memory place.

The primary use of memory places is to have an entity with which we associate ranges during the value-range analysis. Moreover, we need a class that can convert an instance of `cl_operand` into an appropriate memory place. This class is named `OperandToMemoryPlace` and works as follows:

(1) If the operand is just a variable, it uses its UID to convert it into a memory place.

(2) If the operand is an array dereference, like `s[5]`, it uses the pair `(UID, 0)`, where `UID` is the UID of the array, and `0` marks that the array was dereferenced. As we have already pointed out, we merge all array dereferences into a single dereference so `s[1]` and `s[10]` are indistinguishable and both map to `(UID, 0)`.

(3) If the operand is an access to a field of a structure of the form $\texttt{x.y}_1\texttt{.}\cdots\texttt{.y}_n$ for some $n \geq 1$, we use the tuple `(UID, offset1, offset2, ...)`, where `UID` is the UID of `x`, and `offsetX` is the offset of the `X`th variable in the structure, where $1 \leq \texttt{X} \leq n$. For example, given the following structure

```
struct A {
    int i;      // offset 0 (in A)
    double d;   // offset 1 (in A)
    struct B {
       char c; // offset 0 (in B)
       int b;  // offset 1 (in B)
    } a;        // offset 2 (in A)
} x;
```

where the UID of `x` is 124, the operand `x.a.b` is represented as `(124, 2, 1)`.

(4) If the operand is a combination of structure accesses and array dereferences, like `a.b.c[10].e`, we combine (2) and (3) to get a proper memory place.

Having described numbers, ranges, and memory places, we can move to the heart of this work—the value-range analysis itself.

5.5 Value-Range Analysis

The main part of our value-range analysis is represented by class `ValueAnalysis`. The analysis is intraprocedural (see Section 2.4), which means that all functions are analysed in isolation. This implies the top-level algorithm, shown in Algorithm 2.

The preliminary analyses on line 1 deal with global variables and loops, and are discussed later in Sections 5.6 and 5.7. Then, on lines 2 through 6, we compute the ranges for all the

Algorithm 2: Top-level value-range analysis algorithm

Input: intermediate representation of the input C program
Output: value ranges of variables at the input and output of all basic blocks in the program

```
1 preliminary analyses;
2 foreach function func in the program do
3 |   if func has a body then
4 |   |   compute the ranges for func;
5 |   end
6 end
7 print the computed ranges;
```

basic blocks in every function, and we end the algorithm by printing the computed ranges (line 7). The output format is described in Section 6.9, so in the remainder of this section, we focus only on the computation of ranges.

The computation of ranges for a function is based on the work list algorithm (see Section 2.4). The computed values are propagated along with the control flow—that is, the underlying data-flow analysis utilizes a forward-flow direction (see Section 2.1). To every basic block `block` of the function, we assign two mappings. The first one, `IN[block]`, maps variables to ranges that they have when they enter the basic block. The second mapping, `OUT[block]`, stores the ranges of variables at the end of the basic block, i.e. on its output. The main idea behind the computation is to keep traversing over the basic blocks, re-computing the ranges, and checking which ranges change. When there is no change, i.e. a fixed point has been reached, we end the computation. This idea is expressed in Algorithm 3.

Algorithm 3: Computation of value ranges for input and output of all basic blocks in the given function

Input: intermediate representation of a function `func` and its control-flow graph
Output: value ranges of variables at the input and output of all basic blocks in the function

```
1  create empty mappings IN[block] and OUT[block] for every block of func;
2  create an empty todo queue;
3  put the first basic block of func into todo;
4  while todo is not empty do
   |   // There is a basic block to be processed.
5  |   remove the front block from todo and store it to block;
6  |   store a copy of OUT[block] into oldRanges;
7  |   perform the value analysis of block to compute new IN[block] and OUT[block];
8  |   if oldRanges differs from OUT[block] then
   |   |   // The output ranges have changed.
9  |   |   schedule the successors of block for further processing;
10 |   end
11 end
```

On line 1, we create the two mappings for every basic block of the function. Initially, they are both empty. Lines 2 and 3 initialize the `todo` queue. Then, we keep iterating over the `todo` queue, until it becomes empty. During every iteration, we pop the current basic block, backup the old ranges on the output of the block, perform the analysis for the block, and when the new output ranges differ from the old ranges, we schedule the successors of the block for further processing. First, let us give the scheduling algorithm, expressed in Algorithm 4.

Algorithm 4: Scheduling the successors of a basic block for further processing

Input: a basic block `block` of a function `func`, the control-flow graph of `func`, and the `todo` queue from Algorithm 3
Output: updated `todo`

```
1 foreach successor succ of block in func do
2 |   if succ is not in todo then
3 |   |   enqueue succ to todo;
4 |   end
5 end
```

In the algorithm, we traverse all the successors of the block by using the control-flow graph. If the successor is not already in the `todo` queue, we enqueue it.

Next, we describe the analysis of a basic block, which is done on line 7 in Algorithm 3. This analysis is expressed in Algorithm 5.

Algorithm 5: Value analysis of a basic block

Input: a basic block `block` of a function `func`, the control-flow graph of `func`, and the two mappings `IN` and `OUT` for `block` and all its predecessors
Output: new `IN[block]` and `OUT[block]`

```
1 update IN[block] based on the output ranges of all predecessors;
2 clear OUT[block];
3 foreach instruction insn in block do
4 |   update OUT[block] based on insn;
5 end
```

First, on line 1, we update the input ranges of the block, which we describe shortly. Then, we clear the output mapping in order to compute the new one based on the updated input ranges. After that, in a loop on lines 3 through 5, we inspect all instructions and update the output ranges based on the operations the instructions do. Before presenting the algorithm, let us first turn our attention on the update of input ranges, done on line 1.

What we have to do is to compute the new input ranges for the block based on the output ranges from the block's predecessors. Additionally, however, we utilize the computation of trim ranges as described in Section 5.3. This leads to more precise results. Before describing the algorithm in pseudocode, we illustrate it by an example. Consider the control-flow graph from Figure 5.1, where there is a block `block` with two predecessors, `pred1` and `pred2`.

Our task is to compute the range for `i` in `IN[block]`. Assume that the range for `i` in `OUT[pred1]` is $\langle -100, 100\rangle$, and that the range for `i` in `OUT[pred2]` is $\langle 1, 10\rangle$. Since the `pred1` block ends with `i < 5`, the trim range for `i` is $\langle \texttt{INT\_MIN}, 4\rangle$. The intersection

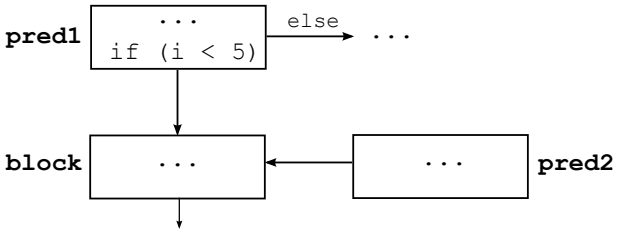


Figure 5.1: A sample control-flow graph to illustrate Algorithm 6

of $\langle -100, 100\rangle$ and $\langle$`INT_MIN`$, 4\rangle$ is $\langle -100, 4\rangle$. Therefore, the range for `i` at the input of `block` is

$$\langle -100, 4\rangle \cup \langle 1, 10\rangle = \langle -100, 10\rangle.$$

Notice that we have to take into account the output of all predecessors because at the input of `block`, `i` might have the values from the output of both `pred1` and `pred2`.

The above-described update of input ranges is given in Algorithm 6.

Algorithm 6: Update of input ranges for a basic block

Input: a basic block `block` of a function `func`, the control-flow graph of `func`, and the two mappings `IN` and `OUT` for `block` and all its predecessors
Output: updated `IN[block]`

```
1  create a new empty mapping NEW_IN[block];
2  foreach predecessor pred of block do
3  |  compute the trim ranges for variables in OUT[pred];
4  |  foreach variable var in OUT[pred] do
5  |  |  if there is a trim range for var then
6  |  |  |  store the trim range for var into varTrimRange;
7  |  |  |  store the range for var from OUT[pred] into varOutRange;
8  |  |  |  add the intersection of varTrimRange and varOutRange to NEW_IN[block];
9  |  |  else
10 |  |  |  add the range for var from OUT[pred] to NEW_IN[block];
11 |  |  end
12 |  end
13 end
14 merge NEW_IN[block] into IN[block];
```

Let us argue about the necessity of the last step of Algorithm 6, done on line 14. Our value-range algorithm is based on a fixed point computation. To this end, if we just assigned `NEW_IN[block]` to `IN[block]`, we might reduce the ranges that are already in `IN[block]`, which might result into an infinite loop in the analysis. Therefore, we need to unite the ranges in `NEW_IN[block]` with those that are already in `IN[block]`.

Now, the only algorithm to be presented is the update of output ranges by inspecting all instructions of a basic block. This is done on line 4 of Algorithm 5. The idea behind the algorithm is to call an appropriate operation over ranges, depending on the type of the instruction and on its operands, and update the output of the block. This is described in Algorithm 7.

Algorithm 7: Update of `OUT[block]` based on the given instruction

Input: a basic block `block`, an instruction `insn` in the block, and the current mapping `OUT[block]`
Output: updated `OUT[block]`

```
1 switch type of insn do
2 |   convert all operands of insn into memory places (or constants);
3 |   perform the operation over ranges associated to these memory places;
4 |   update the range of the destination memory place;
5 endsw
```

First, depending on the type of the instruction, we convert all of its operands into memory places by using the `OperandToMemoryPlace` class (see Section 5.4). If some of the operands are literals, like floating-point or integral constants, we convert them directly into an instance of `Number` because literals are not represented by memory places (there is no need for that). Then, by using the appropriate function from `Range` (see Section 5.3 and Table 5.1), we compute the resulting range after performing the operation over ranges that are associated to the converted memory places (or `Number` instances). After that, we update the range of the destination variable of the instruction by modifying `OUT[block]`.

Consider the following statement, which Code Listener represents by an instruction of type `CL_INSN_BINOP` with subcode `CL_BINOP_PLUS`:

```
c = x.i + 5;
```

Let us assume that the type of `c` is `int`, and that `x` is a structure with its first field being `int i`. Then, the algorithm converts `c` and `x.i` into their respective memory places, and gets the current range of `x.i`. For example, let the range of `x.i` be $\langle 5, 10\rangle$. The literal `5` is converted into the interval $\langle 5, 5\rangle$. By using the addition operation over ranges, whose implementation is given later in Section 6.2, the result of $\langle 5, 10\rangle + \langle 5, 5\rangle$ is $\langle 10, 15\rangle$. This result is then assigned to the memory place of `c` and stored into `OUT[block]`.

To preserve safety, if there is a function call of the form `dst = func()`, then we set the range of `dst` to be the maximal range. The reason is that since we do an intraprocedural analysis, we do not know what is the exact range returned from `func()`. Moreover, if there appears an operand without some already assigned range, we also use the maximal range. This corresponds to the fact that in C, the value of uninitialized local variables is undefined [24].

Having described the design of the value range-analysis, we move to describing two analyses that are used to improve the precision of the obtained results.

5.6 Global Variables Analysis

Since the value-range analysis is intraprocedural, meaning that every function is analysed in isolation, we need to make sure that accesses to global variables are handled correctly. For example, if a global variable is changed in function `A`, it may affect the value of other variables in function `B`. We solve this in the following way.

Before the main part of the value-range analysis itself, we perform another analysis, represented by class `GlobAnalysis`, in which we compute which global variables may be changed when running the program. This is done by going through all the instructions in program and checking which assignments have global variables on their left-hand sides.

Based on the above analysis, if a global variable is never modified, only read, then we can use the initializer as its range. Otherwise, if the variable may be modified, we over-approximate its value to the maximal range of its type. In this way, we preserve the safety property of our analyser.

This analysis is done as part of line 1 in Algorithm 2.

5.7 Analysis of Loops

Consider the following program, consisting of a simple `for` loop:

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int c = 1;
6     for (int i = 0; i <= 10; i++) {
7         c = 2 * c;
8     }
9
10    printf("%d\n", c);
11    return 0;
12 }
```

By applying the value-range algorithm as described in Section 5.5, we would get that the range of `c` at the end of the function is $\langle$`INT_MIN`, `INT_MAX`$\rangle$. The reason is that we keep iterating until there are no changes, and `c` changes until it reaches the maximal range. This is not very helpful as this is the maximal range of `int`—the type of `c`. Instead, what we would like to have is the information that `c` changes at most the number of times `i` changes. This is the task of the analysis that is described next.

The `LoopFinder` class analyses the program to find loops and tries to compute their *trip count*—the number of iterations of the loop under analysis. To compute it, we need the following information:

- The so-called *induction variable*, which is the variable over which we are iterating. In our example, this is `i`.
- The initial value of the induction variable. In our case, it is `0`.
- The end value of the induction variable. This is `10` in the example above.
- The step, which gives us the information on how the induction variables changes after every iteration. In our example, it is `+1`.

All the pieces of information above can be checked by traversing the control-flow graph. We illustrate the process on the example above. Its control-flow graph is shown in Figure 5.2.

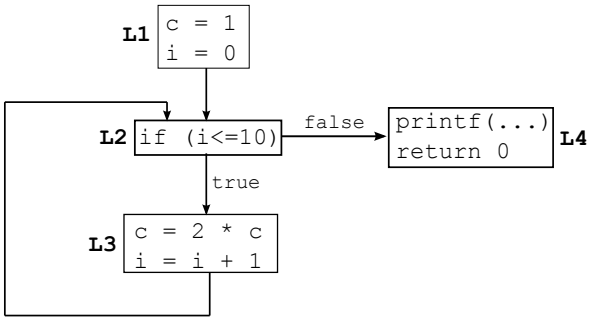


Figure 5.2: Control-flow graph of the example at the beginning of Section 5.7

The only viable start of a loop is `L2`, which tests whether `i <= 10`. When traversing its successors, we immediately find that it is indeed a loop—the only successor of `L3` is `L2` and `L4` is a return from the function. The induction variable has to be `i` because it is the only variable tested in the condition of `L2`. To check that `i` is indeed the induction variable, we traverse the predecessor of `L2`, which is `L1`. Since it contains `i = 0`, the initial value is `0`. From the condition of `L2`, we have that the end value is `10`. Since the only increment of `i` is in `L3`, we see that the step is `+1`.

Based on the above analysis, we have that the trip count of the found loop is `11`. This means that the loop is executed `11` times, and thus `c` is changed `11` times. This gives us a more precise value of `c`. Of course, this was just a simple example to demonstrate the concept. In reality, the structure of the loop may be more complex. This analysis is done as part of line 1 in Algorithm 2. Moreover, in Algorithm 3, we add a limitation of the number of passes for blocks which form a loop with a known trip count.

5.8 Limitations

As mentioned earlier in this chapter, the primary design criterion is that the analyser should be safe. Moreover, considering the limits of existing tools for value-range analysis and research papers on this topic (see Chapter 3), it is obvious that designing and implementing a value-range analyser that handles all the imaginable cases (in addition, over the course of just two semesters), is impossible. Therefore, we had to impose some restrictions on the inputs in order for the analyser to work correctly and produce safe output. This section summarizes such limitations.

The input C program has to satisfy the following restrictions for the analyser to work correctly.

- **No indirect manipulation of data.** Variables may be modified only directly, i.e. without using pointers. Indirect calls by means of pointers to functions, pointer arithmetic and work with dynamically allocated memory are unsupported as well.
- **No unsafe type conversions.** There may not be any conversions that the bypass the C type system in an undefined or implementation-defined way. For example, there may not be any conversions between integers and pointers, or using unions to store some value to a field `X` and then reading the value from another field `Y`.

- **No unions.** The union data type is not supported.
- **No complex numbers**. Code Listener does not support complex numbers, available since C99 (see Section 7.3 of [24]), and so neither does the analyser.
- **No partial initializations.** Variables of a structure or array data type have to be either fully initialized or left completely uninitialized.

When the input program uses the above features of the C language, the analyser may not work properly or its output may not be safe.

Chapter 6

Implementation

In this chapter, we describe the implementation of our analyser, whose design is given in Chapter 5. The analyser is implemented in C++, and conforms to the ISO C++ 1998 standard [23]. The ways of testing the implementation and an evaluation of the obtained results are the subjects of Chapter 7.

The present chapter is organized as follows. Section 6.1 describes the implementation of class `Number`, which is a unified representation of numbers. Then, Section 6.2 presents how ranges, represented by class `Range`, are implemented. After that, Sections 6.3 and 6.4 discuss the representation of memory places and a conversion of Code Listener's operands to them, respectively. Section 6.5 presents the implementation of the main part of our value-range analysis. The analysis of global variables, represented by class `GlobAnalysis`, is the subject of Section 6.6. Section 6.7 deals with the implementation of loop analysis. The implementation of various supportive functions is discussed in Section 6.8. Section 6.9 describes the interface and output format of the analyser. Finally, Section 6.10 closes the chapter by giving several metrics regarding the implementation.

Let us note that due to space constraints, we focus only on the fundamental implementation details. Other parts of the implementation can be seen in the heavily commented sources of the analyser.

6.1 `Number`: Unified Representation of Numbers

The `Number` class, designed in Section 5.2, is implemented in files `Number.{cc,h}`. As an internal storage for integral types, such as `char` or `long`, we use arbitrary-width integers from the GMP library [27]. More specifically, we utilize `mpz_class`, which is a C++ wrapper class around the corresponding integral GMP type. This choice of the underlying type simplifies the operations over integers as we do not have to explicitly watch for overflows or underflows, Instead, instances of `mpz_class` simply increase their width, and after the operation is done, we fit them into the width of the actual integral type the number represents. The details will be explained later.

To represent floating-point numbers, however, we directly use `long double`. The reason for not choosing `mpf_class`, which is the floating-point equivalent of `mpz_class` from GMP, is twofold. First, `mpf_class` does not support special values such as `NaN` or infinities. Second,

the semantics of several operations differ from their semantics in C. Here, the underflow or overflow is not an issue because `long double` never overflows [23]. Instead, it becomes the negative or positive infinity. Moreover, the floating-point semantics in C++98 matches the one in C99, so we can use this representation. As with integral types, if the actual type differs from `long double`, after performing an operation, we fit the number into its actual width.

Integral numbers are stored into the `intValue` member variable, and floating-point numbers are stored in `floatValue`. To distinguish between integral and floating-point types, we use the following enumeration:

```
enum Type {INT, FLOAT} type;
```

As `intValue` is of type `mpz_class`, which is an object, we cannot use a union because C++98 prohibits that [23]. Alongside with the actual value, we store the actual width of the type, provided by Code Listener, and in terms of integral values, their sign (stored in a boolean `sign` flag).

Upon creation, we set the number's type, value, width, and signess (in the case of an integral type). Then, we set the internal limits by calling `setIntLimits()` or `setFloatLimits()`. What these functions do is that they compute the maximal (non-infinite) numbers that can be represented on the given type, width, and with the specified signess. Then, we call `fitIntoBitWidth()`, which makes sure that the given value fits into the specified type. If not, it performs conversions, such as overflows or underflows that would be done in C to make the number fit. This function is also called after performing operations over numbers because the numbers may overflow.

Apart from the functions above, the class provides many supportive functions, such as `getBitWidth()`, `isIntegral()`, `isSigned()`, `isFloat()`, `isInf()`, `isMin()`, `isMax()`, `toBool()`, and many more. To convert between integral and floating-point values, the static `floatToInt()` and `intToFloat()` functions may be used. The `assign()` function enables us to simulate the assignment of a number into another number, just like it would be done in C.

The concept of integer promotion, briefly mentioned in Section 5.2, is implemented as described in Section 6.3.1.1 of [24]. This is done in function `integralPromotion()`. Usual arithmetic conversions, mentioned in Section 5.2, are implemented in `extensionByCRules()` according to Section 6.3.1.8 of [24].

The core of the class is the implementation of all the operations from Table 5.1. As our space in this work is limited, we only focus on the implementation of three operations—addition, unary minus, and bit right shift.

- *Addition.* The addition of two numbers is performed in `operator+()`, which, as its name suggests, is the overload of the `+` operator. First, we have to perform the usual arithmetic conversions of the numbers by calling `extensionByCRules(op1, op2)`. Now, both of the operands have the same type. Then, depending on whether we are adding two integers or two floating-point numbers, we perform the addition either over `mpz_class` or `long double`. Finally, as there might have been an overflow (or the value is supposed to be infinity when dealing with floating-point numbers), we call `fitIntoBitWidth()`. As we are performing the floating-point addition over `long`

`double`, it correctly handles all the special cases, such as adding positive infinity with `NaN`.

- *Unary minus.* The unary minus of a number is done in `operator-()` having a single operand (the two-operand version is binary subtraction). If the operand is a floating-point number, we simply use the unary minus over `long double`, and return the result. Otherwise, if the operand is an integral number, we call `integralPromotion()` to perform integral promotions. Then, we use the `operator-()` operation from `mpz_class`, call `fitIntoBitWidth()` to make sure the result fits into the width, and return the result. Notice that in the case of floating-point numbers, we do not have to call `fitIntoBitWidth()`. The reason is that `-x` for a floating-point value `x` is always representable on the same number of bits as `x` because they differ only on the value of the sign bit [46].

- *Bit right shift.* The bit right shift over two numbers is done in `bitRightShift()` (for bit and logical operations, we do not use the `operatorX` functions because for example, there is no operator for logical non-equivalence (xor) in C++ [23]). The `bitRightShift()` function calls `performShift(op1, op2, false)`, where the last `false` denotes the fact that we are doing a right shift. The reason is that the implementation of left and right shifts is similar so we use a single function for it. First, we check whether both operands are integral numbers (bit shifts are defined only on integers, so it is an error to call the function over floating-point numbers). Moreover, when the value of the right operand is negative, the result is undefined [24], so in this case, we also do not perform the operation. This is appropriately handled in the `Range` class, where in such a case, we return the maximal range to capture the fact that the result may be any value. After this, we promote both the operands by calling `integralPromotion()`, which is what the standard requires [24].

  If the left operand is negative, the standard leaves implementation-defined whether the shift is arithmetical or logical [24]. Therefore, instead of performing the shift over `mpz_class`, we first convert the operands into appropriate integers (based on their width), perform the shift over them, and convert them back to `mpz_class`. As the behaviour of bit right shifts of C++98 matches the one of C99 [23, 24], this solution is correct.

  After performing the operation, we call `fitIntoBitWidth()` to fit the result into its width, and we return it.

6.2 `Range`: Representation of Ranges

The `Range` class, designed in Section 5.3, is implemented in files `Range.{cc,h}`. To represent an interval, we use

```
typedef std::pair<Number, Number> Interval;
```

Then, as a range represents the union of intervals, we implement it as

```
std::vector<Interval> data;
```

It is a private attribute of `Range`. The class provides the following set of constructors:

```
Range();
explicit Range(Number n);
explicit Range(Interval i);
Range(Interval i1, Interval i2);
Range(Interval i1, Interval i2, Interval i3);
Range(Interval i1, Interval i2, Interval i3, Interval i4);
```

The default constructor is provided for convenience, e.g. when constructing an empty range—no operation works on empty ranges. The second constructor is simply a briefer way of writing `Range(Interval(n, n))`. The class supports creation of ranges having up to four intervals. When having more intervals, they have to be added manually after the construction.

When a range is created, either by the user or after performing an operation over other ranges, the `normalize()` function is called. It performs the following actions:

- When there are more intervals of the form $\langle \texttt{NaN}, \texttt{NaN} \rangle$, only one of them is kept. Moreover, if this interval exists, it is always the first interval in the range. This simplifies many operations.
- Intervals of the form $\langle x, y \rangle$, where $x > y$, are converted to the union $\langle \texttt{MIN}, y \rangle \cup \langle x, \texttt{MAX} \rangle$. For example, the interval $\langle 100, 1 \rangle$ on 8 bits (signed) is converted into $\langle -128, 1 \rangle \cup \langle 100, 127 \rangle$, which is the correct range for such input. Otherwise, the range would be invalid.
- The intervals are sorted, and when possible, joined together. As suggested in Section 5.3, if we have an integral range formed by the intervals $\langle 1, 4 \rangle$, $\langle 3, 7 \rangle$ and $\langle 8, 10 \rangle$, it is normalized into $\langle 1, 10 \rangle$.

The `Range` class provides many supportive functions. Among them belong functions for iterating over the intervals, `size()`, `empty()`, `operator[]`, `containsZero()`, `isIntegral()`, `isFloatingPoint()`, `containsOnlySingleNumber()`, `getMax()`, `getMin()`, and many others. To make the union and intersection of two ranges, `unite()` and `intersect()` can be used. To assign a range into another range while converting the intervals of the source range into the type of the destination range, there is the `assign()` function.

As in the case of `Number`, the core of the class is the implementation of all the operations from Table 5.1. In contrast to `Number`, these operations are performed over ranges rather than over single numbers. As our space in this work is limited, we only focus on the implementation of two operations—addition and logical and.

- *Addition.* This operation has two input ranges, `r1` and `r2`. The basic idea behind the implementation is to iterate over both of these ranges, and compute the addition of every interval in `r1` with every interval in `r2`. This is done in two nested `for` loops. Let $\langle \texttt{x}, \texttt{y} \rangle$ and $\langle \texttt{z}, \texttt{w} \rangle$ be the two current intervals. Then, we include the following five intervals into the resulting range:

  $$\langle \texttt{x} + \texttt{z}, \texttt{y} + \texttt{w} \rangle \qquad \langle \texttt{y} + \texttt{z}, \texttt{y} + \texttt{w} \rangle \qquad \langle \texttt{w} + \texttt{x}, \texttt{w} + \texttt{y} \rangle$$
  $$\langle \texttt{x} + \texttt{z}, \texttt{x} + \texttt{w} \rangle \qquad \langle \texttt{z} + \texttt{x}, \texttt{z} + \texttt{y} \rangle$$

  These intervals cover all possible numbers that can arise when adding numbers from the two input ranges. Finally, we call `normalize()` on the result, which takes care of possible overflows, and we return the result.

- *Logical and.* As in the case of addition, this operation, implemented in `logicalAnd()`, takes two ranges, `r1` and `r2`. First, we create two numbers that represent the values 0 and 1. Then, we create the resulting range by performing the following two actions:
  - If `r1` or `r2` contains a number that evaluates to `false`, we include the interval $\langle 0, 0 \rangle$ into the result. This check is done by calling `containsFalse()` on the two ranges.
  - If both `r1` and `r2` contain a number that evaluates to `true`, we include the interval $\langle 1, 1 \rangle$ into the result. This check is done by calling `containsTrue()` on the two ranges.

  Before returning the range, we call `normalize()` to join the intervals $\langle 0, 0 \rangle$ and $\langle 1, 1 \rangle$ to $\langle 0, 1 \rangle$ (if they are both present in the range).

In several operations, we need to return an over-approximation. In this case, there are the `overApproximateUnaryOp()` and `overApproximateBinaryOp()` functions. The actual choice depends on the number of operands of the performed operation.

Finally, we discuss the implementation of range trimming, whose example is given in Section 5.5. In what follows, we only mention how the trim range for `x` in `x <= y` is computed. For the implementation of other relational operators, see the source code. Moreover, when computing the trim range for `y`, we can use the converse relational operation—that is, for `y` in `x <= y`, we would compute the trim range from `y > x`. In this way, it suffices for our trim-range-computing functions to return the trim range for the first operand.

The trim range for `x` in `x <= y` is computed in `computeRangeForLtEq()`, which takes two ranges, for `x` and `y`, as its input. First, we perform the usual arithmetic conversions over both ranges, which is a generalization of such conversions from the `Number` class (see Section 6.1). After that, both ranges are of the same type. Then, we iterate over the intervals in the first of the ranges after the conversions, and perform the following actions. Let `maxR2` be the maximal number of the second range and let $\langle$`i`, `j`$\rangle$ be the current interval from the first range.

- If both `i` and `j` are lower or equal to `maxR2`, we include the interval $\langle$`i`, `j`$\rangle$ into the result.
- If `i` is lower or equal to `maxR2` but `j` is greater than `maxR2`, we include the interval $\langle$`i`, `maxR2`$\rangle$ into the result.
- Otherwise, we know that `i` is greater than `maxR2`, so we stop the iteration since no more intervals can be added into the result.

After that, we use `assign()` and `normalize()` to convert the resulting range to the type of the first input range, and return the result.

6.3 `MemoryPlace`: A Representation of Memory Places

The `MemoryPlace` class, whose design is given in Section 5.4, is implemented in files `MemoryPlace.{cc,h}`. Its constructor simply takes a string representation of the memory place, and the information whether it corresponds to an artificial variable or not.

These properties can be obtained by calling `asString()` and `isArtificial()`, respectively. Moreover, the `representsElementOfArray()` function returns `true` if the memory place represents an element of an array. This check is done by searching for "`[]`" in the memory place's name.

6.4 `OperandToMemoryPlace`: From Operands To Memory Places

The conversion of operands into memory places is represented by `OperandToMemoryPlace`, a class designed in Section 5.4, and implemented in files `OperandToMemoryPlace.{cc,h}`. This class provides a static function `convert()` that is able to convert the given `cl_operand` into a pointer to `MemoryPlace`.

As described in Section 5.4, to uniquely represent a memory place, a tuple of numbers is used. To implement tuples, we use `std::vector<Int>`, `typedef`ed as `UidVector`, where `Int` is a `typedef` to `mpz_class` (an integer representation from the GMP library, see Section 6.1). Then, there is a static map `std::map<UidVector, MemoryPlace*> memoryPlaceMap` that maps such a vector into an existing memory place. If for a vector there is no memory place, it means that no memory place has been created, so we create one. Otherwise, the `convert()` function can immediately return the proper memory place.

The `convert()` function implements the conversion algorithm, given in Section 5.4. It takes a `cl_operand` and, optionally, a list of indexes. This list is represented by `std::deque<int>`. The reason for choosing this type of a container is that it supports random access and has fast adding and removing of elements from both sides. When the function is called with an empty list of indexes, `convertSimpleOperand()` is called, which means that the operand is either just a variable, an element of an array, or a field in a structure. The indexes for the `UidVector` key to `memoryPlaceMap` are computed by traversing the accessors of the operand, available through `operand->accessor`.

Otherwise, if the input list of indexes, passed as `std::deque<int>`, is non-empty, it means that the operand is a complete structure, not just a single field of a structure. Alongside with the indexes, the operand's type is used to properly construct the needed `std::deque<int>` list to index `memoryPlaceMap`. The reason for distinguishing between a structure access from a complete structure is that when a complete structure is passed, there are no accessors in the Code Listener's representation, so the only way to navigate is through the operand's type and the input list of indexes.

6.5 `ValueAnalysis`: Value-Range Analysis

The `ValueAnalysis` class, whose design is given in Section 5.5, is implemented in files `ValueAnalysis.{cc,h}`. Among many private functions, it provides two public static functions: `computeAnalysisForFnc()`, which performs the value-range analysis over the given function, and `printRanges()`, which prints the result of the analysis into the given stream.

Algorithm 2 from Section 5.5 is implemented in file `vra.cc` (see also Section 6.9). In there, `computeAnalysisForFnc()` is called for every function definition in the input program. The type of the maps `IN[block]` and `OUT[block]`, which store the assignment of memory places to their ranges, is

```
typedef std::map<const MemoryPlace*, Range>
    MemoryPlaceToRangeMap;
```

Then, the `IN` and `OUT` maps have the following type:

```
typedef std::map<const CodeStorage::Block*, MemoryPlaceToRangeMap>
    BlockToResultMap;
```

The `todo` queue is represented by

```
typedef std::queue<const CodeStorage::Block *>
    SchedulerQueue;
```

and

```
typedef std::set<const CodeStorage::Block *>
    SchedulerSet;
```

The reason for using two types is that when checking whether a basic block is already in the queue, we need only $O(\log(n))$ comparisons when using a set instead of $O(n)$ when using a queue (n is the number of elements in the container). The only downside is that we have to simultaneously update both the set and the queue. However, the obtained speedup is worth the price.

Algorithm 3 is implemented in `computeAnalysisForFnc()`. To store a copy of `OUT[block]`, we make a shallow copy of the map storing the mapping of memory places to their ranges. The scheduling of blocks, described in Algorithm 4, is done in `scheduleBlock()`. The analysis of a single basic block from Algorithm 5 is performed in `computeAnalysisForBlock()`. This function first calls `computeInputRanges` to update the input ranges of the current basic block (Algorithm 6), and then it goes over all the instructions in the basic block, updating `OUT[block]` (Algorithm 7).

6.6 `GlobAnalysis`: Analysis of Global Variables

The `GlobAnalysis` class, whose design is given in Section 5.6, is implemented in files `GlobAnalysis.{cc,h}`. This class provides static functions to perform our analysis of global variables. The entry point is `computeGlobAnalysis()`, which takes an instance of `CodeStorage::Storage`. It visits all the function definitions in the program by calling `computeGlobAnalysisForFnc()` over each of them. This function implements a simple work list algorithm (see Section 2.4), which traverses all the instructions in the function. Whenever a statement that assigns something into a global variable is encountered, the UID of the global variable is stored into the private static `std::map<int, bool> globVarInit` map. Having a global variable `x`, `globVarInit[x] == true` indicates that `x` may be modified during the program execution. From the outside, this map can be accessed by calling `isModified(id)`, where `id` is the UID of the global variable we want to check.

Apart from the functions mentioned above, there are several auxiliary functions, such as `isGlobal(id)`, which checks whether the given variable is global, or `printGlobAnalysis()`, which can be used to print the result of the analysis during development or debugging.

6.7 `LoopFinder`: Analysis of Loops

The `LoopFinder` class, designed in Section 5.7, is implemented in files `LoopFinder.{cc,h}`. Due to space requirements, we omit the implementation details as this class straightforwardly implements the analysis designed in Section 5.7.

6.8 `Utilities`: Various Auxiliary Functions

The `Utilities` class, implemented in files `Utilities.{cc,h}`, provides two auxiliary functions. The first one, `getMaxRange()`, takes a `cl_operand`, and based on its type, it returns the maximal range for that type. It is implemented as a switch over the `cl_operand`'s code, and internally uses `Range::getMaxRange()`. If the operand is a variable that is nested in a structure or an array, we use the second parameter of `Utilities::getMaxRange()`, named `indexes`, in an analogous way to `OperandToMemoryPlace::convert()`.

The second function provided by this class is `convertOperandToNumber()`. It is used to convert the given `cl_operand` into an instance of `Number`. Once again, its body is just a switch over the `cl_operand`'s code. This code has to be either `CL_OPERAND_CST` or `CL_OPERAND_VAR`. Otherwise, the operand cannot be converted into a number.

6.9 Interface and Output Format of the Analyser

After building the Code Listener infrastructure and the analyser (see the `README` file on the enclosed CD), it can be run by issuing the following command:

```
./gcc-install/bin/gcc -fplugin=vra_build/libvra.so input.c
```

where `-fplugin=vra_build/libvra.so` tells `gcc` to use our value-range analysis plugin, and `input.c` is the input C source file to be analysed.

The output format is as follows. For every function in the input source file, its name is emitted, and then, the list of its basic blocks, including the ranges of variables at the input to the block as well at the output of the block are emitted. For example, for the following trivial program

```
1 int main() {
2     int i = 0;
3     return i;
4 }
```

the analyser emits

```
1 ---------- Function main() ----------
2 Block L1[IN] at lines from 2 to 3:
3 Block L1[OUT]:
4         i = { <0, 0> }
5 Block L2[IN] at lines from 3 to 4:
6         i = { <0, 0> }
7 Block L2[OUT]:
8         i = {<(0, 0> }
```

To see the intermediate representation of the parsed input source code, run the above command also with the following parameter:

```
-fplugin-arg-libvra-dump-pp
```

This will cause the analyser to emit the intermediate representation, including the labels of basic blocks referenced in the analyser's output:

```
1 main():
2                   goto L1
3
4         L1:
5                   %mF1716:i := 0
6                   %r1718 := %mF1716:i
7                   goto L2
8
9         L2:
10                  ret %r1718
```

It should be noted that this emission is already provided by Code Listener, it is not part of the implementation of the developed analyser. A complete example of running the analyser and analysing its output is provided in Appendix A.

6.10 Metrics

Finally, we present several metrics of the implemented analyser. More precisely, they are given in Table 6.1. This table shows the total number of lines in source files implementing the used classes, and the total number of functions which the classes implement (`public`, `private`, `static`, member, and `friend` functions are all included). Testing code is not included in these metrics. For metrics of the testing code, see Chapter 7.

Table 6.1: Metrics regarding the implemented classes

Class	Number of code lines	Number of functions
`GlobAnalysis`	567	16
`LoopFinder`	1295	26
`MemoryPlace`	68	4
`Number`	1592	66
`OperandToMemoryPlace`	248	3
`Range`	2761	87
`Utility`	188	2
`ValueAnalysis`	1352	23

All in all, the implementation (without tests) has 8210 lines, including comments and empty lines.

Chapter 7

Testing and Evaluation

This chapter discusses the ways of testing the implementation, and evaluates the obtained results. First, Section 7.1 gives a brief introduction to test-driven development, which was the method used to develop all the low-level classes that underly the implementation of the analyser. So-called unit tests were the primary means of testing the implementation of the underlying classes. Their description is also included in Section 7.1. After that, Section 7.2 discusses overall tests, which were used to check the implementation of the complete analyser. These tests inspect the combined functionality of all parts of the analyser and focus on verifying whether the output is correct. The present chapter is concluded by Section 7.3, which evaluates the results obtained by testing.

7.1 Unit Tests

The underlying classes `Number`, `Range`, `Utility`, `MemoryPlace`, and `OperandToMemoryPlace` were developed by means of *test-driven development*, popularized by Kent Beck [6]. The basic idea behind test-driven development is that before any production code is written, a test for it is added. Only after that the programmer writes the actual code that passes the test. In a greater detail, the development is structured into very short cycles, composed of the following three steps:

(1) Write an automated test case that fails.

(2) Make the test pass by writing the actual implementation.

(3) Clean up the implementation.

In Step 1, a new test case is added. It is important for the test to be automated so we can quickly check whether it succeeds or fails, and it has to initially fail. If it does not fail, then either there are no improvements needed (i.e. the implementation suffices) or there is an error in the test itself. In this way, by first checking that the test fails, we verify that the test is correct.

After that, in Step 2, we write the actual implementation. By having an automated test case, we know when we are done—when the test passes. During this step, we focus only on

writing code that makes the test pass. Indeed, if we think of an idea for a new feature, we write it down on a piece of paper and leave it for the next cycle.

Finally, in Step 3, the implementation is cleaned-up by means of *refactoring*, which is a process of restructuring existing code without changing its external behaviour [22]. Since the test passes, during this step, we may solely focus on improving the code without worrying of breaking something. Indeed, the test tells us if we go astray by failing. The reason for dividing the implementation and refactoring into two steps is that it leads to much cleaner code [22]. In this way, we can first focus on writing a correct implementation, and then, we focus only on making the code as clean as possible. Therefore, we have to juggle only a single ball at a time. For more information on test-driven development, see [6].

An appropriate method of testing during test-driven development is *unit testing*, where we test a single component at a time in isolation [41]. This way of testing enables us to write tests for any public method of the developed class and ensure that the code meets its design and behaves as intended. To ease the creation of unit tests, we have used an open-source unit testing framework called Google Test [52]. We have chosen this framework because it is free, easy-to-use, written in C++, runs on a variety of platforms (GNU/Linux, Mac OS X, MS Windows) and has a rich set of features. Among them are automatic test discovery, many types of assertions, death tests, and various options for running the tests.

As an example of a test written in the Google Test framework, consider the following piece of code:

```
1 TEST_F(RangeTest,
2 DivisionByZeroResultsIntoOverApproximation)
3 {
4   // x / 0 -> undefined behaviour in C (we have to over-approximate)
5   EXPECT_EQ(Range(Interval(I<int>(vmin<int>()), I<int>(vmax<int>()))),
6       exact_div(Range(Interval(I<int>(1), I<int>(5))),
7                 Range(Interval(I<int>(0), I<int>(0)))));
8 }
```

Here, we have declared a test case `DivisionByZeroResultsIntoOverApproximation` within a fixture `RangeTest`. In terms of the Google Test framework, a *test fixture* is a class that groups test cases for a single component. In this test case, we check that when we are dividing two ranges, where the second contains zero, the resulting range is the maximal range, i.e. the result is an over-approximation because dividing an integral number by zero is undefined in C (see §5 of Section 6.5.5 in [24]). For testing purposes, we have defined the templates `I<>`, `vmin<>` and `vmax<>`, which construct an instance of `Number`, return the minimal number of the given type and return the maximal number of the given type, respectively. The `EXPECT_EQ` macro is provided by the Google Test framework to check the equality of two values. Its first parameter is the reference result, and the second one is the actual result.

When we run the test suite for `RangeTest` with a working implementation, we obtain the following output[1], which says that our test has passed:

```
1 $ ./RangeTest
2  ...
3 [ RUN      ] RangeTest.DivisionByZeroResultsIntoOverApproximation
4 [       OK ] RangeTest.DivisionByZeroResultsIntoOverApproximation
5  ...
```

[1]Only the relevant part is shown while the rest of the output is ommitted for clarity.

However, if we have a flawed implementation, we get an error:

```
$ ./RangeTest
 ...
[ RUN      ] RangeTest.DivisionByZeroResultsIntoOverApproximation
RangeTest.cc:4512: Failure
Value of: exact_div(Range(Interval(I<int>(1), I<int>(5))),
                    Range(Interval(I<int>(0), I<int>(0))))
  Actual: { (0, 2147483647) }
Expected: Range(Interval(I<int>(vmin<int>()), I<int>(vmax<int>())))
Which is: { (-2147483648, 2147483647) }
[  FAILED  ] RangeTest.DivisionByZeroResultsIntoOverApproximation
 ...
```

The output clearly says what is the expected output and what we have actually got from `exact_div()`. The main advantage of using a unit testing framework like this is that it is very simple to add and run tests, and quickly evaluate which tests have passed and which have not. Otherwise, we would either have to come up with a hand-crafted solution, thus reinventing the wheel, or to check the outputs manually, which is tedious and error-prone.

The overall number of unit tests created for the underlying classes is shown in Table 7.1. They are placed in `vra/tests-unit`.

Table 7.1: Overall number of unit tests for underlying classes

Class	Number of unit tests
`Number`	201
`Range`	309
`MemoryPlace`	12
`OperandToMemoryPlace`	17
`Utility`	19

All in all, the unit-testing code has 11136 lines, including comments and empty lines. To run all the unit tests, one can use the `run-all-unit-tests.sh` script.

7.2 Overall Tests

Apart from unit tests, discussed in the previous section, another type of tests was used to test the analyser: overall tests. They are focused on testing the analyser as a whole by providing an input C source code and a reference output from the analyser. We have then created a shell script, `tests-run.sh`, which runs the analyser over all tests, gathers their outputs, and compares them with the reference outputs. This way of testing also gives us a feedback when changing something in the analyser because when a test fails, a diff is provided.

When all tests pass, the following output is emitted:

```
$ ./tests-run.sh
Running the analyser on tests in 'tests'...

test-0001.c          [OK]
test-0002.c          [OK]
```

```
 6 test-0003.c          [OK]
 7 ...
 8
 9   Passed: 82/82
10   Failed: 0/82
```

However, when a test fails, a `FAIL` message is emitted, including a diff:

```
 1 $ ./tests-run.sh
 2 Running the analyser on tests in 'tests'...
 3
 4 test-0001.c          [OK]
 5 test-0002.c          [FAIL]
 6   21c21
 7   <     x.a = { <2, 2> }
 8   ---
 9   >     x.a = { <1, 1> }
10 test-0003.c          [OK]
11 ...
12
13   Passed: 81/82
14   Failed: 1/82
```

We can then check whether the new output is correct or not. For example, when improving the analysis, the new result may be more accurate.

Another script that we have created, called `tests-gen-ref-outputs.sh`, can be used to (re)generate the reference test outputs. It may be used, for example, if we just change the format of the output from the analyser and we want to regenerate all the reference tests to match the new format.

As we have already said, the overall tests are focused on testing the analyser as a whole rather than testing the individual components (for this purpose, we use unit tests). They include tests that check the results of value analysis over programs having multiple functions, assignments, control-flow statements like `if`, `if-else` and loops.

We have created over 80 overall tests. They are placed in `vra/tests-overall`.

7.3 Evaluation

The developed analyser was tested on two systems: a 64b Gentoo Linux distribution running kernel `3.6.11` and `gcc` version `4.7.2`, and a 64b Arch Linux distribution with kernel `3.8.6` and `gcc 4.8.0`. The used version of Code Listener was `2013-04-28-fa6664314a`, obtained from its Git repository [16]. On both of these systems, all the tests successfully passed. In all the performed tests, the analyser returned correct results with respect to safety, i.e. all the returned ranges were either precise or an over-approximation of the precise results.

Finally, the created unit tests will be of a great help when extending the analyser in the future, for the following two reasons. The first one is that they provide a safety net when modifying the classes. If a modification that breaks the functionality is introduced, their failure can alert us that what we did might not have been correct. The second one is that they show a sample way of using the API of the classes underlying the analyser. Indeed, every unit test shows how a particular class is used under certain circumstances. This may help to grasp the API more easily than from a standalone Doxygen documentation.

Chapter 8

Conclusion

In this work, we have presented the design and implementation of a value-range analyser over C programs. The analyser is built on top of the Code Listener architecture, which simplified its implementation because we did not have to deal with parsing of the input C program. The analyser was tested by an extensive suite of unit tests, and also by tests that check the analyser as a whole. The evaluation of the performed tests suggest that the analyser is safe, which was the primary goal. Its code base is formed by nearly 20 000 lines of code (including unit tests). In a direct connection with the topic of the present work, we have published a paper in the local student conference EEICT 2013 [54].

In its general form, value-range analysis represents a topic for a dissertation thesis rather than for a master's thesis. This is caused by a narrow focus of many methods that have been designed to deal with value-range analysis of C programs. Indeed, most of them require certain condition to be satisfied in order for the method to work properly. The reason is that the C language standard leaves many scenarios implementation-defined or even undefined. Moreover, its weak type system allows one to bypass type checks and create hard-to-properly-analyse code.

To sum up our experience with developing a tool for value-range analysis, we were able to design and implement an analyser that can handle on certain type of programs (see the limitations in Section 5.8) during the course of two semesters. The hardest work was to properly implement the underlying classes representing numbers, ranges, and operations over them. The reason is that the C standard requires many types of conversions to be done, and, as we have already mentioned, leaves many situations implementation-defined or undefined. This had to be properly reflected in the underlying classes. As for the Code Listener architecture, its documentation is sometimes unclear, but its author responded promptly to our emails, which eased the development.

The analyser can be extended in several ways. First, switching from intraprocedural analysis into interprocedural analysis would make the results more precise. Second, an implementation of some sort of alias analysis would make the analyser safe also in cases when pointers are used (at least when they are used in a type-safe way). Third, the designed loops analysis (see Section 5.7) can be improved to handle more types of loops. Finally, a graphical user interface may be added to the analyser to make the output fancier.

The developed analyser is freely available on the following web site:

`http://www.stud.fit.vutbr.cz/~xduric00/vra/`

It is distributed under the GNU GPL v3 license. Apart from sources of the analyser, the web site also provides a documentation of the application interface.

Appendix A

Example of Analysis

In this appendix, we present a complete example of using the developed analyser to find a buffer-overflow vulnerability in a simple C program. Consider the source code in Figure 3.1 from Section 3.1, which is for convenience repeated below.

```
1 #include <stdio.h>
2
3 int main(int argc, const char *argv[])
4 {
5     int importantData = 1;
6     int buffer[10];
7
8     int i;
9     for (i = 0; i <= 10; i++) {
10        buffer[i] = 9999;
11    }
12
13    printf("importantData = %d\n", importantData);
14
15    return 0;
16 }
```

Let us assume that this code is stored in a file named `example.c`. After compiling and building our analyser (see the `README` file on the enclosed CD), we may run it over the code by issuing the following command:

```
./gcc-install/bin/gcc -fplugin=vra_build/libvra.so -fplugin-arg-libvra-
    dump-pp example.c
```

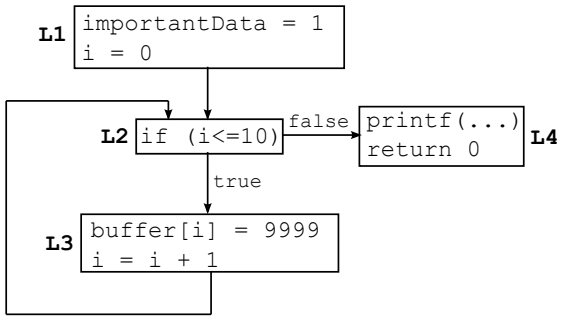
We force `gcc` to use our value-analysis plugin to inspect the source code. What we obtain is an output divided into two parts. The first part is a dump of the Code Listener's intermediate representation of our program:

```
1 main(%arg1: %mF2160:argc, %arg2: %mF2161:argv):
2                 goto L1
3
4         L1:
5                 %mF2164:importantData := 1
6                 %mF2166:i := 0
7                 goto L2
8
9         L3:
```

```
10                  %mF2165:buffer[%mF2166:i] := 9999
11                  %mF2166:i := (%mF2166:i + 1)
12                  goto L2
13
14          L2:
15                  %r1048577 := (%mF2166:i <= 10)
16                  if (%r1048577)
17                          goto L3
18                  else
19                          goto L4
20
21          L4:
22                  printf("importantData = %d\n", %mF2164:importantData)
23                  %r2171 := 0
24                  goto L5
25
26          L5:
27                  ret %r2171
```

In this dump, we see that there is a single function, `main()`. It is divided into five basic blocks, labeled `L1` through `L5`. As you can see, there is no more any evidence of the `for` loop that was present in the original source code. Morever, all operations were converted into a uniform form. For example, the expression `i <= 10` is first evaluated and the result is stored into an artificial variable `%r1048577`. This variable is then used in the subsequent condition. Nevertheless, even though the `for` loop is no longer present in the code in its original, structured form, by inspecting the code, we immediately see that it does the same computation as the original program.

For clarity, a graphical representation of the above code is given below in the form of a control-flow graph:



Notice that as `buffer` is unititialized in `L1`, it does not appear in the dump from the analyser nor in the control-flow graph.

The second—and the most important—information provided by the analyser are the ranges the variables may have in the basic blocks `L1` through `L5`:

```
1 ---------- Function main() ----------
2 Block L1[IN] at lines from 5 to 9:
3 Block L1[OUT]:
4         i = { <0, 0> }
5         importantData = { <1, 1> }
6 Block L2[IN] at lines from 9 to 9:
7         buffer[] = { <9999, 9999> }
8         i = { <0, 11> }
```

```
 9         importantData = { <1, 1> }
10 Block L2[OUT]:
11         buffer[] = { <9999, 9999> }
12         i = { <0, 11> }
13         importantData = { <1, 1> }
14 Block L3[IN] at lines from 10 to 10:
15         buffer[] = { <9999, 9999> }
16         i = { <0, 10> }
17         importantData = { <1, 1> }
18 Block L3[OUT]:
19         buffer[] = { <9999, 9999> }
20         i = { <1, 11> }
21         importantData = { <1, 1> }
22 Block L4[IN] at lines from 13 to 15:
23         buffer[] = { <9999, 9999> }
24         i = { <0, 11> }
25         importantData = { <1, 1> }
26 Block L4[OUT]:
27         buffer[] = { <9999, 9999> }
28         i = { <0, 11> }
29         importantData = { <1, 1> }
30 Block L5[IN] at lines from 16 to 16:
31         buffer[] = { <9999, 9999> }
32         i = { <0, 11> }
33         importantData = { <1, 1> }
34 Block L5[OUT]:
35         buffer[] = { <9999, 9999> }
36         i = { <0, 11> }
37         importantData = { <1, 1> }
```

The output contains the ranges of variables for every input to a basic block and every output of the block. For example, the ranges in `L3[IN]` say that at the beginning of the original `for` loop's body (line `10`), `i` may have values from the interval $\langle 0, 10 \rangle$. Since we know that `buffer` has 10 elements, and we are accessing it by `i` in the loop, this may imply that there is an invalid access. This needs not to be the case as the analyser may over-approximate the values, but after a quick inspection, we find out that we have really run into an "off-by-one" error here.

Even though the output is not as precise as the hypothetical analyser in Figure 3.2 from Section 3.1, it may still be valuable when searching for buffer overflow vulnerabilities or other related issues.

Appendix B

Contents of the Enclosed CD

The enclosed CD has the following contents:

- `README`: a file summarizing the contents of the CD,
- `thesis.pdf`: an electronic version of this text,
- `thesis/`: a source code of this text,
- `predator/`: a source code of the Code Listener infrastructure (from the time of writing this thesis),
- `vra/`: a source code of the value-range analyser,
- `api-documentation/`: a generated documentation of the analyser's application interface (API).

Bibliography

[1] AbsInt team. StackAnalyzer: Stack Usage Analysis. `http://www.absint.com/stackanalyzer/index.htm`. [cit. 2013-01-25].

[2] AbsInt team. ValueAnalyzer: Static Value Analysis. `http://www.absint.com/valueanalyzer/index.htm`. [cit. 2013-01-25].

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd ed.)*. Addison-Wesley, 2006. ISBN 0201100886.

[4] P. Baudin, L. Correnson, and Z. Dargaye. WP Plug-in (Draft) Manual. `http://frama-c.com/download/wp-manual-Oxygen-20120901.pdf`. [cit. 2013-01-21].

[5] P. Baudin, P. Cuoq, J. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI/ISO-C Sprecification Language. `http://frama-c.com/download/acsl-implementation-Oxygen-20120901.pdf`. [cit. 2013-01-25].

[6] K. Beck. *Test Driven Development: By Example*. Addison-Wesley, 2002. ISBN 978-0321146533.

[7] L. Boettger. The Morris Worm: How it Affected Computer Security and Lessons Learned by it. `http://www.giac.org/paper/gsec/405/morris-worm-affected-computer-security-lessons-learned/100954`. [cit. 2013-02-26].

[8] R. Bonichon and B. Yakobowski. Frama-C's metrics plug-in. `http://frama-c.com/download/metrics-manual-Oxygen-20120901.pdf`. [cit. 2013-01-21].

[9] J. Boulanger, editor. *Static Analysis of Software: The Abstract Interpretation*. Wiley, 2012. ISBN 978-1-84821-320-3.

[10] V. Campos, R. Rodrigues, I. Costa, and F. Pereira. Tool for Static Analysis of Whole Programs. `http://range-analysis.googlecode.com/svn-history/r203/trunk/doc/Manuscript/DTool/paper.pdf`. [cit. 2013-02-24].

[11] J. Cong, Y. Fan, G. Han, Y. Lin, J. Xu, Z. Zhang, and X. Cheng. Bitwidth-aware scheduling and binding in high-level synthesis. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, pages 856–861, New York, NY, USA, 2005. ACM.

[12] L. Correnson, P. Cuoq, F. Kirchner, V. Prevosto, A. Puccetti, J. Signoles, and B. Yakobowski. Frama-C User Manual. `http://frama-c.com/download/user-manual-Oxygen-20120901.pdf`. [cit. 2013-01-21].

[13] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[14] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C': A software analysis perspective. `http://sefm2012.city.academic.gr/other/presentations/SEFM/session4/Yakobowski.pdf`. [cit. 2013-01-24].

[15] P. Cuoq, V. Prevosto, and B. Yakobowski. Frama-C's value analysis plug-in. `http://frama-c.com/download/value-analysis-Oxygen-20120901.pdf`. [cit. 2013-01-21].

[16] K. Dudka. Predator. `https://github.com/kdudka/predator`. [cit. 2013-04-20].

[17] K. Dudka, P. Peringer, and T. Vojnar. Code listener. `http://www.fit.vutbr.cz/research/groups/verifit/tools/code-listener/`. [cit. 2013-01-07].

[18] K. Dudka, P. Peringer, and T. Vojnar. An easy to use infrastructure for building static analysis tools. `http://www.fit.vutbr.cz/research/groups/verifit/tools/code-listener/cl-ppt.pdf`. [cit. 2013-01-07].

[19] K. Dudka, P. Peringer, and T. Vojnar. FAV 2011: GCC plug-ins, predator. `https://www.fit.vutbr.cz/study/courses/FAV/public/Lectures/additional-lecture-predator.pdf`. [cit. 2013-01-07].

[20] K. Dudka, P. Peringer, and T. Vojnar. Predator. `http://www.fit.vutbr.cz/research/groups/verifit/tools/predator/`. [cit. 2013-01-13].

[21] K. Dudka, P. Peringer, and T. Vojnar. An easy to use infrastructure for building static analysis tools. In *13th International Conference on Computer Aided Systems Theory—EUROCAST'11*, pages 328–329, Las Palmas, Spain, 2011. Springer-Verlag. ISBN 978-84-693-9560-8.

[22] M. Fowler et al. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999. ISBN 978-0201485677.

[23] International Organization for Standardization. [ISO/IEC 14882:1998] ISO/IEC. Programming Languages–C++. Geneva, Switzerland, 1998.

[24] International Organization for Standardization. [ISO/IEC 9899:1999] ISO/IEC. Programming Languages–C. Geneva, Switzerland, 1999.

[25] Frama team. The Frama-C platform. `http://frama-c.com`. [cit. 2013-01-23].

[26] Frama team. Frama-C's Slicing plug-in. `http://frama-c.com/slicing.html`. [cit. 2013-01-26].

[27] Free Software Foundation. The GNU multiple precision arithmetic library. `http://gmplib.org/`. [cit. 2013-04-25].

[28] GCC Contributors. Gimple - GNU GCC internals. `http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html`. [cit. 2013-01-13].

[29] W. J. Gilbert. *Modern Algebra with Applications*. Wiley-Interscience, 2002. ISBN 978-0471235439.

[30] P. Habermehl, L. Holík, A. Rogalewicz, J. Šimácek, and T. Vojnar. Forester. `http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/`. [cit. 2013-01-13].

[31] W. H. Harrison. Compiler analysis of the value ranges for variables. *Software Engineering, IEEE Transactions on*, SE-3(3):243 – 250, 1977.

[32] P. Herrmann. Frama-C's annotation generator plug-in. `http://frama-c.com/download/rte-manual-Oxygen-20120901.pdf`. [cit. 2013-01-21].

[33] U. P. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009. ISBN 978-0849328800.

[34] B. Křena and T. Vojnar. Automated formal analysis and verification: an overview. *International Journal of General Systems*, 2013(42):335–365, 2013.

[35] G. Lann. An Analysis of the Ariane 5 Flight 501 Failure—A System Engineering Perspective. `http://www.niwotridge.com/Resources/Ariane5Resources/78890339.pdf`. [cit. 2013-02-26].

[36] O. Lengál and T. Vojnar. FAV 2012: Abstrat interpretation. `https://www.fit.vutbr.cz/study/courses/FAV/public/Lectures/fav-lecture-09.pdf`. [cit. 2013-02-21].

[37] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 136–146, Washington, DC, USA, 2009. IEEE Computer Society.

[38] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 20:1355–1371, 2006.

[39] F. Nielson, H. R. Nielson, and Ch. Hankin. *Principles of Program Analysis*. Springer, 2005. ISBN 3-540-65410-0.

[40] NIST. National vulnerability database. `http://nvd.nist.gov/home.cfm`. [cit. 2013-02-26].

[41] R. Osherove. *The Art of Unit Testing*. Manning Publications, 2009. ISBN 978-1933988276.

[42] R. C. J. Patterson. Accurate static branch prediction by value range propagation. *SIGPLAN Not.*, 30(6):67–78, 1995.

[43] PolySpace team. Static Analysis with Polyspace Products. `http://www.mathworks.com/products/polyspace/`. [cit. 2013-01-25].

[44] T. Salminen and E. Korhonen. GCC Gimple. `https://wiki.aalto.fi/display/t1065450/GCC+Gimple`. [cit. 2013-01-07].

[45] A. Simon. *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*. Springer, 2008. ISBN 978-1848000162.

[46] IEEE Computer Society. IEEE standard for floating-point arithmetic. IEEE. doi:10.1109/IEEESTD.2008.4610935. ISBN 978-0-7381-5753-5. IEEE Std 754-2008.

[47] R. Sol, C. Guillon, F. M. Q. Pereira, and M. A. S. Bigonha. Dynamic elimination of overflow tests in a trace compiler. In *Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software*, CC'11/ETAPS'11, pages 2–21, Berlin, Heidelberg, 2011. Springer-Verlag.

[48] M. Stephenson, J. Babb, and S. Amarasinghe. Bidwidth analysis with application to silicon compilation. *SIGPLAN Not.*, 35(5):108–120, 2000.

[49] N. Stouls and V. Prevosto. Aoraï Plugin Tutorial. `http://frama-c.com/download/aorai-manual-Oxygen-20120901.pdf`. [cit. 2013-01-21].

[50] S. Tallam and R. Gupta. Bitwidth aware global register allocation. *SIGPLAN Not.*, 38(1):85–96, 2003.

[51] GCC Team. GCC, the GNU Compiler Collection. `http://gcc.gnu.org/`. [cit. 2013-02-26].

[52] Google Test Team. Google test – Google C++ Testing Framework. `http://code.google.com/p/googletest/`. [cit. 2013-03-20].

[53] Sparse Team. Sparse. `https://sparse.wiki.kernel.org/`. [cit. 2013-01-17].

[54] D. Ďuričeková. Static value-range analysis over C programs. In *Proceedings of the 19th Conference STUDENT EEICT 2013 Volume 2*, pages 304–306, Brno, CZ, 2013.

[55] A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded c programs. *SIGPLAN Not.*, 39:231–242, 2004.

[56] D. Wagner, S. F. Jeffrey, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *In Network and Distributed System Security Symposium*, pages 3–17, 2000.

[57] B. Yakobowski and R. Bonichon. Frama-C's Mthread plug-in. `http://frama-c.com/download/mthread-manual-Oxygen-20120901.pdf`. [cit. 2013-01-21].