ABSTRACTION VIA PROGRAM TRANSFORMATION

HENRICH LAUKO

PHD THESIS

July 2023

SUPERVISOR
prof. RNDr. Jiří Barnat, Ph.D.

CONSULTANT
RNDr. Petr Ročkai, Ph.D.

Faculty of Informatics
Masaryk University

Declaration

Thereby I declare that this thesis is my original work, which I have created on my own. All sources and literature used in writing the thesis, as well as any quoted material, are properly cited, including full reference to its source.

Henrich Lauko

Supervisor
prof. RNDr. Jiří Barnat, Ph.D.

Abstract

In the field of computer-aided verification, abstraction techniques are indispensable as they play a critical role in reducing the complexity of the verification process to manageable sizes. However, these techniques are usually tightly integrated into tools, resulting in undesired complexity and neglect of reusable design. If one wants to employ a particular abstraction in several tools, the common approach is to implement the abstraction separately for each tool. However, this results in unnecessary duplication of an effort and inconsistent results as different implementations may vary. This thesis aims to address this challenge by developing a solution to reduce the duplication and complexity of abstraction. The overarching objective is to create a tool-independent program abstraction and utilize it to design program analysis techniques that can be applied by any tool, thereby reducing its complexity and allowing abstraction reusability.

What is common between tools, is the program representation they operate with. One such representation that has gained popularity is the LLVM intermediate representation of the Clang compiler. It can serve as a shared source of truth between tools and encode additional information for program analysis. In this doctoral thesis, we propose a solution to tool-independent abstraction by reconceiving it in terms of the intermediate representation. The key idea of the presented approach is to express abstract semantics in terms of intermediate representation, resulting in the abstraction being understandable to any tool which uses LLVM IR. We refer to this approach as compilation-based abstraction, as it essentially compiles abstract semantics into the program being analyzed.

The compilation-based abstraction approach has broad applicability in the field of program analysis. This thesis demonstrates its potential in recasting symbolic execution as program abstraction, enabling explicit tools to perform symbolic analysis or even run symbolic programs natively. The focus will also be on more complex program primitives such as C arrays or strings, which require elaborated abstraction. The approach is extended to encompass aggregate types and library-level abstractions. Additionally, attention is given to dynamic memory abstraction, specifically the problem of ambiguous dynamic memory layout, which is often overlooked in analysis tools. Furthermore, the compilation-based abstraction approach opens up new possibilities for refinement techniques that can enhance the precision of the employed abstraction. This thesis explores the potential for instrumenting refinement directly into the program and counterexample-guided syntactic abstraction of program representation.

In summary, this thesis provides an autonomous solution for instrumenting abstraction into program representation. The presented approach addresses various issues related to program domain interactions, control flow, and dynamic memory. It includes adaptations of symbolic execution, software model checking, and syntactic refinement loops. Moreover, by enabling the native execution of compiled abstraction, this approach not only streamlines the program abstraction design process but also enhances the efficiency of program analysis. Our main contributions culminated in the development of an LLVM Abstraction & Refinement Tool – LART. This tool implements all discussed abstractions and has been shown to be effective both in conjunction with other tools and in producing natively executable abstractions.

Keywords

program analysis, software verification, model checking, symbolic model checking, abstract execution, DIVINE, LLVM, program transformation, compilation, abstraction, refinement, heap analysis, string analysis, LART, C, C++, implementation

Acknowledgements

I would like to express my deepest gratitude to all of you who have supported me in completing this thesis. Your contributions, guidance, and encouragement have been invaluable, and I am truly grateful for your involvement. My first thoughts go to Mornfall and Jiří Barnat for jointly providing excellent guidance throughout my study. I am truly fortunate to have had the opportunity to work closely with them. I would also like to extend my heartfelt appreciation to my wife, Dominika, for her unwavering support, patience, and understanding. Her presence and encouragement have been a constant source of strength throughout this journey. My thanks also go to the members and former members of the ParaDiSe and Formela laboratories for their camaraderie and contributions. Your stimulating discussions and shared experiences have greatly influenced my PhD life. Last but not least, I would like to thank my family, friends, and all the good people who helped me keep my sanity and focus and made this phase of my life truly enjoyable.

Contents

1 Introduction

The field of computer science is rapidly evolving, with an increasing number of techniques, tools, and algorithms being developed. However, despite the many advancements in the field, many of new tools and algorithms are not designed with reusability in mind. This trend has caused researchers to reimplement standard techniques whenever they want to extend or adapt existing work, wasting their time and resources. Moreover, these re-implementations often produce results that are hardly comparable with previous approaches, as the performance is becoming implementation-dependent.

The field of computer-aided verification is no exception to this trend. The goal of verification is to determine whether a system under test satisfies a given specification. While this task is generally undecidable, many verifiers are capable of reasoning about complex systems through the use of abstraction techniques. In general, abstraction reduces the amount of information that a verifier needs to consider, thereby reducing the program's state space but at the cost of approximated results. This can be achieved, for example, by maintaining only specific properties of program values. Given that abstraction-based techniques are indispensable for the successful verification of larger systems, the verification community has been inundated with various implementations of abstraction techniques.

Abstraction is applied at many levels of the program verification process, including the model of computation on which the verifier operates. In general, it would be impractical, if not impossible, to verify the entire software stack, from the bottom of the hardware layer, through an operating system environment up to the top-level program under test. Hence, the verifier abstracts from the environment and reasons only about the program and a subset of its environment. The program representation can also be abstracted by first translating the program into a simpler verification-friendly format, such as byte-code intermediate representation or a program graph.

The focus of our interest is an abstraction of the program environment, which is also referred to as nondeterministic inputs to the program. To ensure accurate verification, it is necessary to examine the unpredictable nature of the program environment's interaction. A verification tool must take into account all potential program behaviors that are dependent on program inputs and other sources of nondeterminism. To handle a vast number of possibilities, verifiers employ abstraction to categorize similar behaviors and, thus, reason about a reduced set of options. Alternatively, verification tools use abstraction to retain only the behaviors that are relevant to the verified property.

Sources of Nondeterminism

Analyzing programs that interact with their environment is a complex task. We may observe numerous different outcomes that depend on the

interactions with a user, system, and the overall environment. These interactions are referred to as nondeterministic choices and can quickly become a significant problem for analysis as they influence a combinatorial growth of a program state space.

Although a certain level of combinatorial explosion is inevitable, in common programs, a significant amount of choices does not influence the program's correctness. For instance, in a reliable network protocol, the received message must be identical to the one transmitted. However, the contents of the message are of little importance. A developer usually employs unit tests to validate the protocol's behavior. In tests, the message payloads are mostly placeholder data. Despite the importance of such abstraction, it is not practical to expect developers to manually perform this process for all data types. In fact, in program analysis, we want to automate the abstraction process. Yet, it is crucial to acknowledge that while developers have a unique insight into the program's structure that allows them to identify abstraction opportunities, automated tools lack such an advantage.

In the context of program analysis, low-level representations such as LLVM bitcode pose a challenge due to a limited understanding of the internal workings of a program. The input data often lack structure, leading to byte-by-byte examination and decision-making without knowledge of emergent behavior. Enumerating all possible outcomes is computationally expensive and often unnecessary as the majority of inputs will quickly lead the program to identify the input as invalid and terminate. To reduce the number of necessary program paths in analysis, it is ideal to capture the structure of the input in abstraction. While analyzing raw programs and entire code bases is desirable, it is already valuable to develop abstractions at the unit test level. Although traditional testing uses fixed environment inputs, structured information can be used to mock up the environment in automated tools. Nevertheless, it is still easy to write unit tests that cause exhaustive tools to run out of resources or to cause deep bugs that evade bounded analysis. One common deep types of errors are a failure at a boundary condition, such as overflowing an integral type, or off-by-one errors near hard-coded limits, such as static buffer sizes or message size limits. Humans analyzing such errors may artificially simplify the problem when debugging by lowering limits, but a tool designed to detect such issues automatically cannot rely on human insight. [Roč15]

[Roč15]: Ročkai (2015), "Model Checking Software"

Program Abstraction

The goal of abstraction is to provide a generic solution that can reliably eliminate unnecessary noise and choices that are irrelevant to the program's correctness. This allows the program's state space to be more manageable and reduces the risk of incomplete program analysis. The abstraction-based tools automate this process. However, as mentioned previously, these tools scarcely provide efficiently extendable algorithms or abstraction techniques. Moreover, due to a lack of standardization, they are hardly reusable or composable between each other.

According to the classification of techniques used in the recent verification competition SV-COMP [Bey19], many tools utilize a similar set of

[Bey19]: Beyer (2019), "Automatic Verification of C and Java Programs: SV-COMP 2019"

abstraction techniques (see Figure 1.1), namely counterexample-guided abstraction refinement [Cla+00], predicate abstraction [FQ02], shape analysis [Yan+08], lazy abstraction [Hen+02], or symbolic data representation [Bur+90; Kin76; MR18]. Even though, these techniques share similar ideas, each of the tools ships with its own implementation. Moreover, the design of these algorithms is not well suited for reusability.

PARTICIPANT	CEGAR	Predicate Abstraction	Symbolic Execution	Bounded Model Checking	k-Induction	Property-Directed Reach.	Explicit-Value Analysis	Numeric. Interval Analysis	Shape Analysis	Separation Logic	Bit-Precise Analysis	ARG-Based Analysis	Lazy Abstraction	Interpolation	Automata-Based Analysis	Concurrency Support	Ranking Functions	Evolutionary Algorithms
2LS				✔	✔			✔			✔						✔	
AProVE			✔				✔	✔		✔	✔						✔	
CBMC				✔							✔					✔		
CBMC-Path				✔							✔					✔		
CPA-BAM-BnB	✔	✔					✔				✔	✔	✔	✔				
CPA-Lockator	✔	✔					✔				✔	✔	✔	✔		✔		
CPA-Seq	✔	✔		✔	✔		✔	✔	✔		✔	✔	✔	✔		✔	✔	
DepthK				✔	✔						✔					✔		
DIVINE-explicit							✔				✔					✔		
DIVINE-SMT							✔				✔					✔		
ESBMC-kind				✔	✔						✔					✔		
JayHorn	✔	✔				✔		✔					✔	✔				
JBMC				✔							✔					✔		
JPF				✔			✔	✔			✔							
Lazy-CSeq				✔							✔					✔		
Map2Check				✔							✔							
PeSCo	✔	✔		✔	✔		✔	✔	✔		✔	✔	✔	✔		✔	✔	
Pinaka			✔	✔							✔							
PredatorHP									✔									
Skink	✔						✔							✔	✔			
Smack	✔			✔		✔					✔		✔			✔		
SPF			✔						✔							✔		
Symbiotic			✔					✔			✔							
UAutomizer	✔	✔									✔		✔	✔	✔		✔	
UKojak	✔	✔									✔		✔	✔				
UTaipan	✔	✔									✔		✔	✔	✔			
VeriAbs	✔			✔	✔		✔	✔										
VeriFuzz				✔				✔										✔
VIAP																		
Yogar-CBMC	✔			✔							✔		✔			✔		
Yogar-CBMC-Par.	✔			✔							✔		✔			✔		

Figure 1.1: Techniques that SV-COMP 2019 competitors offer. The table is adapted from competition report [Bey19].

However, one key point that is shared among many of these tools is the program representation — for instance, LLVM bitcode is a popular choice. This is due to its simplicity and preservation of program semantics. Its single static assignment form and described concrete semantics allow for a straightforward abstraction design. In practice, there are numerous tools that implement abstraction over LLVM bitcode [CDE08; Cha+21; Che+22; Gur+15; PF20], however, their abstraction is rarely exposed for convenient use.

In this thesis, I aim to investigate the idea that the intermediate representation can be used to represent the abstraction and potentially improve the efficiency and reusability of abstraction-based tools by leveraging the commonality of LLVM representation. I aim to adapt the techniques

[Cla+00]: Clarke et al. (2000), "Counterexample-Guided Abstraction Refinement"

[FQ02]: Flanagan et al. (2002), "Predicate Abstraction for Software Verification"

[Yan+08]: Yang et al. (2008), "Scalable Shape Analysis for Systems Code"

[Hen+02]: Henzinger et al. (2002), "Lazy Abstraction"

[Bur+90]: Burch et al. (1990), "Symbolic model checking: 10^{20} states and beyond"

[Kin76]: King (1976), "Symbolic Execution and Program Testing"

[MR18]: Majumdar et al. (2018), "Symbolic Model Checking in Non-Boolean Domains"

[CDE08]: Cadar et al. (2008), "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs"

[Cha+21]: Chalupa et al. (2021), "Symbiotic 8: Beyond Symbolic Execution (Competition Contribution)"

[Che+22]: Chen et al. (2022), "SYMSAN: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis"

[Gur+15]: Gurfinkel et al. (2015), "The SeaHorn Verification Framework"

[PF20]: Poeplau et al. (2020), "Symbolic execution with SymCC: Don't interpret, compile!"

used in verification tools for abstraction as stand-alone algorithms using LLVM IR, especially its concrete semantics, as a communication medium. This includes identifying common properties among these techniques, developing new self-contained algorithms, and designing an interface between the new algorithms and verification tools. Additionally, I plan to create a framework that facilitates constructing and testing new abstract domains. With the ability to use stand-alone abstraction, these abstract domains can then be easily integrated into dynamic analyses using the same intermediate representation. Furthermore, removing the abstraction engine from the verification tool can simplify the overall process, resulting in more robust verification.

A way to perform abstraction in the means of intermediate representation (LLVM IR) is to realize the abstraction as LLVM IR computation, i.e., to use the concrete semantics that the other tools understand. I will refer to this approach as *compilation-based abstraction* since one can see it as a compilation (static instrumentation) of abstract computation into LLVM IR.

In my research, I focus on a specific approach to extracting abstraction techniques out of verification tools, where the responsibility for abstract computation is shifted from the interpreter to a preceding static transformation — compilation-based abstraction. The static transformation is responsible for directly incorporating abstractions into the program using a common intermediate representation, such as LLVM bitcode, which facilitates the integration of the abstraction with various verification tools. As a result, the program will continue to maintain abstract data and perform abstract operations instead of concrete computation. Thus, a verifier can interpret the modified program without the need for any knowledge of abstract semantics.

Applications

I envision that a versatile abstraction-refinement tool or framework would benefit various applications. Currently, my primary focus is using it for software model checking, with both explicit-state and symbolic (bounded) backends. I will also explore possibilities of native execution of the abstraction. Nevertheless, I do not restrict the design to these applications.

Explicit-State Model Checking

For an explicit-state model checker, nondeterminism is expensive by default, and it relies on abstraction to make the problem of open-ended programs (even of the structured variety) tractable. Most, or all, input values need to be abstracted away for an explicit-state model checker to work reasonably efficiently.

The compilation-based abstraction fits explicit analysis perfectly since it is optimized for the execution of concrete semantics. The explicit-state model checkers can flawlessly interpret instrumented abstraction, eliminating input non-determinism.

Bounded Model Checking

For a bounded model checker, the complexity of dealing with input nondeterminism is deferred to its back-end decision procedure, usually an SMT solver. In some cases, the SMT solver is based on abstraction-refinement, for instance by bit-blasting *abstract* formulas into SAT problems (the abstraction is especially useful for the array theory). The principle is that the SMT solver has more insight into the problem than its back-end procedure can represent. This same principle can be replicated one level higher in a bounded model checker by exploiting knowledge about the program that is no longer present in the SMT representation.

One of the strategies behind this approach is to defer expensive transformations on the problem into the already-abstracted stages of processing. In a bounded model checker, loop unrolling is an example of such a costly transformation. Abstractions that remove entire loops from a program could substantially affect the problem size presented to the SMT solver and save memory and time on the model checker's side. In many cases, some loops in a program are entirely redundant for a particular correctness criterion, not unrolling those loops and not feeding their unrolling to the SMT solver could save a lot of work. Even an astute SMT solver will have a hard time figuring out that a particular subset of the SMT problem represents a single loop and that the entire lump would be a good candidate for SMT-level abstraction.

In essence, while program-level abstraction/refinement is not as crucial for a bounded model checker as for an explicit-state one, it could substantially improve its efficiency, especially over more complex programs.

Native Execution

One of the advantages of using a compilation-based approach to abstraction is the potential for improved performance by leveraging the native execution of the program. This is in contrast to interpretation-based abstraction tools, where the abstraction is implemented in an interpreter. By compiling the program with the abstraction built-in, we can take advantage of the program's native execution and potentially benefit from the use of binary-targeted tools, such as memory safety analysis using Valgrind [NS07], on the abstract program. The most recent research shows that similar approaches are getting wider adoption and outperform interpretation-based techniques [Che+22; PF20; PF21].

[NS07]: Nethercote et al. (2007), "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation"

[Che+22]: Chen et al. (2022), "SYMSAN: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis"

[PF20]: Poeplau et al. (2020), "Symbolic execution with SymCC: Don't interpret, compile!"

[PF21]: Poeplau et al. (2021), "SymQEMU: Compilation-based symbolic execution for binaries"

1.1 Structure of the Thesis

In this thesis, we will discuss the methods of program abstraction and how they can be adapted to compilation-based abstraction. Each chapter will focus on a particular aspect of abstraction or solves a problem related to the abstraction application for software verification. The objective is to provide a comprehensive presentation of compilation-based abstraction, including its subtleties and applications, all within a self-contained format. As such, the discussed techniques are implemented as part of the compilation-based abstraction framework LART.

The first two chapters unify the introduction to my published work. The rest of the thesis will focus on the applications of compilation-based abstraction and the research problems I had to solve to make the technique applicable. Each chapter summarizes its relevant state-of-the-art and presents my solutions to the problems concerning the adaptation to the compilation-based approach.

Chapter 2 presents an introduction to discussed formal methods, abstraction and lattice theory.

Chapter 3 establishes a simplified language that we will encounter throughout the thesis. The chapter formalizes the concrete trace and operation semantics of programs and how we relate them to program analysis. The proposed language joins my published work in a unified way. The unified description is an extension of the already published work and allows for linking the details between topics described in this thesis. Further, we describe presented formal methods from Chapter 2 in the discussed computational model.

Chapter 4 describes the abstraction of scalar values. This includes multiple non-relational abstract domains, like interval abstraction or sign domain. We state the limits of non-relational abstract domains in the compilation-based abstraction. In the chapter's second half, we devise a novel runtime refinement approach for non-relational abstract domains.

Chapter 5 extends the compilation-based abstraction to non-scalar analysis. The central pillars of this chapter are parametric domains for string and array abstraction. The chapter describes how to craft a domain that represents various aspects of objects using different domains, in this case, string characters and indices to the string. We will extend the ability to abstract entire functions, not just primitive operations. This will allow us to automatically lift entire library functions into abstract semantics and perform them more efficiently, e.g., libc string manipulating functions.

Chapter 6 covers the final aspect of program abstraction — dynamic memory. To demonstrate memory abstraction, we will present a method that enables reasoning about nondeterministic dynamic memory layout in a scalable manner: a parametric pointer arithmetic abstract domain.

Chapter 7 presents techniques for improving the precision and efficiency of compilation-based abstractions through various refinement methods. We identify two main categories of refinement: interdomain and intradomain. interdomain refinement involves augmenting abstract values within a domain to increase their precision, such as through backward constraint propagation (described in Chapter 4). The primary focus of this chapter, however, is on intradomain refinement, which involves improving the precision of analysis by considering interactions between multiple domains. To facilitate universal intradomain analysis, we introduce the concept of a metadomain (a domain of domains).

Chapter 8 explores the possible approaches to implementing program abstraction. Especially how to partition responsibilities between the compiler, interpreter, and the system under test. We examine the advantages and disadvantages of each partitioning. Moreover, we discuss how formal methods, such as abstract execution and model checking, interact with compiled abstraction.

Chapter 9 focuses on the technical details of compilation-based abstraction, specifically on syntactic abstraction. Syntactic abstraction is responsible for incorporating abstract semantics into the concrete program. We explore how program abstraction interacts with program control flow, memory, and other program domains, particularly how to resolve interactions between concrete and abstract computation.

Chapter 10 presents the application of compilation-based abstraction beyond the simple program analysis. We will explore its usage in program decompilation and program lifting, native execution, and program model checking. We will delve into aspects of syntactic program refinement, which increases the precision of syntactic abstraction from Chapter 9. Furthermore, the chapter will outline potential avenues for future research and provide an overview of ongoing research focused on incremental program analysis.

Although chapters draw from the corresponding papers, the content has been revised to ensure coherence with the rest of the work and supplemented with additional information not present in the published papers. The referenced publications were primarily focused on a particular application of compilation-based abstraction and did not delve deeply into its details. Therefore, this thesis features two chapters to fill the whole picture, namely Chapter 8, which compares different approaches to abstraction and their adaptability to compilation-based analysis, and Chapter 9, which explores the topic of syntactic abstraction, an integral part of the presented approach. The use of metadomains for intradomain refinement, as presented in Chapter 7, is also a novel and previously unpublished aspect of the discussed abstractions.

1.2 Contribution

In this section, I will summarize my contributions to the publications that form the basis of this thesis. Further, I will list publications in which I participated but which are not directly related to the topic of compilation-based abstraction presented in this thesis. I acknowledge my PhD consultant Petr Ročkai for collaborating with me on most of the publications. We share most of the contributions, and he also provided valuable insights into the algorithms and overall design of the topics presented in the listed publications.

Core of the Thesis

⊸ ICTAC 2018 | Symbolic Computation via Program Transformation | H. Lauko, P. Ročkai, and J. Barnat | [LRB18]

My contribution: I designed, implemented, and evaluated symbolic representation and program transformation to perform symbolic computation in the program semantics. I have been involved in writing the text.

76 % (algorithms: 75 %, implementation: 90 %, evaluation: 90 %, text: 50 %)

⊷ TACAS 2019 | Extending DIVINE with Symbolic Verification Using SMT | H. Lauko, V. Štill, P. Ročkai, and J. Barnat | [Lau+19]

My contribution: I modified DIVINE to participate in SV-COMP by utilizing symbolic abstraction and program transformation technique. Furthermore, I authored the paper describing this contribution.

93 % (algorithms: 100 %, implementation: 90 %, text: 90 %)

⊷ SPIN 2019 | String Abstraction for Model Checking of C Programs | P. Ročkai, H. Lauko, M. Olliaro and A. Cortesi | [Roč+19]

My contribution: I adapted, implemented, and evaluated the M-String domain for model checking (dynamic analysis) using the compilation-based abstraction technique. I have been involved in writing the text.

70 % (algorithms: 50 %, implementation: 90 %, evaluation: 100 %, text: 40 %)

⊷ AS 2020 | Abstracting Strings for Model Checking of C Programs | H. Lauko, M. Olliaro, A. Cortesi and P. Ročkai | [Lau+20]

This is an extended version of the conference paper [Roč+19].

My contribution: I adapted, implemented, and evaluated the M-String domain for model checking (dynamic analysis) using the compilation-based abstraction technique. I have been involved in writing the text.

70 % (algorithms: 50 %, implementation: 90 %, evaluation: 100 %, text: 40 %)

⊷ TOSEM 2022 | Verification of Programs Sensitive to Heap Layout | H. Lauko, L. Korenčik and P. Ročkai | [LKR22]

My contribution: I contributed to the design and refinement of the domain, and personally implemented the pointer arithmetic domain along with the primary portion of the program refinement algorithm. Furthermore, I was responsible for designing most of the experiments and participated in the writing of the paper.

65 % (algorithms: 50 %, implementation: 80 %, evaluation: 80 %, text: 50 %)

⊷ TACAS 2022 | LART: Compiled Abstract Execution | H. Lauko, P. Ročkai | [LR22]

My contribution: I have implemented the native execution of program abstraction and prepared it for the SV-COMP competition. Furthermore, I authored the paper describing this contribution.

96 % (algorithms: 100 %, implementation: 100 %, text: 90 %)

⊷ ASV Book (forthcoming) | DIVINE: Model Checker for C++ | H. Lauko, P. Ročkai, V. Štill and J. Barnat | [Lau+ng]

My contribution: I coauthored and was the main editor of a chapter on DIVINE in a forthcoming book on automatic software verification.

65 % (text: 65 %)

Other Related Publications

⊷ QRS 2020 | On Symbolic Execution of Decompiled Programs | L. Korenčik, P. Ročkai, H. Lauko and J. Barnat | [Kor+20]

My contribution: I was involved in the discussions regarding the integration of compilation-based abstraction into the proposed decompilation process. Further, I contributed to the adaptation of the abstraction technique employed in the decompilation.

⊷ ATVA 2017 | Model Checking of C and C++ with DIVINE 4 | Z. Baranová, J. Barnat, K. Kejstová, T. Kučera, H. Lauko, J. Mrázek, P. Ročkai and V. Štill | [Bar+17]

My contribution: I was one of the main contributors to DIVINE 4 release.

⊷ TACAS 2017 | Optimizing and Caching SMT Queries in SymDIVINE | J. Mrázek, M. Jonáš, V. Štill, H. Lauko and J. Barnat | [Mrá+17]

My contribution: I was involved in writing the paper, and a minor part of its implementation and evaluation.

⊷ SPIN 2016 | SymDIVINE: Tool for Control-Explicit Data-Symbolic State Space Exploration | J. Mrázek, P. Bauch, H. Lauko and J. Barnat | [Mrá+16]

My contribution: I was involved in writing the paper, and a minor part of its implementation and evaluation.

⊷ CAV 2022 | From Spot 2.0 to Spot 2.10: What's New? | A. Duret-Lutz et al. | [Dur+22]

My contribution: As part of my internship, I made contributions to the development of several translation algorithms within the Spot tool. Further, I implemented an algorithm for generating Büchi automatons that could be consumed by DIVINE.

Software

In this section, I will provide a summary of the software that I have developed or significantly modified during the course of my doctoral studies. All the software is licensed under open-source licenses. The majority of publications include slight modifications to the listed tools. We provide links to these modified artifacts in the supplementary materials for each publication.

DIVINE

DIVINE is an explicit state model checker designed for the verification of real-world code written in high-level programming languages, particularly in C/C++.

My contribution: I have implemented the majority of support for abstractions and program transformations. Further, I have made major contributions towards supporting symbolic computation and DIVINE's integration with multiple SMT solvers.

LINKS:

- https://divine.fi.muni.cz/
- https://github.com/paradise-fi/divine

Throughout my doctoral studies, I implemented two versions of a program transformation that is discussed in this thesis. The first version was closely integrated into the DIVINE toolchain, while the second version was a more standalone tool targeting the native execution of abstraction. We provide links to both versions in the following list.

LART

LART is an LLVM Abstraction & Refinement Tool. The goal of this tool is to provide LLVM-to-LLVM transformations that implement various program abstractions. It is the main driver for the proposed compilation-based abstraction.

My contribution: I developed the syntactic abstraction of programs, which is used for instrumenting abstract computation into programs. I also developed the dataflow analysis that the abstraction uses to identify the instructions that should be abstracted. I am the only author and designer of the second version of LART used to perform native abstract execution.

LINKS:

- https://github.com/paradise-fi/divine/tree/master/lart
- https://github.com/xlauko/lart

LAMP

A library of abstract metadomain packages (LAMP) provides predefined metadomains and building blocks for their construction. A metadomain, which is a domain of domains, is used to resolve interactions between domains and defines which domains are used for abstracting specific program primitives.

My contribution: I am the primary author and designer of the library.

LINKS:

- https://github.com/xlauko/lart/tree/master/lamp
- https://github.com/paradise-fi/divine/tree/master/dios/lamp

LAVA

A Library of Abstract VAlues (LAVA) is a header-only C++ library which provides a number of abstract (value) domains. Each domain is implemented as a class (or class template, for parametric domains) and provides a number of methods which implement the individual operations of the domain.

My contribution: I am the primary author and designer of the library.

LINKS:

- https://github.com/xlauko/lart/tree/master/lava
- https://github.com/paradise-fi/divine/tree/master/dios/lava

DIPOT

This tiny tool implements an interface between SPOT and DIVINE. It creates a C program that simulates transition-based Büchi automaton from the LTL formula, that is accepted by DIVINE.

My contribution: I am the primary author and designer of the tool.

LINK: https://github.com/paradise-fi/dipot

Related Public Talks

EGRAPHS 2022 | On the Optimization of Equivalent Computations

LLVM Meeting 2022 | VAST: MLIR for program analysis of C/C++

Preliminaries 2

This chapter describes the common notation and conventions employed throughout the thesis. Additionally, it offers a concise introduction to formal methods and techniques pertinent to our discussion, with a particular emphasis on program domains and the underlying principles of abstraction. This includes a summary of the lattice theory employed in the study and a detailed explanation of value domains.

2.1 Notation & Conventions

The foundation of program abstraction lies in the use of various abstract domains. In order to make it easier to identify and relate the different domains discussed throughout the thesis, we will use a unified notation for specific categories, despite potential variations in the original publications.

We will represent generic domains with calligraphic font, for instance, $\mathcal{C}$ for the concrete domain, $\mathcal{B}$ for a Boolean domain, or $\mathcal{A}$ for an abstract domain. As we aim to perform analysis of `C` programs, the common domain of computation is a domain of fixed-length integers (bit-vectors) — denoted as $\mathcal{BV}$. The power set of a set S is denoted as $\wp(S)$.

Domain names are used interchangeably to refer to both the domain as a whole and the set of values that make up the domain. To indicate that a value v belongs to the domain $\mathcal{D}$, we use the notation $v \in \mathcal{D}$. Nevertheless, the domain as a whole includes other components, such as operations and the ordering of values.

When the context does not make it clear, we use the symbol $\hat{\_}$ over elements related to abstraction to distinguish between abstract and concrete values. For instance, we use $\hat{v}$ to represent the abstraction of the concrete value v, and $\hat{+}$ to represent an abstract addition operator. In addition, we use a lower index to indicate the specific domain of an operation. For example, abstract addition in the domain $\mathcal{A}$ is denoted as $\hat{+}_{\mathcal{A}}$.

In a logic environment, `T` represents a true value from the Boolean domain, and `F` represents a false value. However, during abstraction, one may encounter situations where an abstract predicate can be both true and false simultaneously. To handle such cases, we utilize a three-value logic, where the ambiguous value is represented by `M` (maybe).

Each chapter commences with a list of the original research materials that I have published regarding the topic, along with an outline of the research objectives that the chapter endeavors to address. Whenever a citation is first mentioned or when it can improve the reading experience, it is expanded into a more detailed format in the margin column. Most of the chapters or more extended sections begin with research questions, marked as **RQ**, which the section aims to resolve. Furthermore, I summarize each chapter's key takeaways in the boxes at the chapter's end.

2.2 Formal Methods

The idea of mathematically-based reasoning about program execution dates back to the origin of computer science [Tur49]. The groundbreaking work of Hoare [Hoa69] and Floyd [Flo93] initiated the first endeavors toward logic-based formal methods. However, due to a lack of automation, these techniques did not enjoy widespread adoption. In fact, Rice [Ric53] proved that all non-trivial properties of programs are undecidable, thereby establishing the fundamental limitation that prevents full automation of program analysis. Nonetheless, this limitation did not impede further research in the field. Instead, it gave rise to a variety of methods for circumventing the theoretical constraints of program analysis. Depending on which aspect we sacrifice, such as automation, generality, or completeness, we can achieve distinct flavors of program analysis, as categorized by Cousot [CC10]:

[Tur49]: Turing (1949), "Checking a large routine"

[Hoa69]: Hoare (1969), "An Axiomatic Basis for Computer Programming"

[Flo93]: Floyd (1993), "Assigning Meanings to Programs"

[Ric53]: Rice (1953), "Classes of recursively enumerable sets and their decision problems"

[CC10]: Cousot et al. (2010), "A gentle introduction to formal verification of computer systems by abstract interpretation"

- *Deductive methods* build upon the logical foundations laid by Hoare and Floyd. These methods are partly automatized but rely on human guidance in the proof process. This category encompasses interactive proof assistants like Coq [BC10] or Lean [MU21], as well as interactive theorem provers such as PVS [ORS92] or HOL [NPW02].

[BC10]: Bertot et al. (2010), *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*

[MU21]: Moura et al. (2021), "The Lean 4 Theorem Prover and Programming Language"

[ORS92]: Owre et al. (1992), "PVS: A prototype verification system"

[NPW02]: Nipkow et al. (2002), *Isabelle/HOL: a proof assistant for higher-order logic*

- *Model Checking* restricts program analysis problems to decidable fragments [CES86]. At first, this technique was only applied to finite models. However, it was since extended to infinite regular models using symbolic model checking [McM93]. To analyze real-world programs, model checkers usually limit the analysis to a finite portion of their executions and perform a form of bounded model checking, as is done in tools such as CBMC [CKL04]. In order to perform the analysis, a state-space must be extracted from the program being tested.

[CES86]: Clarke et al. (1986), "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications"

[McM93]: McMillan (1993), *Symbolic model checking*

- *Static Analysis* is a program verification technique in which we perform direct analysis of the program under test, we want to examine all possible executions automatically and utilize approximations to make the analysis efficient and usable. However, as a result, the abstraction is incomplete and can miss some properties, resulting in false alarms, i.e., the program is correct, but the analyzer cannot prove it.

[CKL04]: Clarke et al. (2004), "A Tool for Checking ANSI-C Programs"

This thesis is centered around the core concepts of abstract interpretation, which I have adapted to the compilation and runtime environment. I have designed program verification abstractions that use compilation-based technique capabilities. But I do not restrict the technique just to static analysis. One of my primary goals is to provide abstraction capabilities to explicit-state model checkers, too.

2.3 Program Domains

The unifying concept behind the variety of abstraction techniques is that of a domain, which describes the behavior of a program in terms of values which the program manipulates: their representation and the operations defined on them. Domains are of two basic types: *concrete domains* (which describe a program as it actually is) and *abstract domains* which describe the program in a form more suitable for analysis. Please note that the

distinction between concrete and abstract domains is solely in their use: formally, they are the same kind of object.

The starting point of abstract interpretation, then, is the concrete semantics of a program, formally described by a concrete domain $\mathcal{C}$. Even though technically, each programming language admits exactly one concrete domain, it is often more practical to create specialized concrete domains[1] that only focus on the specific behaviors relevant to the problem at hand.

1: In theory, the concrete domain should be unambiguously given by the semantics of the programming language in question. In practice, a programming language may not have complete, unambiguous semantics. Even if it does, such semantics are usually very complicated, and a concrete domain derived this way would be an impractical starting point for abstraction.

Domains can be combined to form new domains in a number of ways. Perhaps the simplest, and most important, is a direct (cartesian) product $\mathcal{D}_1 \times \mathcal{D}_2$, where values are pairs and operations are performed elementwise. A domain constructed like this models two variables, where the first variable is represented using $\mathcal{D}_1$ and the second using $\mathcal{D}_2$. This construction readily generalizes to any number of variables (including infinitely many). This way, the program state as a whole can be represented in terms of a single domain. It may be beneficial to attach a label to each position in the resulting n-tuples (corresponding to, e.g., variable names), though this is not essential. We will call domains of this type *composite*, since they are built up from simpler constituent domains.

With this in mind, it is customary to describe the entire behavior of a program in terms of a single (concrete or abstract) domain. This domain then, in addition to modelling individual values[2] which appear in the program, covers such aspects as variable identifiers or memory addresses. However, this all-encompassing domain is usually constructed from simpler domains, as outlined above: these simpler domains may describe the behavior of, say, an individual scalar value. We will call the latter *value domains* (fully defined later in Definition 2.3.6), though like with concrete vs abstract domains, there is no strict formal distinction.

2: There are typically three types of values in a program: those bound to names (variables), unnamed intermediate values (e.g., the result of addition in `(a + b) * c`) and unnamed values stored in pointer-accessible memory.

In abstraction, we always operate with at least one concrete domain $\mathcal{C}$ and one abstract domain $\mathcal{A}$. The relationship between these two domains is given by *abstraction* $\alpha : \mathcal{C} \to \mathcal{A}$ and *concretization* $\gamma : \mathcal{A} \to \mathcal{C}$.[3]

An abstraction is said to be *overapproximating* if the set of abstracted values contains all the original concrete values, that is, $c \subseteq \gamma(\alpha(c))$. On the other hand, an *underapproximating* abstraction may exclude some concrete values but cannot generate any new values: $\gamma(\alpha(c)) \subseteq c$.

3: Note that we think of elements of domains as of sets of values. In an abstract domain, elements inherently represent some set of concrete values, which is not true of the concrete domain. Therefore, we think of an element of a concrete domain as a set of concrete values. For example, we say that abstraction of values $\{1, 2, 5\}$ into interval domain $\mathcal{A}_i$ is

$$\alpha_i(\{1, 2, 5\}) \triangleq [1, 5]$$

Similarly, the concretization is given as:

$$\gamma_i([1, 5]) \triangleq \{1, 2, 3, 4, 5\}$$

2.3.1 Lattice Theory

In practice, it is convenient to think about program domains as lattices. We use lattices to describe the granularity of abstraction, the relationship between an abstracted set of values, and the relationships between domains. Throughout the thesis we will expect the reader to be familiar with basics of lattice theory [Bir40]. This section condenses the fundamental parts of the theory and notation we will use.

[Bir40]: Birkhoff (1940), *Lattice Theory*

Lattice theory is an abstraction of set theory. In the context of program analysis, it abstracts sets of program/variable properties $\mathbb{P}$. Given the implication $\sqsubseteq$ on these properties, we get the foundation of abstract analysis, the poset $\langle \mathbb{P}, \sqsubseteq \rangle$.

Definition 2.3.1 *A **poset** $\langle S, \sqsubseteq \rangle$ is a set S equipped with a partial order $\sqsubseteq$ that is* reflexive, antisymmetric *and* transitive.

In program analysis, we will operate with poset elements as we determine more generic or more precise program properties. These transformations are provided as *meet* and *join* operations on program properties.

The most precise generalization of multiple facts is their *least upper bound* in the poset, also called *join*. We denote join of two properties $x \sqcup y$. Whereas, the intersection of program properties can be computed as their *greatest lower bound*, also called *meet*, denoted as $x \sqcap y$.

[Cou21]: Cousot (2021), *Principles of Abstract Interpretation*

Lattices are posets with the following properties [Cou21]:

- *join semilattice* $\forall x, y \in S. x \sqcup y$ exists in S,
- *meet semilattice* $\forall x, y \in S. x \sqcap y$ exists in S,
- *lattice* $\forall x, y \in S. x \sqcup y$ and $x \sqcup y$ exists in S.

Definition 2.3.2 *A poset $\langle S, \sqsubseteq \rangle$ is a **complete lattice** if every subset $s \in \wp(S)$ has both a least upper bound and a greatest lower bound.*

Therefore, every complete lattice has a supremum $\sqcup S = \top$ called *top*, and an infimum $\sqcup \emptyset = \bot$ called *bottom*. From the program analysis perspective, the top represents the fact that the abstracted value satisfies all properties, e.g., in interval analysis, the top spans from $-\infty$ to ∞, representing all possible variable values. While the bottom usually denotes an unfeasible state.

Galois connection formalizes correspondence between concrete properties and abstract properties in case there is always most precise abstract property overapproximating any concrete property [Cou21].

Definition 2.3.3 (Galois connection) *Given posets $\langle \mathcal{C}, \subseteq \rangle$ (the concrete domain) and $\langle \mathcal{A}, \sqsubseteq \rangle$ (the abstract domain), the pair $\langle \alpha, \gamma \rangle$ of increasing functions $\alpha : \mathcal{C} \to \mathcal{A}$ (the **abstraction**) and $\gamma : \mathcal{A} \to \mathcal{C}$ (the **concretization**) is a Galois connection if and only if*

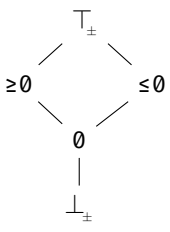
$$\forall c \in \mathcal{C}. \forall a \in \mathcal{A}. \alpha(c) \sqsubseteq a \iff c \subseteq \gamma(a)$$

which we write

$$\langle \mathcal{C}, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{A}, \sqsubseteq \rangle$$

4: This kind of Galois connection, introduced by Jürgen Schmidt [Sch53], is also called *increasing Galois connection*, because α has increasing nature.

Intuitively Galois connection[4] describes how concrete properties in $\mathcal{C}$ are approximated by abstract properties in $\mathcal{A}$. The concretization γ gives the concrete meaning to abstract properties $a \in \mathcal{A}$ in such a way that $\gamma(a)$ is the smallest concrete representation that can be overapproximated by a.



$$\begin{aligned}
\gamma_{\pm}(\bot_{\pm}) &\triangleq \emptyset \\
\gamma_{\pm}(\texttt{≥0}) &\triangleq \{n \mid n \geq 0\} \\
\gamma_{\pm}(\texttt{0}) &\triangleq \{0\} \\
\gamma_{\pm}(\texttt{≤0}) &\triangleq \{n \mid n \leq 0\} \\
\gamma_{\pm}(\top_{\pm}) &\triangleq \mathbb{Z}
\end{aligned}$$

Figure 2.1: Example of a simple sign abstraction lattice $\langle \mathcal{A}^{\pm}, \sqsubseteq^{\pm} \rangle$. The sound abstractions of $\{1, 2\}$ are both $\top_{\pm}$ and $\texttt{≥0}$ because $\{1, 2\} \subseteq \gamma_{\pm}(\texttt{≥0}) \subseteq \gamma_{\pm}(\top_{\pm})$ holds for both abstract properties.

Definition 2.3.4 *Abstract property $a \in \mathcal{A}$ is **sound overapproximation** of concrete property $c \in \mathcal{C}$ whenever $c \subseteq \gamma(a)$.*

In the context of the Galois connection, $\alpha(c)$ represents the most precise and sound overapproximation of c in the abstract domain $\mathcal{A}$, also known as *best approximation.* The increasing nature of γ is essential as it ensures the preservation of abstract implications in a concrete domain. Consequently, proof in the abstract domain is also valid in the concrete domain.

Formally, we call abstraction $a_1 \in \mathcal{A}$ is *better* (more precise) than abstraction $a_2 \in \mathcal{A}$ of property $c \in \mathcal{C}$, if both abstractions are sound overaproximations of c and $a_1 \sqsubseteq a_2$.

Remark 2.3.1 (Galois connection composition) *The essential property of abstraction defined via the Galois connection is its composition. It holds that the composition of two Galois connections* $\langle \mathcal{D}_1, \sqsubseteq \rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle \mathcal{D}_2, \preceq \rangle$ *and* $\langle \mathcal{D}_2, \preceq \rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle \mathcal{D}_3, \subseteq \rangle$ *is the Galois connection:*

$$\langle \mathcal{D}_1, \sqsubseteq \rangle \underset{\alpha_2 \circ \alpha_1}{\overset{\gamma_1 \circ \gamma_2}{\leftrightarrows}} \langle \mathcal{D}_3, \subseteq \rangle$$

Therefore, even when composing abstraction, we retain the property of best overapproximation.

Furthermore, Galois connections can be extended to the cartesian product of domains, which is useful when defining computation over tuples of values possibly belonging to different domains. The Galois connection preserves the abstraction properties even in this scenario.

Theorem 2.3.1 (Cartesian product of Galois connections) *For two Galois connections* $\langle \mathcal{C}_1, \subseteq_1 \rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle \mathcal{A}_1, \sqsubseteq_1 \rangle$ *and* $\langle \mathcal{C}_2, \subseteq_2 \rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle \mathcal{A}_2, \sqsubseteq_2 \rangle$ *it holds that* $\langle \mathcal{C}_1 \times \mathcal{C}_2, \subseteq_\times \rangle \underset{\alpha_\times}{\overset{\gamma_\times}{\leftrightarrows}} \langle \mathcal{A}_1 \times \mathcal{A}_2, \sqsubseteq_\times \rangle$ *is also Galois connection, where* $\subseteq_\times$ *and* $\sqsubseteq_\times$ *are defined component-wise, and abstraction function:*

$$\alpha_\times(\langle c_1, c_2 \rangle) \triangleq \langle \alpha_1(c_1), \alpha_2(c_2) \rangle$$

and concretization:

$$\gamma_\times(\langle a_1, a_2 \rangle) \triangleq \langle \gamma_1(a_1), \gamma_2(a_2) \rangle$$

In order to perform abstract execution, we must provide an abstract alternative to concrete operations. Fortunately, the notion of abstraction can be naturally adapted from domain elements to domain operators:

Definition 2.3.5 *Given an abstract domain* $(\mathcal{A}, \sqsubseteq)$ *with concretization function* γ*, for concrete operator* $f : \mathcal{C} \to \mathcal{C}$ *and its abstraction* $\hat{f} : \mathcal{A} \to \mathcal{A}$ *it holds:*

1. $\hat{f}$ *is a* ***sound abstraction*** *of* f *if* $\forall a \in \mathcal{A}. f(\gamma(a)) \subseteq \gamma(\hat{f}(a))$,
2. $\hat{f}$ *is an* ***exact abstraction*** *of* f *if* $f \circ \gamma \equiv \gamma \circ \hat{f}$.

In cases where domains form a Galois connection, it becomes possible to apply the concept of "best abstraction" to operators as well. Specifically,

the optimal abstraction of an operator f is given by $\alpha \circ f \circ \gamma$. However, although this definition provides valuable guidance for the creation of abstractions, implementing them directly in practice is usually not feasible. This is because the components of best abstraction, namely α, and γ, are unlikely to be computable due to their non-constructive mathematical definition. As a result, it is more practical to implement abstract operators using approximations that aim to capture the essence of best abstraction — approximate it according to the definition.

Example 2.3.1 Let us examine a basic operation that increments its argument: $f(X) \triangleq \{x + 1 \mid x \in X\}$. The simplest sound abstraction is to consider all possibilities. For instance, in the interval domain, $\hat{f}([a, b]) \triangleq [a + 1, b + 1]$ is an exact and sound abstraction of f. A sound but not exact abstraction would be if $\hat{f}$ returned $[-\infty, \infty]$, which overapproximates all possible outcomes.

Now consider multiplication by two: $g(X) \triangleq \{2x \mid x \in X\}$. The expected abstraction, $\hat{g}([a, b]) \triangleq [2a, 2b]$, is the best possible abstraction in the interval domain, but it is not entirely precise. This inaccuracy results from the fact that the operation's image does not always span the entire interval. For instance, $g(\{1, 2\}) \triangleq \{2, 4\}$, while $\hat{g}([1, 2]) \triangleq [2, 4]$ contains an additional value, namely 3.

It is desirable to keep the abstraction as modular and composable as possible. As we will see in the next section, the program's semantics are given as a sequence of atomic semantic steps from a limited set of language operations. This approach aligns with the desire for modular abstraction since we can abstract only the atomic language operations and compose the abstraction using the same rules as in concrete semantics. Operation composition has two interesting properties that are worth noting [Min04]:

[Min04]: Miné (2004), "Weakly relational numerical abstract domains"

Remark 2.3.2 *Consider concrete operations $f, g : \mathcal{C} \to \mathcal{C}$ and their abstractions $\hat{f}, \hat{g} : \mathcal{A} \to \mathcal{A}$:*

- *If $\hat{f}$ and $\hat{g}$ are **sound abstractions** of f and g, and f is monotonic, then $\hat{f} \circ \hat{g}$ is a **sound abstraction** of $f \circ g$.*
- *If $\hat{f}$ and $\hat{g}$ are **exact abstractions** of f and g, then $\hat{f} \circ \hat{g}$ is an **exact abstraction** of $f \circ g$.*

One would think from previous that the straightforward composition of best abstractions would yield a best abstraction. However, this is not the case. When best abstractions are combined, we get $\alpha \circ f \circ \gamma \circ \alpha \circ g \circ \gamma$, which leads to a superfluous transformation through the abstract domain, thereby introducing imprecision. The optimal abstraction for the composition of f and g would be $\alpha \circ f \circ g \circ \gamma$.

It is important to note that the composition of two optimal abstractions may not necessarily result in an optimal abstraction, whereas the composition of an optimal and exact abstraction results in an optimal abstraction.

The lack of general composability of best abstraction has practical implications for the design of abstractions. Specifically, the analysis's precision depends on the abstraction's granularity. Finer abstractions are more likely to introduce imprecision, while larger block abstractions are less modular. Ultimately, there is always a trade-off between modularity and precision. To conclude, we must remember that despite our best efforts, abstraction rarely gives us the most precise result, even if we solely employ best abstractions.

2.3.2 Value Domain

As stated before, a *value domain* is a fundamental building block of program abstraction. We can now define it more formally:

Definition 2.3.6 *An **abstract value domain** $\mathcal{D}$ is given by:*

- *a likewise named set $\mathcal{D}$ of computer-representable abstract values,*
- *an effective partial order $\sqsubseteq$ on $\mathcal{D}$,*
- *a monotonic concretization function $\gamma_{\mathcal{D}} : \mathcal{D} \to \mathcal{C}$,*
- *a monotonic abstraction function $\alpha_{\mathcal{D}} : \mathcal{C} \to \mathcal{D}$,*
- *a smallest $\bot_{\mathcal{D}} \in \mathcal{D}$ and a largest element $\top_{\mathcal{D}} \in \mathcal{D}$,*
- *a sound and effective abstraction of program operations,*
- *a sound and effective abstraction of set union $\cup_{\mathcal{D}}$ and set intersection $\cap_{\mathcal{D}} : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$, such that:*

$$\forall a_1, a_2 \in \mathcal{D} : \gamma_{\mathcal{D}}(a_1) \cup \gamma_{\mathcal{D}}(a_2) \subseteq \gamma_{\mathcal{D}}(a_1 \cup_{\mathcal{D}} a_2)$$
$$\forall a_1, a_2 \in \mathcal{D} : \gamma_{\mathcal{D}}(a_1) \cup \gamma_{\mathcal{D}}(a_2) \subseteq \gamma_{\mathcal{D}}(a_1 \cup_{\mathcal{D}} a_2)$$

Summary

This chapter introduced the conventions adopted throughout the thesis, such as the syntactic differentiation between abstract and concrete environments. Additionally, we discussed the lattice-based foundations of value-centered abstraction. The key takeaway is that the lattice framework enables us to combine multiple domains to facilitate multi-variable analysis while upholding the guarantees of elementary domains. Moreover, the chapter emphasizes the significance of abstraction granularity, as finer abstraction can significantly impact the precision of the overall abstraction due to the properties of the composition of best abstractions.

Computation Model 3

This chapter centers on the concrete semantics of programs and related analysis program analysis techniques. To that end, we define syntax and semantics for a simplified language used for examples. This allows us to unify semantics in the subsequent chapters, as the definitions in the original publications vary slightly from those presented here. As the implementation part of this thesis is based on the LLVM intermediate representation, which is an assembly-like low-level language used for unified program optimization in the Clang compiler, the language used to formalize the program semantics will closely resemble LLVM IR. In the syntax, we will omit LLVM technicalities irrelevant to program analysis, primarily optimization, and architecture-dependent artifacts. Later in the chapter, we introduce analysis techniques related to the topic, such as dataflow analysis, abstract interpretation, and software model checking.

The content of this chapter draws upon all of my publications as it unifies their definitions. The semantics presented here is primarily influenced by the semantics provided in the publication on heap abstraction, while the program analysis techniques discussed later in the chapter are adapted from the latter two publications:

- Henrich Lauko, Lukáš Korenčik, and Petr Ročkai. "Verification of Programs Sensitive to Heap Layout." In: *ACM Transactions on Software Engineering and Methodology* 31 (4 2022) [LKR22].
- Henrich Lauko. *Abstractions via Program Transformations.* Brno, 2020. (PhD Thesis Proposal) [Lau20].
- Henrich Lauko, Petr Ročkai, Vladimír Štill, and Jiří Barnat. "DIVINE – Model Checker for C++." In: *Automatic Software Verification.* Ed. by Dirk Beyer. (forthcoming) [Lau+ng].

3.1 Program Representation

The verification of C++ (and to a smaller degree of C) requires handling of complex syntactical constructs and statements which may often represent multiple steps (e.g., statement `++x` can represent three operations – load, increment, and store – an effect which can be critical in parallel programs). Therefore, it can be advantageous to first translate C or C++ code into a more explicit form, which can be handled more easily (i.e., one in which operations admit small step semantics of the verified program). Most state-of-the-art program analysis tools opt for using intermediate representations (IRs). These can be adapted from compiler infrastructures like LLVM IR [LA04] or verification targeted like Boogie [Lei08], CIL [Nec+02], or Crab IR [GN21]. One way or another, they occupy a sweet spot between architecture-specific instruction sets and high-level sources. IR typically captures program behavior at a higher level while using fewer instructions, making it easier to carry out complete and reliable program analysis.

[LA04]: Lattner et al. (2004), "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation"

[Lei08]: Leino (2008), "This is Boogie 2"

[Nec+02]: Necula et al. (2002), "CIL: Intermediate language and tools for analysis and transformation of C programs"

[GN21]: Gurfinkel et al. (2021), "Abstract interpretation of LLVM with a region-based memory model"

In our case, such a form is the LLVM Intermediate Representation. It is a low-level typed language used by the LLVM compilation infrastructure (which also includes the Clang compiler for C, C++, and Objective-C). The LLVM IR can be seen as a language midway between C and assembly languages – it has some features of high-level programming languages (such as static types and the ability to represent code for different platforms), but it contains simple instructions and can be reasonably well processed, transformed, and optimized. In the LLVM compilation infrastructure, LLVM IR is mainly used for optimizations and as a common input for the code generator, which generates the given platform's assembly.

The advantage of LLVM IR for program analysis is that it faithfully represents the program semantics and does not require a custom frontend to parse programs. Moreover, due to broad adoption, it also allows the reuse of other analysis tools.

There are, of course, some disadvantages of using LLVM IR for program analysis. First, some information can be lost in translation from C++ to LLVM IR simply because it cannot be represented in LLVM IR and is not relevant for program optimization and code generation. In some cases, this information is important for program analysis. For example, the information about the exact scopes of variables in functions might be vital: if a local variable is created in a loop or more generally in a higher scope, and a pointer to it is taken, this pointer should not be used after the scope ends, or even in the next iteration of the loop.

Another drawback is that even though LLVM IR is intended to be platform-agnostic, specific details about the target platform can still leak into the representation. This includes the sizes of data structures specific to the target platform, which are calculated by the compiler using information about platform-dependent types or the handling of C-style variadic functions.[1]

1: These are mostly limitations of the Clang compiler, which sometimes emits platform-specific code even if platform-agnostic code is also possible.

There is no formal semantics of LLVM IR or any formal guarantee of correct translation from C/C++ to LLVM IR. This, however, is in line with the reality of analysis of C/C++ in general – international standards define these languages, but these standards are not rigorous formal definitions of language semantics. Many aspects of translation are implementation-defined – meaning that compilers are free to choose the behavior which is later hardly relatable to the source program. Nevertheless, by translating to LLVM IR, it is possible, at least in some cases, to use this IR both for verification and for building the resulting program, which eliminates part of problems which could be introduced by compilers as problems introduced by translation to LLVM IR can be detected by such analysis.

To demonstrate techniques in this thesis we will use a simplified language similar to LLVM IR bitcode. Syntactically, LLVM IR bitcode is in a partial single static assignment form (SSA) [Cyt+91]: the result of each statement is stored in one of the registers (described by the set Reg). The SSA form is inherently partial due to the dynamic untyped memory present in LLVM IR, including both stack and heap allocations. For those familiar with earlier versions of LLVM IR, it's worth noting that the latest version (LLVM 15) now defaults all pointers to opaque type, with only memory accessing operations remaining typed. This new approach enables storing and loading objects of different types from the same memory location,

[Cyt+91]: Cytron et al. (1991), "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph"

reducing issues associated with unfaithful information present in the previous design of weakly typed pointers. Ultimately, this change results in simplified bitcode and facilitates program analysis, such as alias analysis.

We will differentiate the contents of SSA registers and memory since the abstraction of register values requires a different toolset for efficient analysis. Both memory and registers can hold values from some concrete domain $\mathcal{C}$. We further distinguish scalar integer values $\mathcal{C}_\mathsf{s} \subseteq \mathcal{C}$, boolean values $\mathcal{C}_\mathsf{b} \subseteq \mathcal{C}_\mathsf{s}$, and pointer values $\mathcal{C}_\mathsf{p} \subseteq \mathcal{C}$. For simplicity, we will consider aggregate values only as sequences of scalar values in the memory. Based on the type of stored value, we recognize scalar registers Reg_s that hold values from the set of concrete values $\mathcal{C}_\mathsf{s}$, boolean registers Reg_b holding boolean value $\mathcal{C}_\mathsf{b}$, and pointer registers Reg_p with values from $\mathcal{C}_\mathsf{p}$, where $\mathsf{Reg}_\mathsf{s} \cup \mathsf{Reg}_\mathsf{b} \cup \mathsf{Reg}_\mathsf{p} = \mathsf{Reg}$.

In addition to registers, the program can access addressable read-write memory, which consists of a collection of dynamically allocated memory blocks Blk, each of which is uniquely identified by a base address. The set of valid base addresses is denoted by Id. Every element within a block is known as a *cell*, identified by the base address and an offset $o \in \mathcal{C}_\mathsf{s}$. We denote a cell as $\langle a, o\rangle \in \mathsf{Cell}$. The effective address of a cell $\langle a, o\rangle$ is simply $a + o \in \mathcal{C}_\mathsf{p}$. In our model, the null pointer is represented as $\langle 0, 0\rangle$. This representation enables byte addressing of our memory, as we assume that all offsets are expressed in terms of the number of bytes.

Each memory block $\mathsf{blk} = \langle s, v\rangle \in \mathsf{Blk}$ consists of two components: size s, which represents the number of valid bytes in the allocated block, and a content value function $v : \mathcal{C}_\mathsf{s} \to \mathcal{C}_\mathsf{s}$. The content value function v maps valid offsets to the stored bytes in the block. However, these bytes may constitute larger objects, such as integers, pointers, or aggregates. Therefore, we often inspect the memory block content v from a high-level view, where we see a multibyte object at a particular offset. In our language, we define typed memory accesses that reconstruct larger objects. For example, a typed load from cell $\langle a, o\rangle$ with type `ty` will use the content value function v to obtain the bytes starting at offset $a + o$ up to $a + o + n$ where n is a number of bytes of type `ty`. These bytes are then reassembled into a value of type `ty`. Similarly, a typed store to address $\langle a, o\rangle$ with type `ty` will break the value of type `ty` into bytes and store them in the corresponding offsets.

This representation of memory results in an induced memory graph (cf. Figure 3.5), where memory objects (blocks) are the vertices, and pointers stored in those memory objects form the edges [Roč+18]. We label the edges by the offset at which the pointer giving rise to the edge is stored, while vertices are unlabeled – this way, the graph only captures the shape of the heap, not its content.

[Roč+18]: Ročkai et al. (2018), "DiVM: Model checking with LLVM and graph memory"

3.2 Program Syntax

This section outlines the syntax of the programming language used throughout this thesis. We simplify the LLVM IR language to semantically interesting operations for abstract value analysis. For the time being, we will only consider single-threaded integer programs and will

not delve into many orthogonal aspects of the modeled language. Other features, such as floating point values, missing arithmetic, relational, or control flow operations, can be incorporated into our language without substantial semantic changes.

In the following, we use $\mathtt{r} \in \mathsf{Reg}$ to denote registers and $\mathtt{c} \in \mathcal{C}$ to denote constants. When we refer to values in general terms, we mean both constant and register values. We distinguish between three main value categories: scalars (`s`), booleans (`b`), and pointers (`p`). Using minuscule notation, we refer to any value of a given kind. To specify that a constant or register is of a particular category, we use subscript notation. For example, $\mathtt{c}_\mathtt{s}$, $\mathtt{c}_\mathtt{b}$, and $\mathtt{c}_\mathtt{p}$ refer to constants of the scalar, boolean, and pointer categories, respectively.

Furthermore, we distinguish between scalar integer types (`i`) of various bitwidths, which are indicated by the suffix n. For example, the type `i32` represents a 32-bit integer type. Pointers are represented by an opaque `ptr` type. Similarly to LLVM IR, we treat booleans as `i1`. Conventionally we denote the set of all types and its elements as $\mathtt{ty} \in \mathbb{T}$.[2]

2: We exclude floating-point scalar values from the formal description to reduce verbosity, as they behave similarly to integer scalar values. Note that they are still included in the implementation.

To simplify matters, we assume that the program is well-typed. Therefore, we do not include types of operations that do not depend on them. Constants and registers inherently carry type constraints, meaning that scalar values can have either a specific integer type, while pointer values can only have an opaque `ptr` type. These constraints determine which operations are allowed on given values. The only operations that depend on types are memory-accessing operations and cast operations. For concreteness, we consider the following expressions grammar.

Definition 3.2.1 *Allowed program **expressions** are:*

- *Arithmetic expressions* $\mathtt{A} \in \mathbb{A}$*:*

  $$\mathtt{A} ::= \mathtt{s} \mid \mathtt{s} \circ \mathtt{s} \mid \mathbf{amb}(\mathtt{c}_\mathtt{s}^*) \mid \mathbf{trunc}\ \mathtt{s}\ \mathbf{to}\ \mathtt{ty} \mid \mathbf{zext}\ \mathtt{s}\ \mathbf{to}\ \mathtt{ty} \mid \mathbf{sext}\ \mathtt{s}\ \mathbf{to}\ \mathtt{ty}$$

  where $\mathtt{s} \in \{\mathtt{r}_\mathtt{s}, \mathtt{c}_\mathtt{s}\}$, $\circ \in \{+, -, *, /, \%, >>, <<\}$ *and* $\mathtt{ty} \in \mathbb{T}$ *is an integer type.*

- *Boolean expressions* $\mathtt{B} \in \mathbb{B}$*:*

  $$\mathtt{B} ::= \mathtt{b} \mid \mathbf{not}\ \mathtt{b} \mid \mathtt{b}\ \mathbf{and}\ \mathtt{b} \mid \mathtt{b}\ \mathbf{or}\ \mathtt{b} \mid \mathtt{v} \diamond \mathtt{v}$$

  where $\mathtt{b} \in \{\mathtt{r}_\mathtt{b}, \mathtt{c}_\mathtt{b}\}$, $\mathtt{v} \in \{\mathtt{r}_\mathtt{s}, \mathtt{c}_\mathtt{s}, \mathtt{r}_\mathtt{p}, \mathtt{c}_\mathtt{p}\}$ *and* $\diamond \in \{<, >, =, \neq\}$

- *Pointer expressions* $\mathtt{P} \in \mathbb{P}$*:*

  $$\mathtt{P} ::= \mathtt{p} \mid \mathtt{p} + \mathtt{i} \mid \mathtt{p} - \mathtt{p}$$

  where $\mathtt{p} \in \{\mathtt{r}_\mathtt{p}, \mathtt{c}_\mathtt{p}\}$ *and* `i` *is an integer scalar value*

The set of all expressions is $\mathbb{E} \stackrel{\text{def}}{=} \mathbb{A} \cup \mathbb{B} \cup \mathbb{P}$.

The expression **amb** stands for the ambiguous operator, which expresses program nondeterminism resulting from interactions with the environment or program inputs. The operator takes a variable number of values and splits the computation into multiple possible futures, producing a

single value for each future. We will use abbreviation **amb** without arguments to indicate arbitrary value, and $\textbf{amb}[x, y] \stackrel{\text{def}}{=} \textbf{amb}(\{c \mid x \leq c \leq y\})$ to denote a nondeterministic interval of values.

Note that we do not allow compound expressions with intermediate values. That is to simplify the definition of program semantics. From an interpretation perspective, all expressions are atomic. This mirrors the atomicity of expressions from LLVM IR.

We define three utility functions to query essential expression traits for later use in semantic definitions, which allow us to obtain the type and bitwidth of expression results.

Definition 3.2.2 (Expression traits) *To determine the* type *of expression* e *we will write* $ty(e)$*, where* $ty : \mathbb{E} \to \mathbb{T}$*, whereas to obtain its* bitwidth*, we will write* $bw(e)$*, where* $bw : \mathbb{E} \to \mathbb{N}$*.*

We also define sizeof $: \mathbb{T} \to \mathbb{N}$ *and sizeof* $: \mathbb{E} \to \mathbb{N}$ *to obtain the number of bytes required to represent a given type or value.*

In our language, we will use high-level control flow statements in contrast to LLVM IR, which uses block-based control flow. This simplifies the examples further and does not make a difference in the abstraction and program analysis. Otherwise, we keep the language constructs close to LLVM IR, e.g., memory management, function handling, and terminal statements. We define following program statements:

Definition 3.2.3 *Register assignments* $\texttt{stmt}_A \in \mathbb{S}_A$*:*

$$\texttt{stmt}_A ::= \texttt{r}_\texttt{s} \longleftarrow \texttt{A} \mid \texttt{r}_\texttt{b} \longleftarrow \texttt{B} \mid \texttt{r}_\texttt{p} \longleftarrow \texttt{P}$$

where $\texttt{A} \in \mathbb{A}$, $\texttt{B} \in \mathbb{B}$, $\texttt{P} \in \mathbb{P}$.

Definition 3.2.4 *Memory manipulations* $\texttt{stmt}_M \in \mathbb{S}_M$*:*

$$\begin{aligned}
\texttt{stmt}_M ::= {} & \textbf{store}\ \texttt{v} \longrightarrow \texttt{r}_\texttt{p} && \textit{(write to memory)} \\
\mid {} & \texttt{r} \longleftarrow \textbf{load}\ \texttt{r}_\texttt{p}\ \textbf{of}\ \texttt{ty} && \textit{(read from memory)} \\
\mid {} & \texttt{r}_\texttt{p} \longleftarrow \textbf{malloc}\ \texttt{s} && \textit{(heap allocation)} \\
\mid {} & \textbf{free}\ \texttt{r}_\texttt{p} && \textit{(heap deallocation)}
\end{aligned}$$

where $\texttt{ty} \in \mathbb{T}$.

Definition 3.2.5 *Program step statements* $\texttt{stmt}_{PS} \in \mathbb{S}_{PS}$*:*

$$\begin{aligned}
\texttt{stmt}_{PS} ::= {} & \textbf{skip} && \textit{(noop statement)} \\
\mid {} & \textbf{exit} && \textit{(exit statement)} \\
\mid {} & \textbf{error} && \textit{(error statement)}
\end{aligned}$$

Definition 3.2.6 *Control flow statements* $\texttt{stmt}_C \in \mathbb{S}_C$*:*

$$
\begin{aligned}
\texttt{stmt}_C ::= \ & \textbf{if}\ \texttt{B}\ \textbf{then}\ \texttt{stmt}^+\ \textbf{else}\ \texttt{stmt}^+ && \textit{(branch)}\\
| \ & \textbf{while}\ \texttt{B}\ \textbf{do}\ \texttt{stmt}^+ && \textit{(loop)}\\
| \ & \textbf{call}\ \texttt{symbol}\ (\texttt{v}^*) && \textit{(function call)}\\
| \ & \textbf{ret}\ \texttt{v} && \textit{(return)}
\end{aligned}
$$

where $\texttt{stmt} \in \mathbb{S}_A \cup \mathbb{S}_M \cup \mathbb{S}_{PS} \cup \mathbb{S}_C$.

Definition 3.2.7 *Program definitions* $\texttt{stmt}_P \in \mathbb{S}_P$*:*

$$
\begin{aligned}
\texttt{stmt}_P ::= \ & \textbf{fn}\ \ \texttt{symbol}\ (\texttt{r}^*)\ \texttt{stmt}^+ && \textit{(function definition)}\\
| \ & \texttt{stmt}_P^+ && \textit{(list of functions)}
\end{aligned}
$$

where `symbol` *is a name of a function.*

The set of all statements is $\mathbb{S} \stackrel{\text{def}}{=} \mathbb{S}_A \cup \mathbb{S}_M \cup \mathbb{S}_C \cup \mathbb{S}_P$. We will refer to *computational*, or single step statements as $\mathbb{S}_S \subseteq \mathbb{S}_A \cup \mathbb{S}_M \cup \mathbb{S}_{PS}$, that are only statements that do not involve control flow.

Definition 3.2.8 *A control flow abbreviation for a single path if statements:*

$$\textbf{if}\ \texttt{B}\ \textbf{then}\ \texttt{stmt} \stackrel{\text{def}}{=} \textbf{if}\ \texttt{B}\ \textbf{then}\ \texttt{stmt}\ \textbf{else skip}$$

For program analysis, we will leverage abbreviation statements to describe preconditions, postconditions, and general assertions. We can build these constructs from already defined conditional and terminal statements.

Definition 3.2.9 *We define* **assume** *and* **assert** *statements as:*

$$\textbf{assume}\ \texttt{B} \stackrel{\text{def}}{=} \textbf{if}\ !\texttt{B}\ \textbf{then exit}$$

$$\textbf{assert}\ \texttt{B} \stackrel{\text{def}}{=} \textbf{if}\ !\texttt{B}\ \textbf{then error}$$

```
int function(int y) {
  int x = input();
  while x < 10 {
    x++;
  }
  return 0;
}
```

```
define i32 @function(i32 %y) {
e:
  %1 = call i32 @input()
  %2 = icmp slt i32 %2, %y
  br i1 %2, label %t, label %x
t:
  %3 = phi i32 [%1, %e], [%4, %t]
  %4 = add nsw i32 %7, 1
  %5 = icmp slt i32 %4, %y
  br i1 %5, label %t, label %x
x:
  ret i32 0
}
```

```
fn function(y)
  x ← call input()
  while x < y do
    x ← x + 1
  ret 0
```

Figure 3.1: Comparison of C source, LLVM IR and our simplified language.

3.3 Concrete Semantics

The technique of abstract interpretation, originally presented by Cousot and Cousot [CC77a], is designed for big-step operational semantics and collecting semantics. In collecting semantics, the abstraction is computed using a fixed-point algorithm to assign possible valuations to each program location. However, this approach is not suitable for the compilation-based technique and model checking, which will utilize our abstraction.

[CC77a]: Cousot et al. (1977), "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints"

A more natural choice for these purposes is a small-step operational semantics that describes each atomic step of the executed/interpreted program. The small-step semantics is also traditionally used in model checking of safety and liveness properties [BK08]. The defining characteristic of small-step semantics is that it generates a new program state after interpreting each atomic step of the program. This allows us to observe the program after each step and, consequently, apply abstraction with a granularity of these atomic steps [Sch97].

[BK08]: Baier et al. (2008), *Principles of Model Checking (Representation and Mind Series)*

[Sch97]: Schmidt (1997), "Abstract interpretation in the operational semantics hierarchy"

Small-step semantics is well-suited for LLVM IR interpretation, as each instruction in LLVM IR is essentially atomic and does not allow unnamed intermediate values to arise, unlike in high-level programming languages. Each LLVM IR instruction is semantically atomic and mutates only a single value in the program state, and the same holds for computational statements in our simplified language. This lets us define each operation as a single step in the small-step semantics.

Our program's state (conventionally denoted σ when dealing with operational semantics) consists of register values and memory content. It is given as a tuple of partial maps which assign values from $\mathcal{C}$ to registers and memory locations. To represent memory faithfully, it is an untyped contiguous array of bytes similar to actual runtime memory representation. We will discuss memory more in later chapters.

All possible states or environments are described by set $\mathbb{Env}$. Formally, a state $\sigma \in \mathbb{Env}$ is a tuple $(\varepsilon_s, \varepsilon_p, \mu)$ where: $\varepsilon_s : \mathsf{Reg_s} \to \mathcal{C}_\mathsf{s}$ assigns scalar values to scalar registers, $\varepsilon_p : \mathsf{Reg_p} \to \mathcal{C}_\mathsf{p}$ assigns pointer values to pointer registers and $\mu : \mathsf{Id} \to \mathsf{Blk}$ maps base addresses to their respective memory blocks. We present a summary of the main semantic domains in Figure 3.2.

$$
\begin{array}{rcll}
c \in \mathcal{C}_\mathsf{s} & \equiv & \mathcal{BV} & \text{(scalar constants)} \\
p \in \mathcal{C}_\mathsf{p} & \equiv & \mathcal{BV} & \text{(pointers)} \\
a \in \mathsf{Id} & \equiv & \mathcal{BV} & \text{(base addresses)} \\
\langle a, o \rangle \in \mathsf{Cell} & \equiv & \mathsf{Id} \times \mathcal{C}_\mathsf{s} & \text{(memory cells)} \\
v \in \mathbb{V} & \equiv & \mathcal{C}_\mathsf{s} \to \mathcal{C} & \text{(memory block content)} \\
\langle s, v \rangle \in \mathsf{Blk} & \equiv & \mathcal{C}_\mathsf{s} \times \mathbb{V} & \text{(memory blocks)} \\
\varepsilon_s \in \mathbb{Env}_\mathsf{s} & \equiv & \mathsf{Reg_s} \to \mathcal{C}_\mathsf{s} & \text{(scalar registers environment)} \\
\varepsilon_p \in \mathbb{Env}_\mathsf{p} & \equiv & \mathsf{Reg_p} \to \mathcal{C}_\mathsf{p} & \text{(pointer registers environment)} \\
\mu \in \mathbb{Env}_\mu & \equiv & \mathsf{Id} \to \mathsf{Blk} & \text{(memory environment)} \\
\sigma \in \mathbb{Env} & \equiv & \mathbb{Env}_\mathsf{s} \times \mathbb{Env}_\mathsf{p} \times \mathbb{Env}_\mu & \text{(program states)}
\end{array}
$$

Figure 3.2: Concrete semantic domains.

We write $\sigma \xrightarrow{\texttt{stmt}} \Sigma$, where $\Sigma \subseteq \mathbb{E}_{\mathbb{V}}$, to denote step of a program, i.e., starting in state σ, execution of statement `stmt` leads to states Σ. To obtain a value stored in the register, we access corresponding register maps: $\varepsilon_s(s)$ for scalar registers and $\varepsilon_p(p)$ for pointer registers. Assignment of a value c to register r in the respective register map, is denoted as $\varepsilon_s[r \mapsto c]$ for scalar assignment or $\varepsilon_p[r \mapsto c]$ for pointer assignment. Likewise, $\mu(a)$ retrieves block from memory at base address a.

We define the expression valuation semantic function as $[\![\texttt{E}]\!]_\sigma : \mathbb{E} \times \mathbb{E}_{\mathbb{V}} \to \mathcal{C}$ to give concrete meaning to expression `E` in state σ. To model program execution faithfully, we consider concrete values to be bit-vector values ($\mathcal{BV}$), i.e., we operate on bitwidth-bounded values with signed overflow semantics, as in other low-level languages. We define expression values at state $\sigma = (\varepsilon_s, \varepsilon_p, \mu)$ as follows.

Definition 3.3.1 *The valuation semantic of arithmetic expressions is:*

$$
\begin{aligned}
[\![\texttt{c}_\texttt{s}]\!]_\sigma &\stackrel{\text{def}}{=} \{c\}, \textit{ where } c \in \mathcal{C} \textit{ is representation of } \texttt{c}_\texttt{s} \\
[\![\texttt{r}_\texttt{s}]\!]_\sigma &\stackrel{\text{def}}{=} \{\varepsilon_s(\texttt{r}_\texttt{s})\} \\
[\![\texttt{s}_1 \circ \texttt{s}_2]\!]_\sigma &\stackrel{\text{def}}{=} \{s_1 \circ s_2 \mid s_1 \in [\![\texttt{s}_1]\!]_\sigma \wedge s_2 \in [\![\texttt{s}_2]\!]_\sigma\} \\
[\![\textbf{zext } \texttt{s } \textbf{to } \texttt{ty}]\!]_\sigma &\stackrel{\text{def}}{=} \{ext^{[bw(\texttt{ty})]U}(s) \mid s \in [\![\texttt{s}]\!]_\sigma\} \\
[\![\textbf{sext } \texttt{s } \textbf{to } \texttt{ty}]\!]_\sigma &\stackrel{\text{def}}{=} \{ext^{[bw(\texttt{ty})]S}(s) \mid s \in [\![\texttt{s}]\!]_\sigma\} \\
[\![\textbf{trunc } \texttt{s } \textbf{to } \texttt{ty}]\!]_\sigma &\stackrel{\text{def}}{=} \{(s)^{[bw(\texttt{ty})]}(s) \mid s \in [\![\texttt{s}]\!]_\sigma\} \\
[\![\textbf{amb}(\texttt{c}_1, \ldots, \texttt{c}_n)]\!]_\sigma &\stackrel{\text{def}}{=} \bigcup_{i=1}^{n} [\![\texttt{c}_i]\!]
\end{aligned}
$$

where $\circ \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{>>}, \texttt{<<}\}$ *are bit-vector operations. Our semantics align with the common practice of bit-vector theory, where extension expression extend the bit-vector to the bitwidth of type denoted as* $bw(\texttt{ty})$*. We support both signed (S) and unsigned (U) variants of extension. Truncation is performed by keeping only the specified number of bits up to the given bitwidth, as denoted in its superscript.*

These expressions read and modify the values of scalar registers, as represented by the map ε_s. Memory and pointer registers are not involved in these expressions. For simplicity, errors that may occur from improper usage of scalars, such as division by zero or invalid bit shifts, are not addressed here, with the assumption that the underlying analysis tool will identify and report them.

Definition 3.3.2 *The valuation semantic of pointer expressions is:*

$$
\begin{aligned}
[\![\texttt{c}_\texttt{p}]\!]_\sigma &\stackrel{\text{def}}{=} \{p\}, \textit{ where } p \in \mathcal{C}_\texttt{p} \textit{ is representation of } \texttt{c}_\texttt{p} \\
[\![\texttt{r}_\texttt{p}]\!]_\sigma &\stackrel{\text{def}}{=} \{\varepsilon_p(\texttt{r}_\texttt{p})\} \\
[\![\texttt{p}_1 \texttt{+ i}_2]\!]_\sigma &\stackrel{\text{def}}{=} \{p + i \mid p \in [\![\texttt{p}_1]\!]_\sigma \wedge i \in [\![\texttt{i}_2]\!]_\sigma\} \\
[\![\texttt{p}_1 \texttt{- p}_2]\!]_\sigma &\stackrel{\text{def}}{=} \{p_1 - p_2 \mid p_1 \in [\![\texttt{p}_1]\!]_\sigma \wedge p_2 \in [\![\texttt{p}_2]\!]_\sigma\}
\end{aligned}
$$

Unlike arithmetic expressions, pointer expressions do not involve multiplication and division. However, in practice, it is possible to convert

pointer values to integers and perform these operations on them, even though it may not have a valid semantic meaning.

Otherwise, pointer expressions accept a pointer operand and produce a pointer result, possibly also taking additional scalar operands. Only one operation fits this description: the addition of a scalar to a pointer, with the goal to adjust its offset within a memory block. On the contrary the difference of two pointers is a scalar value.

Definition 3.3.3 *The valuation semantic of boolean expressions is:*

$$
\begin{aligned}
[\![\mathsf{T}]\!]_\sigma &\stackrel{\text{def}}{=} \{\mathsf{T}\} \\
[\![\mathsf{F}]\!]_\sigma &\stackrel{\text{def}}{=} \{\mathsf{F}\} \\
[\![\textbf{not } \mathsf{b}]\!]_\sigma &\stackrel{\text{def}}{=} \{\neg b \mid b \in [\![\mathsf{b}]\!]_\sigma\} \\
[\![\mathsf{b}_1 \textbf{ and } \mathsf{b}_2]\!]_\sigma &\stackrel{\text{def}}{=} \{b_1 \wedge b_2 \mid b_1 \in [\![\mathsf{b}_1]\!]_\sigma \wedge b_2 \in [\![\mathsf{b}_2]\!]_\sigma\} \\
[\![\mathsf{b}_1 \textbf{ or } \mathsf{b}_2]\!]_\sigma &\stackrel{\text{def}}{=} \{b_1 \vee b_2 \mid b_1 \in [\![\mathsf{b}_1]\!]_\sigma \wedge b_2 \in [\![\mathsf{b}_2]\!]_\sigma\} \\
[\![\mathsf{v}_1 \diamond \mathsf{v}_2]\!]_\sigma &\stackrel{\text{def}}{=} \{v_1 \diamond v_2 \mid v_1 \in [\![\mathsf{v}_1]\!]_\sigma \wedge v_2 \in [\![\mathsf{v}_2]\!]_\sigma\}
\end{aligned}
$$

where $\diamond \in \{<, >, =, \neq\}$. *Boolean values* `T` ***(true)*** *and* `F` ***(false)*** *are equivalent to single-bit bit-vector zero and one, as in other low-level languages.*

Using the valuation semantics, we can now define the step semantics of statements. In the step semantics, we perform atomic steps to update respective registers or memory maps.

Definition 3.3.4 *The most basic statement is* `skip` *which leaves the program state unchanged:*

$$(\varepsilon_s, \varepsilon_p, \mu) \xrightarrow{\textbf{skip}} (\varepsilon_s, \varepsilon_p, \mu)$$

Definition 3.3.5 *In the assignment statements, we update respective register maps to hold the valuation* v *of the assigned expression::*

$$
\begin{aligned}
(\varepsilon_s, \varepsilon_p, \mu) &\xrightarrow{\mathsf{r_s} \longleftarrow \mathsf{A}} \{(\varepsilon_s[\mathsf{r_s} \mapsto v], \varepsilon_p, \mu) \mid v \in [\![\mathsf{A}]\!]_\sigma\} \\
(\varepsilon_s, \varepsilon_p, \mu) &\xrightarrow{\mathsf{r_b} \longleftarrow \mathsf{B}} \{(\varepsilon_s[\mathsf{r_b} \mapsto v], \varepsilon_p, \mu) \mid v \in [\![\mathsf{B}]\!]_\sigma\} \\
(\varepsilon_s, \varepsilon_p, \mu) &\xrightarrow{\mathsf{r_p} \longleftarrow \mathsf{P}} \{(\varepsilon_s, \varepsilon_p[\mathsf{r_b} \mapsto v], \mu) \mid v \in [\![\mathsf{P}]\!]_\sigma\}
\end{aligned}
$$

It is important to note that in this context, scalar and boolean registers are treated similarly, as boolean values are a subset of scalar values. In the case that valuation results in multiple values, for instance, in the case of `amb` operation, the assignment produces a set of states (see Figure 3.3).

Addressable memory is represented in the program's state by the map μ. Memory allocation operations (`malloc` and `free`) manage memory allocation by creating and destroying memory objects, respectively. These operations add or remove memory blocks from the dynamic memory, similar to how `C` semantic manages memory.

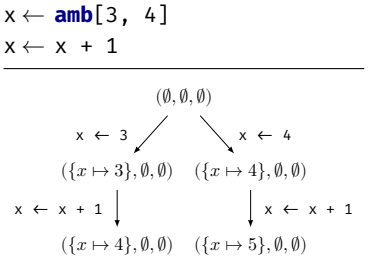


Figure 3.3: Program execution example with `amb` operator.

Definition 3.3.6 *(Heap allocation) we define dynamic memory allocation as:*

$$\sigma \xrightarrow{\mathtt{r_p} \leftarrow \textbf{malloc}\ \mathtt{s}} \{(\varepsilon_s, \varepsilon_p[\mathtt{r_p} \mapsto \langle q, 0\rangle], \mu[q \mapsto \mathit{block}(s)])\}$$

where $q \in \mathsf{Id}$ *is the base address of the allocated block,* $s \in [\![\mathtt{s}]\!]_\sigma$ *and*

$$\mathit{block}(s) \stackrel{\text{def}}{=} \langle s, \{o \mapsto \top \mid 0 \leq o < \mathit{max}(1, s)\}\rangle$$

When a memory block is allocated, a pointer to the start of the block is returned, which points to an unused *memory location. We say memory block with base address* q *is* unused *if it is not allocated in the current state, i.e.,* $\mu(q) = \bot$.

The content of memory block is set to arbitrary value $\top$ for the whole newly allocated block, to reflect the fact that it is not yet initialized and may contain arbitrary (garbage) values (cf. Figure 3.4).

Memory allocation at state σ returns an arbitrary unused block q. To ensures that `malloc(0)` is modeled correctly, in the sense that the resulting object is assigned a unique address as required by the relevant standards, we say that block size is $\mathit{max}(1, s)$ where s is allocation size.

Technically, the arbitrariness in the choice of the location q means that there is not a single result state σ' – instead, many are possible. In fact, too many to enumerate, and for this reason, the semantics of `malloc` are often under-approximated: for example, by only considering a single choice for the value of q. For now, we will assume that the location choice is fixed. However, we will address this issue later in Chapter 6, where we will expound on heap layout abstractions.

```
p ← malloc 4   ⟅p ↦ ⟨a, 0⟩⟆
p ← p + 2      ⟅p ↦ ⟨a, 2⟩⟆
store 7 → p
```

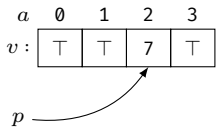


Figure 3.4: Memory allocation example. The picture illustrates an allocated block at address a. The program state holds in memory $\mu[a \mapsto \langle 4, v\rangle]$. The content v is shown after the store at offset 2.

Definition 3.3.7 *(Heap deallocation) To release allocated memory, a program may use* `free` *operation:*

$$(\varepsilon_s, \varepsilon_p, \mu) \xrightarrow{\textbf{free}\ \mathtt{r_p}} \{(\varepsilon_s, \varepsilon_p, \mu[a \mapsto \bot]) \mid \langle a, o\rangle \in [\![\mathtt{r_p}]\!]_\sigma\}$$

Deallocation resets the memory map μ for the object described by the pointer stored in register $\mathtt{r_p}$. It is left up to the implementation (underlying interpreter) how it keeps the information about allocation sizes.[3]

In the following, we give semantics to memory manipulation operations. It is worth noting that these operations may manipulate multiple bytes at once and change or merge previously written data.

```
x ← malloc sizeof(i32)
y ← malloc sizeof(ptr)
z ← malloc 3 * sizeof(ptr)
store 4 → x
store x → y
store x → z
z ← z + 1
store y → z
```

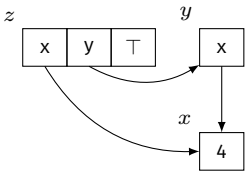


Figure 3.5: Example of the memory graph generated by the program described above. It should be noted that here we abstract from the byte representation of pointers.

Definition 3.3.8 *(Memory update) To update memory content we define* `store` *operation, which transfers values from registers into memory blocks:*

$$\sigma \xrightarrow{\textbf{store}\ \mathtt{v} \longrightarrow \mathtt{r_p}} \{(\varepsilon_s, \varepsilon_p, \mu[a \mapsto \mathit{store}_\sigma(a, o, v)])\}$$

where

$$\langle a, o\rangle \in [\![\mathtt{r_p}]\!]_\sigma \wedge v \in [\![\mathtt{v}]\!]_\sigma$$

3: In DIVINE, the tool we use to demonstrate the compilation-based abstraction, the allocator keeps track of all memory objects and their respective sizes in a memory graph.

Moreover the memory block with base address a is defined ($\mu(a) \neq \bot$).

The store performs an update of bytes in the pointed memory block $\mu(a) = \langle s, v\rangle$ starting from offset o:

$$\mathit{store}_\sigma(a, o, x) \stackrel{\text{def}}{=} \bot \quad \text{if } o + \text{sizeof}(x) > s$$
$$\mathit{store}_\sigma(a, o, x) \stackrel{\text{def}}{=} \langle s, v[o + i \mapsto x^{[i]} \mid 0 \leq i < \text{sizeof}(x)]\rangle \text{ otherwise}$$

The store updates content in a valid memory range, i.e., it updates map of values v for a particular memory block. It performs update for each byte of the written value x, we denote ith byte as $x^{[i]}$. Update returns invalid block $\bot$ in case of write out of bounds.

Definition 3.3.9 *(Memory access) To read from memory we define* `load` *operation, which transfer values from memory location $[\![\mathtt{r}_a]\!]_\sigma$ into registers. Let $n = \text{sizeof}(\mathtt{ty})$ in:*

$$\sigma \xrightarrow{\mathtt{r_s} \longleftarrow \textbf{load}\ \mathtt{r}_a\ \textbf{of}\ \mathtt{ty}} \{(\varepsilon_s[\mathtt{r_s} \mapsto \mathit{access}_\sigma(p, n)], \varepsilon_p, \mu) \mid p \in [\![\mathtt{r}_a]\!]_\sigma\}$$
$$\sigma \xrightarrow{\mathtt{r_p} \longleftarrow \textbf{load}\ \mathtt{r}_a\ \textbf{of}\ \mathtt{ty}} \{(\varepsilon_s, \varepsilon_p[\mathtt{r_s} \mapsto \mathit{access}_\sigma(p, n)], \mu) \mid p \in [\![\mathtt{r}_a]\!]_\sigma\}$$

where memory access reconstructs single multibyte value of size n by concatenating bytes from memory block a from offset o:

$$\mathit{access}_\sigma(\langle a, o\rangle, n) \stackrel{\text{def}}{=} \bot \quad \text{if } o + n > s$$
$$\mathit{access}_\sigma(\langle a, o\rangle, n) \stackrel{\text{def}}{=} \mathit{concat}(\{v(o + i) \mid 0 \leq i < n\}) \text{ otherwise}$$

If the access is out of bounds the value is invalid $\bot$.

```
p ← malloc sizeof(i16)
y ← 7
store y → p
free p
```

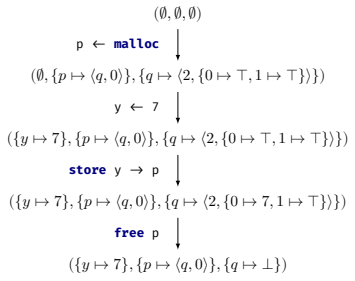


Figure 3.6: Program execution examples.

3.4 Control Flow

We will model control flow using *control flow graphs* (CFG). In the CFG, edges are labeled by actions, which can have two flavors. The first is a computational edge labeled by step statement $s \in \mathbb{S}_S$, and the second is a guard expression that constrains the control flow $c \in \mathbb{B}$. Since the statements can perform only a single atomic change on a program state, we won't have intermediate changes arising on edge execution.

Definition 3.4.1 *A control flow graph $\mathcal{G}$ is defined by following:*

- *Q a finite set of states,*
- *$E \subseteq Q \times (\mathbb{S}_S \cup \mathbb{B}) \times Q$ a finite set of labeled edges,*
- *$q_{start}, q_{exit}, q_{err} \in Q$ initial, final and error state.*

The control flow graph is generated from a given program, with each elementary computation step defining a new edge.[4] In addition to computation steps, our language includes special step statements (`exit` and

4: Note that this differs from LLVM IR, where the control flow graph is defined by basic blocks consisting of elementary operations.

`error`) that always lead to a terminal (q_{exit}) or error state(q_{err}), respectively. A more complex control flow is created from control flow statements such as conditionals, loops, and function calls. It is important to note that these statements create guard statements that limit the control flow, as shown in Figure 3.7.

To represent function calls, we give each function its own CFG (cf. Figure 3.8). The edge with the `call` statement represents the entire evaluation of particular CFG.[5] In the function, `ret` statements always lead to the function exit state.

5: This will eventually let us abstract entire function calls in Chapter 5, or perform context insensitive analysis.

The control flow graph essentially describes what steps are allowed at a specific location, however, it does not describe the actual computation. To account for the computation, we must also consider the program's states. For that, we leverage the step semantics given in the previous section, which describes how states change along the path and whether they satisfy guard conditions. We can now define an execution path – single *program run*. In general, a program run will consist of a sequence of program configurations:

Definition 3.4.2 *A* program configuration *is a pair $\langle q, \sigma \rangle$, where $q \in Q$ is the program location (control flow node) and $\sigma \in \mathbb{E}_{\mathbb{V}}$ is the memory environment.*

Whenever (q_1, lab, q_2) is a control flow graph edge, the *execution step* $\langle q_1, \sigma_1 \rangle \xrightarrow{lab} \langle q_2, \sigma_2 \rangle$ is valid iff one of the following holds:

- $lab \in \mathbb{S}_S$ is a *computation step* and the step is defined, i.e., $\sigma_1 \xrightarrow{lab} \sigma_2$,
- $lab \in \mathbb{B}$ is a *guard step* and the guard holds, i.e., $\mathtt{T} \in [\![lab]\!]_{\sigma_1}$ and environment does not change: $\sigma_1 = \sigma_2$.

A **path** π is a sequence of execution steps, such that all steps form a continuous path in the transition system given by control flow graph $\mathcal{G}$. If the starting point of path π is the initial program location, it is referred to as a **program path**. Moreover, a path π is considered **feasible** if and only if each step in the sequence is a valid execution step in the transition system.

Definition 3.4.3 *A* verification task *for control flow graph $\mathcal{G} = \langle Q, E \rangle$ is to show that error location $q_{err} \in Q$ is unreachable in $\mathcal{G}$ form $q_{start} \in Q$, or to find feasible error path.*

For instance, a feasible error path in Figure 3.8, visualized in red color, is when `y` $\neq$ `2` and that is for program run when `amb` returns 3:

$$\begin{aligned}
&\langle q_1, \{x = \bot, y = \bot\}\rangle && \to \langle q_2, \{x = 3, y = \bot\}\rangle \\
\to\ &\langle f_1, \{a = 1, b = 3, c = \bot, d = \bot\}\rangle && \to \langle f_2, \{a = 1, b = 3, c = 2, d = \bot\}\rangle \\
\to\ &\langle f_3, \{a = 1, b = 3, c = 2, d = \bot\}\rangle && \to \langle f_{exit}, \{a = 1, b = 3, c = 2, d = \bot\}\rangle \\
\to\ &\langle q_3, \{x = 3, y = 1\}\rangle && \to \langle q_5, \{x = 3, y = 1\}\rangle \\
\to\ &\langle q_{err}, \{x = 3, y = 1\}\rangle
\end{aligned}$$

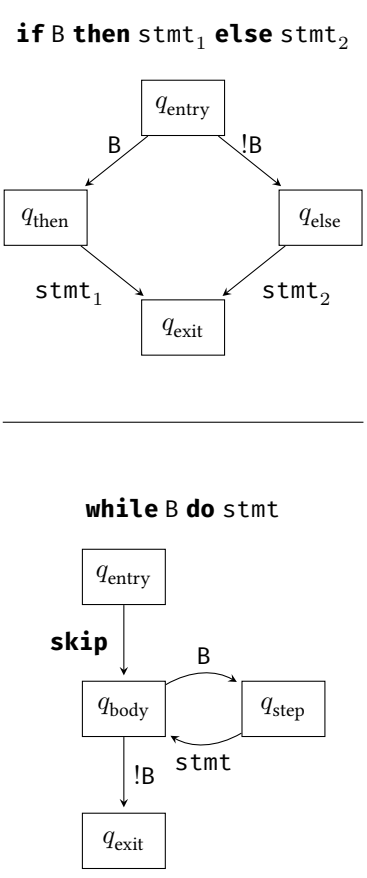


Figure 3.7: Two main building blocks of control flow graphs are conditions and loops.

However, in general, it is undecidable to solve verification tasks on concrete programs. For that, we utilize abstraction to simplify state space

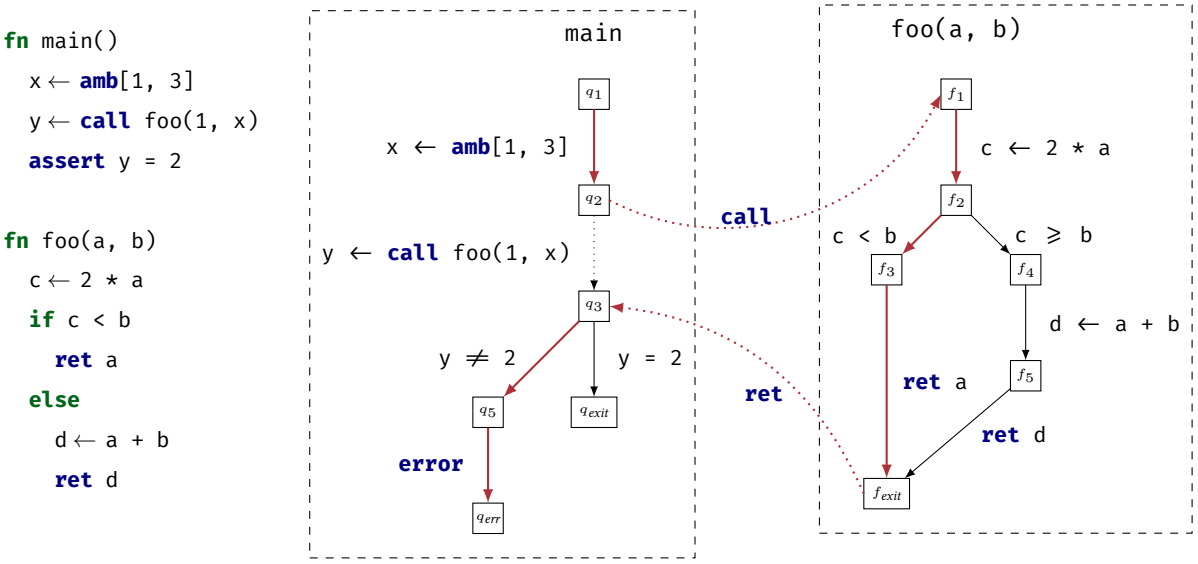


Figure 3.8: Example of an interprocedural control flow graph. Note the difference to program state space, where we distinguish various configurations. Whereas in CFG, statements like `amb` have only a single successor. A possible error path is shown in **red**.

exploration. In abstract execution, we compute with an abstractly represented set of program states instead of computing with a single program environment.

3.5 Abstract Domains

In the following we give a brief overview of abstract domains presented throughout the thesis. We are primarily interested in *property domains*, which reason about program state properties. Depending on the part of the state, we distinguish multiple categories of domains.

The basic building block of property domains is *basis* that expresses how to abstract single program entity (a value or an object) and operations on it. Thanks to the Galois connection the rest of interpretation can be derived automatically as a Cartesian product for all variables [Min04].

[Min04]: Miné (2004), "Weakly relational numerical abstract domains"

Given the state of a program, each of the program variables and registers requires various levels of abstraction, as well as different abstract domains. Depending on what values we abstract, we distinguish abstraction of numerical values, memory content, and also program pointers. Consequently, the entire abstraction of program state is formed by a Cartesian product of all used abstractions. Moreover, some parts of the state might be omitted from the abstraction entirely and represented in the concrete domain.

Scalar domains are used to reason about properties of ground program values like integers and floating point values. Traditional scalar domains are also known in the literature as *numerical abstract domains* defined by Cousot in [CC77a].

We will explore the properties of simple domains and their applicability to program analysis in the context of compilation-based abstraction. In LART, we provide a *sign*, *interval*, or *congruence* domain. These can be used in

various scenarios like bounded loop abstraction using congruence domain. More specific scalar domains that we utilize in the program analysis are the *constant* domain used to interact with the *concrete* domain, the *tristate* domain to express the ambiguity of relational operations, and the single value (*unit*) domain to abstract values entirely away. Lastly, we consider symbolic representation a special case of abstraction. For this purpose, we design a *term* domain that allows reasoning about program variables using various SMT theories. The scalar analysis is further described in Chapter 4.

Aggregate domains, described in Chapter 5, reason about more complex objects and their transformers. It is often expedient to design domain-specific abstractions for high-level program structures. Examples are arrays or strings in the C language. Using the abstraction over the whole aggregates, we can perform an execution with symbolically large objects. This would not be possible if we did the abstraction in an element-wise manner. An advantage of this approach is the ability to specify behavior on symbolically sized objects. In contrast, element-wise abstraction requires consideration of each individual element separately.

Aggregate domains distinguish from scalar domains by the transformers they abstract. The operations that interact with aggregate domains are memory access operations. Moreover, we are interested in abstracting functions over aggregates, like string or array-manipulating standard library functions. Domains we describe in this area are usually closely related to the language-defined aggregates. We implement squashing array domain and segmented string domain. Furthermore, domains used to abstract system entities like files also fall into this category.

Memory domains, or heap-abstracting domains, can be divided into two categories. The first abstracts memory shape, while the second abstracts pointers (possible memory locations). We present a pointer domain for relational analysis later in Chapter 6.

Lastly, we will leverage multiple **generic domains** — parametric functor domains. These are not domains per se, but they extend the capabilities of domains they are parametrized with. Examples of parametric domains are:

- a *product domain*[6] that allows representing a single value in multiple domains at once, possibly supporting each other precision
- a *relational domain* that augments non-relational domain with computation dependencies used for value refinement,
- a *set domain* that allows computing with sets of abstract values, again increasing the abstraction granularity and precision.

6: It is important to distinguish between the concepts of product domain and Cartesian product of domains. The former represents a singular value abstraction across multiple domains, whereas the latter is utilized to refer to the abstraction of multiple values (often an abstract state).

3.6 Program Analysis Methods

As previously mentioned in Section 2.2, there are three general approaches to tackling the undecidability of program analysis. The approach that is of interest to us is abstraction, as it enables us to reason about open programs (program inputs) and potentially describe infinitely large state spaces in a finite manner.

There are two approaches to program abstraction: static and dynamic. In the static approach, program invariants are inferred purely from control flow graph inspection. On the contrary, the dynamic approach involves executing program to prove respectively disprove program properties by examining the actual program state space.

In static analysis, it is common to use *collecting semantics* [AMS20], which assigns a set of possible values of variables to program locations (control flow graph nodes). However, despite the finite size of control flow graphs, the collected sets can easily become uncomputable with concrete values. Therefore, we employ abstraction to describe sets assigned to variables in a more concise manner. Abstracted collected sets are usually computed using *abstract interpretation*, and despite the term *interpretation* in its name, it is considered a static approach as it essentially only interprets CFG statements and applies effects on collected sets.

[AMS20]: Amato et al. (2020), "On collecting semantics for program analysis"

On the other hand, in dynamic execution, *step semantics* is a common choice, as it corresponds to step-by-step program execution. To verify the program dynamically, we need to explore all its possible paths, which can likewise become uncomputable, particularly if the program has infinite paths. To address this, we once again utilize abstraction to describe multiple program states simultaneously and effectively analyze multiple paths simultaneously. Two techniques of interest that fall into this category are abstract/symbolic execution and software model checking. Both of these techniques explore the state space of the program. However, while execution effectively only performs reachability analysis, model checking can verify more complex properties about the program state space, such as temporal properties.

As our subsequent focus is on compilation-based abstraction, which naturally involves program execution, we are primarily interested in dynamic techniques. Nevertheless, we will draw inspiration from abstract interpretation, and therefore, we will provide a more in-depth introduction to all above-mentioned techniques in the following sections.

3.6.1 Abstract Interpretation

The original idea of abstract interpretation dates back to the late 70s, first summarized by Patrick and Radhia Cousot [CC77a]. They describe *abstract interpretation* as a theory of abstraction and constructive approximation of the mathematical structures used in the formal description of programming languages or verification of undecidable program properties [Cou12].

[CC77a]: Cousot et al. (1977), "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints"

[Cou12]: Cousot (2012), "Formal Verification by Abstract Interpretation"

The goal of abstract interpretation is to assign a context to each program location (cf. Example 3.6.1), which is an abstractly represented set of possible values in a given domain (also known as collecting semantics). This is typically achieved by initially assigning all variables at each location with some initial information (usually $\bot$, which denotes an abstract empty set, since we do not know anything about the variable properties before the interpretation begins). Subsequently, we interpret each control flow edge, applying its effect to the collected sets, which is repeated until the fixed point is reached, meaning that no edge changes any collected set. This approach is also known as fixpoint iteration.

Example 3.6.1 In the following, concrete collecting semantics is annotated between $\wr \ldots \wr$. Each location can hold a set of contexts, where each context is described between angle brackets:

```
x ← 0  ≀⟨x = 0⟩≀
while x < 3  ≀⟨x = 0⟩, ⟨x = 2⟩, ⟨x = 4⟩≀ do  ≀⟨x = 0⟩, ⟨x = 2⟩≀
    x ← x + 2  ≀⟨x = 2⟩, ⟨x = 4⟩≀
≀⟨x = 4⟩≀
```

In the following snippet of the same program, we illustrate the concept of abstract collecting semantics using an interval domain. The interval domain abstracts collected sets by their minimum and maximum values. In the subsequent Chapter 4 on scalar domains, we provide a formal definition of the interval domain.

```
x ← 0  ≀⟨x = [0, 0]⟩≀
while x < 3  ≀⟨x = [0, 4]⟩≀ do  ≀⟨x = [0, 2]⟩≀
    x ← x + 2  ≀⟨x = [2, 4]⟩≀
≀⟨x = [4, 4]⟩≀
```

In multi-variable analysis of concrete collecting semantics each context captures all variables:

```
x ← 0  ≀⟨x = 0, y = ⊥⟩≀
y ← 1  ≀⟨x = 0, y = 1⟩≀
while x < y  ≀⟨x = 0, y = 1⟩, ⟨x = 2, y = 2⟩≀ do  ≀⟨x = 0, y = 1⟩≀
    x ← x + 2  ≀⟨x = 2, y = 1⟩≀
    y ← y + 1  ≀⟨x = 2, y = 2⟩≀
≀⟨x = 2, y = 2⟩≀
```

The effects of control flow statements are typically defined using a system of fixpoint equations (cf. Example 3.6.2). It has been proven [CC77a] that the result of fixpoint iteration corresponds to the solution of this system of equations. As a result, each abstract context represents multiple concrete contexts – potential properties or invariants about states of variables at a given program location.

Example 3.6.2 Consider a simple program with labeled locations on the left. The corresponding equation system for interval domain $\vec{\chi} = F(\vec{\chi})$ is depicted on the right:

```
x ← 0  l1
while x < 3 do
  l2  x ← x + 2  l3
  l4
```

$$
\begin{aligned}
\chi_1 &= F_1() = [0, 0] \\
\chi_2 &= F_2(\chi_1, \chi_3) = [-\infty, 2] \sqcap (\chi_1 \sqcup \chi_3) \\
\chi_3 &= F_3(\chi_2) = \chi_2 + [2, 2] \\
\chi_4 &= F_2(\chi_1, \chi_3) = [3, \infty] \sqcap (\chi_1 \sqcup \chi_3)
\end{aligned}
$$

The advantage of abstract interpretation is that we do not have to execute the program. Instead, we can use a fixpoint iteration algorithm to solve equations. This involves starting with the initial solution $\vec{\chi} = \overrightarrow{\bot}$ and repeatedly applying the transfer function F until a fixpoint is reached.

We do not necessarily follow the order of execution but only update the values at locations that have changed. There are many types of iteration strategies with different accelerations and convergence of fixpoint iteration. For instance, it is common to iterate in the weak topological order of program control-flow graph to deal with loops more efficiently [Bra+14]. Likewise, this approach allows us to easily speed up the computation by overapproximating the solution of equations, reaching the fixpoint more quickly. Techniques such as widening or domain refinement can be employed for this purpose, and we will delve into their details in the subsequent chapters.

[Bra+14]: Brat et al. (2014), "IKOS: A Framework for Static Analysis Based on Abstract Interpretation"

3.6.2 Abstract Execution

We can consider abstract execution as a generalized version of symbolic execution, but using an arbitrary domain. Recall the Definition 2.3.6 of an abstract domain $\mathcal{A}$. To perform abstract execution, we must extend the program state definition to allow values from a given abstract domain $\mathcal{A}$.

We denote abstract environment as $\widehat{\mathbb{E}\mathbb{v}}$, an element of abstract environment $\hat{\sigma} \in \widehat{\mathbb{E}\mathbb{v}}$ effectively represents a set of concrete environments $\hat{\sigma} \subseteq \mathbb{E}\mathbb{v}$. Then, the abstract transition relation is of type $\widehat{\rightarrow} : \widehat{\mathbb{E}\mathbb{v}} \times \mathbb{S}_{\mathsf{S}} \times \wp(\widehat{\mathbb{E}\mathbb{v}})$.

During the execution, we want to take advantage of the fact that not all values need to be abstractly represented. To achieve this, we define an abstract state $\hat{\sigma} = (\hat{\varepsilon}_s, \hat{\varepsilon}_p, \hat{\mu})$, which allows for both concrete and abstract values to be carried simultaneously. For example, in the case of scalar value abstraction, we extend the scalar register map $\hat{\varepsilon}_s : \mathsf{Reg}_{\mathsf{s}} \to \mathcal{C}_{\mathsf{s}} \cup \mathcal{A}$ and the memory content map $\hat{v} \in \hat{\mathbb{V}} \equiv \mathcal{C}_{\mathsf{s}} \to \mathcal{C} \cup \mathcal{A}$ to enable the storage of abstract values in both registers and memory blocks.

Example 3.6.3 Imagine a program state with abstract interval values in register r_1 and memory block a_1:

$$\hat{\sigma} = (\{r_1 \mapsto [1,3]\}, \emptyset, \{a_1 \mapsto \langle 1, \{0 \mapsto [0,1]\}\rangle\})$$

Such state represents a set of six concrete states:

$$(\{r_1 \mapsto 1\}, \emptyset, \{a_1 \mapsto \langle 1, \{0 \mapsto 0\}\rangle\}) \quad (\{r_1 \mapsto 1\}, \emptyset, \{a_1 \mapsto \langle 1, \{0 \mapsto 1\}\rangle\})$$

$$(\{r_1 \mapsto 2\}, \emptyset, \{a_1 \mapsto \langle 1, \{0 \mapsto 0\}\rangle\}) \quad (\{r_1 \mapsto 2\}, \emptyset, \{a_1 \mapsto \langle 1, \{0 \mapsto 1\}\rangle\})$$

$$(\{r_1 \mapsto 3\}, \emptyset, \{a_1 \mapsto \langle 1, \{0 \mapsto 0\}\rangle\}) \quad (\{r_1 \mapsto 3\}, \emptyset, \{a_1 \mapsto \langle 1, \{0 \mapsto 1\}\rangle\})$$

Namely, it creates a combination of states for all abstracted register values $1, 2, 3$ and single-byte memory content, which is either 0 or 1.

The distinctive feature of abstract execution is its branching control flow. For example, consider the condition `x < 10`. If the abstract value of `x` is an interval such as $x = [0, 20]$, it is both smaller and greater than `10`. In this case, the condition results in *maybe* value, and we need to explore both paths of the branch in order to have a sound analysis. This occurs when the concretization of the abstract branching condition *cond* is ambiguous: $\gamma(\mathit{cond}) = \{\mathsf{T}, \mathsf{F}\}$.

3.6.3 Abstract Model Checking

One of the applications for compilation-based abstraction presented in this thesis is an extension of explicit state model checking to abstract model checking. In model checking, the generic algorithm enumerates a program's state space and verifies the desired property. The traditional approach uses a Kripke structure [Kri07], essentially a labeled transition system, with states corresponding to program configurations and edges to transitions between configurations. Additionally, states or edges are labeled with atomic properties, distinguishing between state-based and transition-based model checking [WKP09]. This technique is also known as automata-based model checking [VW86].

[Kri07]: Kripke (2007), "Semantical considerations of the modal logic"

[WKP09]: Wehrle et al. (2009), "Transition-based directed model checking"

[VW86]: Vardi et al. (1986), "An automata-theoretic approach to automatic program verification"

For now, we will only consider a simple case of reachability, where the goal of model checking is to disprove the reachability of the error state/edge. That is $\mathtt{G}\neg \mathit{error}$ in LTL specification. Given the concrete small-step semantics and transition relation defined in Section 3.3, it is straightforward to induce the Kripke structure since transitions and states remain the same. Only edges with `error` statements are labeled with error property.

Model checking techniques employ various abstraction techniques to tackle large and possibly infinite state spaces. The idea is similar to value-based abstraction, where we perform the analysis over a set of values at once. In abstraction-based model checking, we try to limit state space size by grouping states and analyzing the transition system over abstract states (set of states). We recognize four primary flavors of model checking depending on the implemented abstraction:

1. *Explicit-state model checking* does not employ data abstraction and keeps all states explicitly. Explicit-state tools traditionally use heavy transition relation reductions to emit only interesting transitions for examined properties, e.g., partial-order reduction [FG05].

[FG05]: Flanagan et al. (2005), "Dynamic Partial-Order Reduction for Model Checking Software"

2. *Symbolic model checking*, instead of exploring states one at a time, explores multiple states described symbolically at one step. It leverages implicit state space description, usually using BDDs [Bur+90].

[Bur+90]: Burch et al. (1990), "Symbolic model checking: 10^{20} states and beyond"

3. *Bounded model checking* unrolls control-flow graph to a specified depth and checks whether the property holds only in the restricted states space. Consequently, the bounded analysis is only bug-hunting (property falsification), not verification. This approach typically involves SAT encoding [Bie+09] of the model and employs iterative deepening of the maximal depth explored.

[Bie+09]: Biere et al. (2009), "Bounded model checking."

4. In *abstract model checking*, we employ a similar approach to abstract interpretation and simplify the state space by representing sets of states abstractly. In contrast to previous techniques, we obtain a system that may satisfy unrealizable properties in the original system as a consequence of overapproximation. To avoid the problem with false positives, tools usually employ CEGAR (counterexample-guided abstraction refinement) that checks the feasibility of possible property violation and refines the abstraction accordingly [Cla+00].

[Cla+00]: Clarke et al. (2000), "Counterexample-Guided Abstraction Refinement"

The essential capability of model checking that distinguishes it from pure interpretation is the detection of equivalent states. That is to determine whether the analysis has already visited the state.

Recall a definition of program state $\sigma = (\varepsilon_s, \varepsilon_p, \mu)$ that describes the content of registers and heap memory. It is given as a tuple of partial maps which assign values from $\mathcal{C}$ to registers and memory locations.

In abstraction, we extend the concretization γ to operate on states. We say that $\gamma(\hat{\sigma})$ represents all concrete states described by abstract state $\hat{\sigma}$. Since abstraction is applied per state elements (registers and memory objects), it is essentially a cartesian product of abstract and concrete values. Based on the cartesian product described in Theorem 2.3.1, we can employ the concretization can be performed piecewise over state elements. The resulting set of concrete states will be a combination of all partial concretizations.

Definition 3.6.1 *Two abstract states $\hat{\sigma}$ and $\hat{\sigma}'$ are equal if $\gamma(\hat{\sigma}) = \gamma(\hat{\sigma}')$.*

The exercise of this thesis is to introduce abstraction noninvasively into concrete execution and explicit-state model checking. The proposed approach is to express abstraction through concrete semantics and instrument such computation into concrete program. For instance, values from the interval domain can be represented as two integer values. The following chapters 4, 5 and 6 will discuss the approach to abstraction being agnostic to whether we employ a compilation-based or more traditional interpretation-based approach. Nevertheless, this distinction should be kept in mind. We will delve into details of the compilation-based technique in later Chapter 8 and Chapter 9 when we have solid abstraction foundations from previous chapters. Specifically, Chapter 8 will discuss possible approaches to abstraction and related techniques, and Chapter 9 will entirely focus on proposed compilation-based abstraction and its technical details, particularly on how we can instrument abstract semantics into the concrete program – referred to as syntactic abstraction.

3.7 Dataflow Analysis

A special instance of abstract interpretation is a dataflow analysis [MZ17]; its goal is to gather information about the way the variables are defined and used in the program. We recognize several elementary dataflow analyses, including *liveness analysis*, which gathers information about live variables at each location, and *reaching definitions analysis*, which identifies an earlier statement whose target variable can reach (be assigned to) the given variable without an intervening assignment (cf. Figure 3.10). Akin to abstract interpretation, dataflow analysis utilizes equations describing how data flows through a program. Essentially, for each control flow edge, we define how input values (in) should be treated, and what information should be forwarded as output (out). These equations can also be referred to as transfer functions, as they transfer dataflow information along.

[MZ17]: Mastroeni et al. (2017), "Abstract Program Slicing: An Abstract Interpretation-Based Approach to Program Slicing"

To compute the general form of inputs and outputs for control flow edge n, we follow these steps (see also Figure 3.9). First, input values are

collected from all predecessors. Then, output values are given by new input values in_n and removing a set of kill_n values and adding new gen_n values. The sets gen and kill are determined by the specific analysis being used. For example, in the case of reaching definitions analysis, new variable definitions are *generated*, while definitions eliminated in the current block are *killed*.

$$\mathsf{in}_n \stackrel{\text{def}}{=} \bigcup_{p \in \mathsf{pred}(n)} \mathsf{out}_p$$
$$\mathsf{out}_n \stackrel{\text{def}}{=} (\mathsf{in}_n - \mathsf{kill}_n) \cup \mathsf{gen}_n$$

Figure 3.9: Definitions of input and output transfer functions for reaching definitions – *forward may* – analysis.

Dataflow analysis can be either forward or backward. In the case of forward analysis, data is propagated in accordance with the direction of control-flow edges, whereas backward analysis propagates in the opposite direction.

Another distinguishing feature of dataflow analysis is whether it provides guaranteed information, which is referred to as *must* analysis, or potential information, also known as *may* analysis. These are differentiated by the use of their confluence operator; in the case of *may* analysis, we unify results, whereas *must* analysis employs an intersection of gathered information in its equations.

```
1 x ← amb
2 x ← 4
3 y ← x
```

Figure 3.10: An example of reaching definitions. Here definition from line 1 is no longer reaching the definition before line 3 because the second definition kills its reach.

As with abstract interpretation, we can utilize the fixpoint iteration to solve the dataflow equations system. We leverage a worklist algorithm that maintains a list of locations that require processing. These locations are intuitively those that have not yet been processed or those that have been influenced by a previous computation.

3.7.1 Points-to Analysis

In our language, it is essential to take into account the flow of data through addressable memory. When we store a value in a particular location, we want to identify all the possible places that can read from that memory, so that we can forward the information to those places. However, in the static analysis, such as dataflow analysis, we do not have a specific location at hand. When we perform a store, we only know a pointer register into which we store the value.

To address this, we need static information about so-called *points-to sets* (alias sets), denoted as Pt, which are sets of pointer registers that may point to the same location. We denote a points-to set for a pointer register $\mathtt{r}_\mathtt{p}$ as $\mathsf{pt}(\mathtt{r}_\mathtt{p}) \in \mathsf{Pt}$. With point-to sets in hand, we can treat them in the dataflow analysis as registers (variables) [Cho+96; Ste95]. This means that the store operation behaves as a value assignment, and the load operation behaves as a read from a "points-to register". This approach allows us to perform dataflow analysis uniformly.

[Cho+96]: Chow et al. (1996), "Effective representation of aliases and indirect memory operations in SSA form"
[Ste95]: Steensgaard (1995), "Sparse functional stores for imperative programs"

To obtain the point-to sets, we use points-to analysis, which comes in various flavors, such as context-sensitive and field-sensitive. The context-sensitive analysis considers various paths to specific memory manipulations, while field-sensitive analysis differentiates between stores to specific fields of memory objects. Irrespective of the algorithm used, we obtain points-to sets with some degree of precision. The simplest option would be to overapproximate all memory manipulations to a single points-to set, which would result in propagation from each store to all memory loads in the program. Regardless of the specific algorithm used, we will leverage the points-to information in the following dataflow analysis through memory.

3.7.2 Reaching Abstraction

In our program, values can originate from two sources: internal (constants) or external (program inputs) resulting from `amb` expressions. All other expressions in the program operate on values from one of these sources, either directly or transitively. In the following, we are interested in categorizing expressions based on the origins of operands they compute with. Specifically, we want to classify expressions to three categories: expressions computing with values of internal origins, external origins, or mixed origins depending on the dynamic control flow. This information is particularly useful when we want to differentiate which operations need to be performed concretely and which abstractly. Therefore, we will refer to these categories as concrete, may abstract, and must abstract, respectively.

Our goal is to determine, for each location in the program, which origins have content in the registers up to that point.[7] Formally we assign numbers (identifiers) N to all the origins in the program. We distinguish two sets of origins: external origins, denoted by $A \subseteq N$, corresponding to `amb` expressions, and internal origins, denoted by $C \subseteq N$. At each location in the program, we want the dataflow analysis to assign a set of numbers from N to each live register. For example, when we assign a constant value to a register, we assign the corresponding number $n \in C$ to the register in that location and overwrite (kill) whatever was assigned to it previously. When we assign the result of a binary operation to a register, we say that its origin is the union of the origins of both operands involved in the operation (we perform a may analysis).

7: Consider the program from Figure 3.10. At the end of the program, we say that all variables have constant origins; however, after the first line, x holds a value of external origin.

Register origins are classified by assigned numbers $R \subseteq N$:

1. a register holds only concrete values if $R \subseteq C$,
2. a register *may* hold an abstract value if $R \cap A \neq \emptyset$,
3. a register *must* hold an abstract value if $R \subseteq A$.

Since this corresponds to a may analysis, where we unify sets of origin numbers, we can simplify the numbering system to only the relevant categories. We use v to represent a concrete set, $\hat{v}$ to represent a completely abstract set, and $\bar{v}$ to represent a diverse set of concrete and abstract origins. The dataflow analysis operates on these values. For example, when an operation has operands characterized as both $\hat{v}$ and v, its result obtains a mixed origin $\bar{v}$. As a matter of fact, we perform this analysis on a lattice (cf. Figure 3.11), which defines how we unify the information in the data flow.

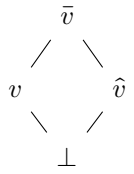


Figure 3.11: Value origin lattice.

This categorization scheme is limited to scalar computations. When considering memory, we must also determine whether a pointer may point to abstract content or whether the pointer itself is of external origin. Additionally, pointers can be nested, which needs to be reflected in the analysis. Therefore, in Definition 3.7.1, we define a more detailed dataflow domain RA that captures this additional information of reaching abstraction.

Definition 3.7.1 *The elements of the reaching abstraction domain Ra are defined inductively:*

- *content can be arbitrary* $\top \in \mathsf{RA}$ *or unspecified* $\bot \in \mathsf{RA}$,
- *scalar content can be* $\mathsf{v}, \hat{\mathsf{v}}, \bar{\mathsf{v}} \in \mathsf{RA}$, *which represent a concrete, a must abstract, or a may abstract value,*
- *if* $x \in \mathsf{RA} \setminus \{\bot\}$ *then* $\mathsf{P}(x), \hat{\mathsf{P}}(x), \bar{\mathsf{P}}(x) \in \mathsf{RA}$, *which represent a concrete pointer, a must abstract pointer, or a may abstract pointer to* x.

Note that the definition allow arbitrarily nested pointers. Fox instance $\mathsf{P}(\mathsf{P}(\hat{\mathsf{v}}))$ is a pointer to a pointer to an abstract value, or $\hat{\mathsf{P}}(\mathsf{P}(\top))$ is an abstract pointer to concrete pointer to anything.

Moreover, we define an ordering $\sqsubseteq$ on the domain RA, which determines how information merges in the data flow. For instance, if a register contains both a concrete value and an abstract value, we assume that the register can hold a mixed value. In more complex cases where a pointer may point to a scalar value or a pointer to a value, we consider the pointer to have an arbitrary destination, denoted as $\mathsf{P}(\top)$. An example of the intended reaching abstraction labeling is shown in Figure 3.12.

```
a ← malloc sizeof(i32)
p ← malloc sizeof(ptr)
x ← amb
c ← amb
if c = 0
    store x → a
else
    store 0 → a
store a → p
b ← load a
```

At the location after the last statement, the data flow of reaching abstraction results in the following labeling:

$\mathsf{x} \mapsto \hat{\mathsf{v}}$, $\mathsf{c} \mapsto \hat{\mathsf{v}}$, $\mathsf{b} \mapsto \bar{\mathsf{v}}$
$\mathsf{a} \mapsto \mathsf{P}(\bar{\mathsf{v}})$, $\mathsf{p} \mapsto \mathsf{P}(\mathsf{P}(\bar{\mathsf{v}}))$

Since `amb` produces abstract values, we wrap them as pointers into allocated locations. The store of constant to `a` creates a mixed value.

Figure 3.12: Dataflow labeling example.

Definition 3.7.2 *Reaching abstraction order* $\sqsubseteq$ *is defined inductively as:*

- $\bot$ *is the smallest element, i.e.,* $\bot \sqsubseteq x$ *for all* $x \in \mathsf{RA}$,
- $\top$ *is the largest element, i.e.,* $x \sqsubseteq \top$ *for all* $x \in \mathsf{RA}$,
- $v \sqsubseteq \bar{v}$ *and* $\hat{v} \sqsubseteq \bar{v}$,
- $\mathsf{P}(x) \sqsubseteq \bar{\mathsf{P}}(x)$ *and* $\hat{\mathsf{P}}(x) \sqsubseteq \bar{\mathsf{P}}(x)$ *for all* $x \in \mathsf{RA}$,
- *if* $x_1 \sqsubseteq x_2$ *then* $p(x_1) \sqsubseteq p(x_2)$ *where* $x_1, x_2 \in \mathsf{RA}$, $p \in \{\mathsf{P}, \hat{\mathsf{P}}, \bar{\mathsf{P}}\}$

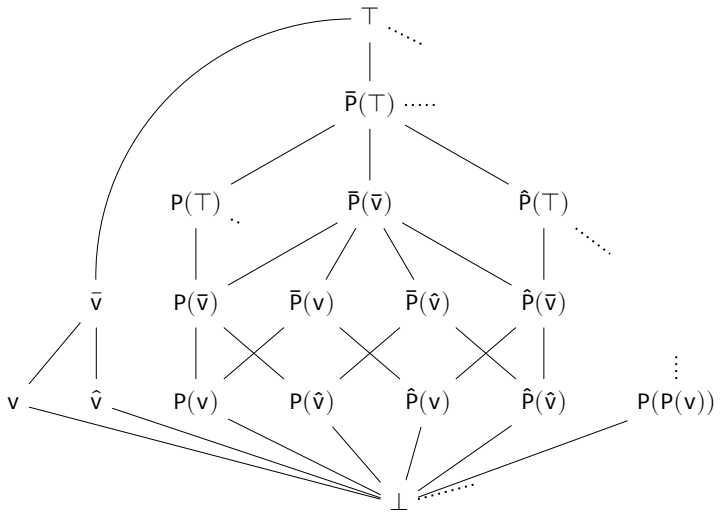


Figure 3.13: Hasse diagram for reaching abstraction domain RA.

Reaching abstraction analysis seeks to create a mapping $\delta : \mathsf{Reg} \cup \mathsf{Pt} \to \mathsf{RA}$ for each program location. This map $\delta \in \Delta$ assigns abstractness information to all live registers Reg and points-to sets Pt at a particular location. This mapping is given as a solution to the transfer equations from Figure 3.14.

The input in Figure 3.14 is computed as the intersection of all maps, meaning that all information about registers and points-to sets is unified according to the RA order definition. The step_n (transfer function) defines how a particular statement n modifies the reaching abstractions map. We will now define step function for our language.

$$\mathsf{in}_n \stackrel{\text{def}}{=} \bigsqcup_{p \in \mathsf{pred}(n)} \mathsf{out}_p$$
$$\mathsf{out}_n \stackrel{\text{def}}{=} \mathsf{step}_n(\mathsf{in}_n)$$

Figure 3.14: Reaching abstraction system of transfer functions.

We define function $\mathsf{gen} : \mathbb{E} \times \Delta \to \mathsf{RA}$ for each expression to determine its origins. For arithmetic, relational, and cast expressions, the gen function typically involves taking the join of the operands.[8] We formally define the gen function as follows:

8: Note that relational operations do not return a concrete boolean since, in abstraction, the result can be one of three values: true, false, and maybe.

Definition 3.7.3 *We write* $\mathsf{gen}(\mathsf{E})_\delta : \mathbb{E} \times \Delta \to \mathsf{RA}$ *to obtain reaching abstraction value for expression* $\mathsf{E} \in \mathbb{E}$ *and map* $\delta \in \Delta$*:*

$$\begin{aligned}
\mathsf{gen}(\textbf{amb})_\delta &\stackrel{\text{def}}{=} \hat{\mathsf{v}} && (\textit{ambiguous inputs})\\
\mathsf{gen}(\mathsf{c})_\delta &\stackrel{\text{def}}{=} \mathsf{v} && (\textit{constants})\\
\mathsf{gen}(\mathtt{r})_\delta &\stackrel{\text{def}}{=} \delta(\mathtt{r}) && (\textit{registers})\\
\mathsf{gen}(\textbf{not}\ \mathsf{b})_\delta &\stackrel{\text{def}}{=} \mathsf{gen}(\mathsf{b})_\delta && (\textit{logic negation})\\
\mathsf{gen}(x_1 \circ x_2)_\delta &\stackrel{\text{def}}{=} \mathsf{gen}(x_1)_\delta \sqcup \mathsf{gen}(x_2)_\delta && (\textit{binary operations})\\
\mathsf{gen}(\textbf{cast}\ \mathsf{s}\ \mathtt{to}\ \mathsf{ty})_\delta &\stackrel{\text{def}}{=} \mathsf{gen}(\mathsf{s})_\delta && (\textit{cast operations})
\end{aligned}$$

With the generation function for expressions, we can define the step function for program statements. We use $\mathsf{step}(\mathtt{stmt})_\delta \stackrel{\text{def}}{=} \delta'$ to indicate a reaching abstraction step that results in a new map δ'. During the dataflow analysis, we always join the dataflow label to a register or points-to set, expressed as $\mathtt{r} \mapsto \delta(\mathtt{r}) \sqcup x$. To abbreviate this notation, we write $r \mapsto_\sqcup x$ to denote the join of x to the current label in $\delta(\mathtt{r})$.

Definition 3.7.4 (Assignment statements) *Register assignment statements update register's label with assigned expression label:*

$$\mathsf{step}(\mathtt{r} \longleftarrow \mathsf{E})_\delta \stackrel{\text{def}}{=} \delta[\mathtt{r} \mapsto_\sqcup \mathsf{gen}(\mathsf{E})_\delta]$$

Definition 3.7.5 (Heap statements) *Allocation creates a pointer to unspecified memory, and deallocation does not influence reaching abstraction at all:*

$$\mathsf{step}(\mathtt{r_p} \longleftarrow \textbf{malloc}\ \mathsf{s})_\delta \stackrel{\text{def}}{=} \delta[\mathtt{r_p} \mapsto_\sqcup \mathsf{P}(\bot)]$$
$$\mathsf{step}(\textbf{free}\ \mathtt{r_p})_\delta \stackrel{\text{def}}{=} \delta$$

Definition 3.7.6 (Memory accessing statements) *Instead of assigning values to registers, the memory manipulation operations act on points-*

to sets that are virtual values representing all possible locations for a particular pointer.

$$\mathsf{step}(\textbf{store}\ x \longrightarrow \mathtt{r_p})_\delta \stackrel{\text{def}}{=} \delta[\mathsf{pt}(\mathtt{r_p}) \mapsto_\sqcup \mathsf{gen}(x)_\delta]$$
$$\mathsf{step}(\mathtt{r} \longleftarrow \textbf{load}\ \mathtt{r_p}\ \textbf{of}\ \mathtt{ty})_\delta \stackrel{\text{def}}{=} \delta[\mathtt{r} \mapsto_\sqcup \delta(\mathsf{pt}(\mathtt{r_p}))]$$

It is important to note that since functions are pointers, we also define point-to sets on their symbol names. This allows us to resolve indirect dataflow calls as well.

Definition 3.7.7 (Call statement) *During a function call, data flow is forwarded from call arguments to function parameters.*

$$\mathsf{step}(\mathtt{r} \longleftarrow \textbf{call}\ \mathtt{fun}(\mathtt{E}_1, \ldots, \mathtt{E}_n))_\delta \stackrel{\text{def}}{=} \delta[\mathtt{r} \mapsto_\sqcup \delta(\mathsf{pt}(\mathtt{fun}))] \sqcup \delta[p_i \mapsto_\sqcup \mathsf{gen}(\mathtt{E}_i)_\delta]$$

where p_i is ith parameter of function `fun`.

Definition 3.7.8 (Return statement) *Return from function propagates to all possible call sites:*

$$\mathsf{step}(\textbf{ret}\ \mathtt{E}) \stackrel{\text{def}}{=} \delta[\mathsf{pt}(\mathtt{fun}) \mapsto_\sqcup \mathsf{gen}(\mathtt{E})_\delta]$$

where here `fun` *is a function in which the return statement is located.*

With the step function defined, we can now describe a work list algorithm for reaching abstraction. This algorithm follows the standard template for the forward dataflow analysis fixpoint algorithm. By default, we assign all values $\bot$. In the dataflow worklist algorithm, we iteratively refine the map.

Algorithm 1: Reaching Abstractions Work list Algorithm

```
Input  : control flow graph G = ⟨Q, E⟩
1  forall q ∈ Q do
2  |  q ← ∅
3  end
4  worklist ← Q
5  while worklist is not empty do
6  |  n ← pop(worklist)
7  |  forall p ∈ pred(n) do
8  |  |  in(n) ← in(n) ⊔ out(p)
9  |  end
10 |  out(n) = step_n(in(n))
11 |  if out(n) changed then
12 |  |  foreach s ∈ succ(n) do
13 |  |  |  push(s, worklist)
14 |  |  end
15 |  end
16 end
```

In summary, the reaching abstraction are computed using the step operation on each operation in the work list. If the operation changes the reaching abstractions map δ, we propagate this information forward to all successor statements. It is important to note that these successors are not control flow successors but rather data dependent statements (cf. Definition 3.7.9).

Fortunately, the SSA form of LLVM IR provides a suitable representation of the dependencies (*def-use* chain) for our dataflow approximation [SX18]. This allows us to utilize the inherent direct dataflow graph to obtain data dependencies. The same approach can be used for our language as well. The direct data dependencies can be obtained by identifying the users of a written register. Moreover, we need to address memory dataflow dependencies, which are represented using points-to sets. As mentioned earlier, these behave as regular registers – we can operate on their users in the same manner. In the following, we give a definition of successors for our language.

Definition 3.7.10 *The* succ *relation refers to a set of dependent statements influenced by a given statement stmt. If the dataflow algorithm updates δ based on stmt, we also need to update the values of its dependent statements.*

$$
\begin{aligned}
\mathsf{succ}(\mathtt{r} \longleftarrow \mathtt{E}) &\stackrel{\text{def}}{=} \mathsf{users}(\mathtt{r}) \\
\mathsf{succ}(\mathtt{r_p} \longleftarrow \textbf{malloc}\ \mathtt{s}) &\stackrel{\text{def}}{=} \mathsf{users}(\mathtt{r_p}) \\
\mathsf{succ}(\textbf{store}\ \mathtt{v} \longrightarrow \mathtt{r_p}) &\stackrel{\text{def}}{=} \mathsf{users}(\mathsf{pt}(\mathtt{r_p})) \\
\mathsf{succ}(\mathtt{r} \longleftarrow \textbf{load}\ \mathtt{r_p}\ \textbf{of}\ \mathtt{ty}) &\stackrel{\text{def}}{=} \mathsf{users}(\mathtt{r}) \\
\mathsf{succ}(\mathtt{r} \longleftarrow \textbf{call}\ \mathtt{fun}(\mathtt{E}_1, \ldots, \mathtt{E}_n)) &\stackrel{\text{def}}{=} \{p \mid p \textit{ is a parameter of } \mathtt{fun}\} \cup \\
&\qquad \mathsf{users}(\mathtt{r}) \\
\mathsf{succ}(\textbf{ret}\ \mathtt{E}) &\stackrel{\text{def}}{=} \mathsf{users}(\mathsf{pt}(\mathtt{fun}))
\end{aligned}
$$

In Figure 3.15, we illustrate the process of reaching abstraction. Starting from amb expression (1), we propagate to all its users (2) and (3), including a loop condition stored in an intermediate register, denoted as *cond* in the locations map. Propagated values are $\bar{v}$ due to combining concrete and abstract origins. Finally, we propagate origins from z to its user w (4).

Figure 3.16 shows dataflow propagation to and from a function. Starting from x (1), we assign an abstract origin to the first parameter (2) and propagate it to the return value (3), resulting in $\bar{v}$ due to mixed-domain computation. We then propagate the information of possible abstraction to all call sites of the function add (4), (5). In the second call, we mark the second parameter as potentially abstract (6), and since computation (7) does not generate new information, we do not need to propagate the result further.

In this thesis's presented compilation-based abstraction, we use the reaching abstraction technique to distinguish between concrete and abstract execution. By doing so, we are able to minimize the number of operations that need to be abstracted. Furthermore, the RA domain granularity allows us to discern the particular type of abstraction being used, whether it is scalar, aggregate, or pointer abstraction.

Definition 3.7.9 *A statement s_r depends on statement s_w if there is an execution in which statement s_r reads from memory or registers written by statement s_w.*

[SX18]: Sui et al. (2018), "Value-flow-based demand-driven pointer analysis for C and C++"

```
x ← amb  (1) ⟅δ[x ↦ v̂]⟆
y ← 1
while x < y  (2) ⟅δ[cond ↦ v̄]⟆
  z ← x + 2  (3) ⟅δ[z ↦ v̄]⟆
  y ← y + 1
w ← z  (4) ⟅δ[w ↦ v̄]⟆
```

Figure 3.15: A simple reaching abstraction example.

After the first call of function add:

```
fn add(a, b)  (2) ⟅δ[a ↦ v̂]⟆
  ret a + b  (3) ⟅δ[r ↦ v̄]⟆

fn main()
  x ← amb  (1) ⟅δ[x ↦ v̂]⟆
  y ← add(x,1)  (4) ⟅δ[y ↦ v̄]⟆
  z ← add(1,x)  (5) ⟅δ[z ↦ v̄]⟆
```

After the second call of function add:

```
fn add(a, b)  (6) ⟅δ[a, b ↦ v̄]⟆
  ret a + b  (7) ⟅δ[r ↦ v̄]⟆

fn main()
  x ← amb
  y ← add(x, 1)
  z ← add(1, x)
```

Figure 3.16: A function call dataflow example.

This approach sets us apart from compilation-based analyses such as SymCC [PF20] and SymSan [Che+22]. These tools instrument all operations in the LLVM IR; consequently, they must determine this information during program execution.

Summary

In this chapter, we introduced the model of computation that we will use throughout the rest of the thesis, which is an LLVM-like language. We provided its syntax and concrete semantics. Furthermore, we gave an overview of program analysis techniques, like abstract interpretation, execution, and model checking, which we have related to the presented execution model. Finally, we have delved into dataflow analysis, presenting a new reaching abstraction analysis. This analysis is fundamental for the syntactic analysis described in Chapter 9.

4 Scalar Abstraction

The content of this chapter is based on the following publication:

- Henrich Lauko, Petr Ročkai, and Jiří Barnat. "Symbolic Computation via Program Transformation." In: *Theoretical Aspects of Computing – ICTAC 2018. 15th International Colloquium, Stellenbosch, South Africa, October 16-19, 2018, Proceedings*. Ed. by Bernd Fischer and Tarmo Uustalu. Cham: Springer, 2018. ISBN: 978-3-030-02507-6 [LRB18]
- Henrich Lauko, Vladimír Štill, Petr Ročkai, and Jiří Barnat. "Extending DIVINE with Symbolic Verification Using SMT. (Competition Contribution)." In: *Tools and Algorithms for the Construction and Analysis of Systems. 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part III*. Cham: Springer, 2019. ISBN: 978-3-030-17501-6 [Lau+19]
- Henrich Lauko, Petr Ročkai, Vladimír Štill, and Jiří Barnat. "DIVINE – Model Checker for C++." In: *Automatic Software Verification*. Ed. by Dirk Beyer. (forthcoming) [Lau+ng]

Scalar abstraction concerns elementary program values such as integers, booleans, and floating-point values. Its goal is to circumvent the need to enumerate all possible ambiguous values and instead work with a set of values at once. The challenge lies in effectively representing this set of abstract values, as full set enumeration does not offer much advantage. Scalar abstractions typically describe sets of values through their properties, whether the value is positive or negative, whether it falls into a certain range, or satisfies some generic predicate. The selection of these properties results in varying levels of abstraction precision, which is a balancing act between domain precision and the efficiency of its operations. As seen in sign analysis, less complex properties often lead to more efficient analysis but may sacrifice precision.

The preceding chapters introduced simple language and common techniques for program analysis. This chapter and the following will explore various abstract domains and formalize their abstract semantics. This chapter focuses on scalar abstract domains with an emphasis on symbolic abstraction, which is the main topic of my published work. The following research questions will be addressed in this chapter:

RQ1 What domains are suitable for program verification, and what are their practical applications?

RQ2 How does abstraction interfere with concrete semantics, control flow and other domains, and what are the implications for program analysis?

RQ3 How can we improve the precision of scalar abstraction during program analysis, and what are effective refinement techniques?

RQ4 How to re-cast symbolic execution as a scalar abstract domain?

To start, we recall a value abstraction, providing necessary operators. We also state the relationship to the concrete execution described in Chapter 3. Thereafter, we present a formal description of commonly known domains for program analysis like interval or sign. Alongside this, we present extensions of these simple domains for verification-specific use cases. For now, we will consider only scalar abstraction.

4.1 State of the Art

We have already mentioned a few scalar domains: ***sign domain*** $\mathcal{A}_{\pm}$ or ***interval domain*** $\mathcal{A}_i$. They are examples of domains from a broad category of *non-relational numeric domains*. These are domains that reason about a single value and can not inherently utilize relational information between program values.

A ***constant propagation domain*** $\mathcal{A}_{cp}$ (see Figure 4.1) is another example of property domain used in optimization techniques [Kil73]. Probably the simplest of domains is a ***single value*** domain $\mathcal{A}_\bigstar$. The single value domain is useful if we want to omit a variable from the model of the program. The two-value domain that contains two possible elements $(\top, \bot)$ is also called the ***definedness domain*** $\mathcal{A}_D$. Because with $\top$, we can denote that value is defined, respectively undefined with $\bot$. Despite its simplicity, it can be useful to detect reachability of program locations.

[Kil73]: Kildall (1973), "A Unified Approach to Global Program Optimization"

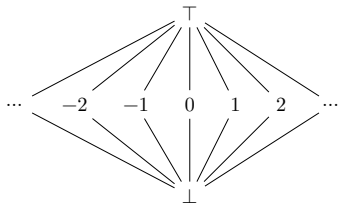


Figure 4.1: Lattice of constant propagation abstract domain $\mathcal{A}_{cp}$.

Since the precision of an abstract domain highly depends on the program structure, many abstract domains were designed to track various program properties. The first domain presented by Cousot [CC77a] to achieve more precise results was an ***interval domain*** $\mathcal{A}_i$. In the interval abstraction, a set of values is represented by its minimal and maximal value. Because in some cases, it is not possible to determine the boundary values, the interval domain utilizes infinities to denote arbitrary bound – we represent arbitrary value $\top_i$ by $[-\infty, \infty]$ and undetermined value $\bot_i$ by an empty interval $\emptyset$.

Unfortunately, with the infinite lattice of the interval domain,[1] abstract interpretation will obtain a possible infinite or impractically long executions (e.g., when the interpretation can not determine the bound of a loop it can increment the boundary of an interval indefinitely never reaching the fixpoint). Therefore, abstract interpreters employ ***widening*** and ***narrowing*** techniques to accelerate convergence [CZ11; CC92]. The idea of widening is to overshoot the least fixpoint after few unsuccessful iterations of the interpretation and subsequently by narrowing to refine the over-approximated solution. By different implementation of widening and narrowing, abstract interpreters employ different strategies to achieve convergence – for example widening with thresholds [Bla+03; LJG11], delayed widening, parma widening [Bag+05] or abstract acceleration [FG10; GH06].

1: The interval domain does not meet ascending chain condition.

[CZ11]: Cortesi et al. (2011), "Widening and narrowing operators for abstract interpretation"
[CC92]: Cousot et al. (1992), "Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation"
[Bla+03]: Blanchet et al. (2003), "A Static Analyzer for Large Safety-critical Software"
[LJG11]: Lakhdar-Chaouch et al. (2011), "Widening with Thresholds for Programs with Complex Control Graphs"
[Bag+05]: Bagnara et al. (2005), "Precise widening operators for convex polyhedra"
[FG10]: Feautrier et al. (2010), "Accelerated Invariant Generation for C Programs with Aspic and C2fsm"
[GH06]: Gonnord et al. (2006), "Combining Widening and Acceleration in Linear Relation Analysis"

An alternative approach to tackle infinite interpretation that does not require any extrapolation operator is ***policy iteration*** [Cos+05; Gau+07; GS07a; GS07b; GS11]. The idea of policies is to compute fixpoint solution of a sequence of simpler semantic equations, such that the least fixpoint is reached after a finite number of iterations. The sequence of policies

defines a strategy to approach the fixpoint either from above or below. Policies are formed from a decomposition of original abstraction.

Scalar abstract domains, in LART, take care of representation of the ground LLVM values, such as integers and floating-point values. The traditional scalar domains are also known in the literature as *numerical abstract domains* defined by Cousot in [CC77a]. In Cousot's original approach, the abstraction is designed only for programs without memory and recursion; hence, abstract only program variables and performs context-insensitive analysis. We will call this restricted model as *Cousot's program model*. Since LLVM IR does not directly correspond to this model, we need to adjust Cousot's model to our state description. In LLVM IR for an abstract domain $\mathcal{A}$, the numerical abstraction extends the state definition so that registers can hold abstract values. These program registers can be handled as Cousot's variables.

However, the problem arises on the boundaries with memory. The simplest solution would be to lower abstract values to concrete domain with concretization function γ, but we would lose any advantage of abstraction. Moreover, the concretization might not even be finite. Hence, we must expand our program's memory to hold abstract scalar values.

Cousots's model can also be extended to context-sensitive analysis, either by abstracting call-stack for interprocedural analysis [BW04; SJ11], or summarization of functions [BH19]. These techniques are extended to handle recursion in multiple works [CC77c; CC01]. Alternatively, interprocedural analysis can be achieved by a dynamic approach. Dynamic execution allows the analysis to preserve context-sensitivity because it preserves call-stack as in the state of the program. Since we aim to perform abstract execution of LLVM dynamically, we will obtain the interprocedural analysis as a consequence of execution.

Furthermore, it is possible to combine multiple domains to reason about multiple properties either by keeping each property separately or by reduced product of domains (see Figure 4.2), which is more precise and more efficient than multiple separate interpretations [CCM11].

Several approaches have been taken to adapt traditional numerical domains for bit-precise analysis. For example, Miné et al. in [Min12] adapted the interval and modular domains to handle computer integers in wrapped arithmetic and used a predicate domain for floating-point computation. [SR17] presents a generic way to adapt domains that reason about affine-inequalities to an overflow-aware analysis, and [DOM21] takes also endianness into account.

As the semantics of computer floating point numbers diverges even further from a mathematical representation of numbers and the precise symbolic representation usually struggles even with small programs, there have been multiple attempts to describe them abstractly. For instance, [LSA16] defines a domain that allows operating also on bit-representation of floating point values.

[BW04]: Bertrand et al. (2004), "Abstracting Call-Stacks for Interprocedural Verification of Imperative Programs"
[SJ11]: Sotin et al. (2011), "Precise Interprocedural Analysis in the Presence of Pointers to the Stack"
[BH19]: Boutonnet et al. (2019), "Disjunctive Relational Abstract Interpretation for Interprocedural Program Analysis: 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13–15, 2019, Proceedings"
[CC77c]: Cousot et al. (1977), "Static determination of dynamic properties of recursive procedures"
[CC01]: Cousot et al. (2001), "Compositional Separate Modular Static Analysis of Programs by Abstract Interpretation"
[CCM11]: Cousot et al. (2011), "The Reduced Product of Abstract Domains and the Combination of Decision Procedures"

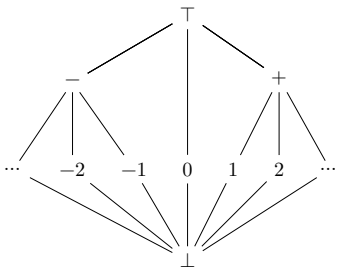


Figure 4.2: $\mathcal{A}_{cs}$ is a joint sign ($\mathcal{A}_{\pm}$) and constant propagation ($\mathcal{A}_{cp}$) domain.

[Min12]: Miné (2012), "Abstract domains for bit-level machine integer and floating-point operations"
[SR17]: Sharma et al. (2017), "Sound Bit-Precise Numerical Domains"
[DOM21]: Delmas et al. (2021), "Static Analysis of Endian Portability by Abstract Interpretation"
[LSA16]: Lee et al. (2016), "Verifying Bit-Manipulations of Floating-Point"

Domain-specific abstractions

Aside from the general-purpose domains discussed earlier, real-world program verification calls for domain-specific abstractions. Most numer-

ical domains, which operate with unbounded numbers, are unsuitable for a sound representation of computer numbers. To accurately reflect properties of bit-precise arithmetic, such as overflows, recent abstract domains have employed bit-vector logic to mimic processor semantics [SR17; STR13]. Similar domains were also developed for floating-point arithmetic [PGM04]. To address memory model properties, predicate abstraction is often used in analysis [Dan+13]. The challenges of machine code analysis are further explored in [Rep+10].

More domain-specific abstractions are used to analyze database querying languages [HC12]; system components like filters [Fer04] or synchronous clocks [Fer05].

An example of system-level static analysis is an Extended Berkely Packet Filter (eBPF) [Fle17; MJ93]. It is a language and run-time used to extend the functionality of the Linux operating system. It serves for generic device drivers communication and permits user code to be executed in kernel context. The Linux kernel has a static analyzer that can prevent the execution of BPF code that violates rules, such as performing illegal operations like dividing by zero, leaking information from kernel data structures, or accessing kernel memory outside the permitted bounds. To accomplish this, the abstractions used must match the low-level operational semantics [Ger+19]. For instance, an abstract domain of tristate numbers *tnum* [Vis+21] keeps track of which bits are `0`, `1`, or unknown – this allows for sound bit-precise static analysis.

[SR17]: Sharma et al. (2017), "Sound Bit-Precise Numerical Domains"
[STR13]: Sharma et al. (2013), *An abstract domain for bit-vector inequalities*
[PGM04]: Putot et al. (2004), "Static Analysis-Based Validation of Floating-Point Computations"
[Dan+13]: Dan et al. (2013), "Predicate abstraction for relaxed memory models"
[Rep+10]: Reps et al. (2010), "There's plenty of room at the bottom: Analyzing and verifying machine code"
[HC12]: Halder et al. (2012), "Abstract interpretation of database query languages"
[Fer04]: Feret (2004), "Static Analysis of Digital Filters"
[Fer05]: Feret (2005), "The Arithmetic-Geometric Progression Abstract Domain"
[Fle17]: Fleming (2017), *A thorough introduction to eBPF*
[MJ93]: McCanne et al. (1993), "The BSD Packet Filter: A New Architecture for User-Level Packet Capture"
[Ger+19]: Gershuni et al. (2019), "Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions"
[Vis+21]: Vishwanathan et al. (2021), "Semantics, Verification, and Efficient Implementations for Tristate Numbers"

Corner Case Domains

Many program errors can usually be detected by examining a small subset of input values that may influence the program's behavior. These can be specifically treated constant values, default-initialized values (null values), or other corner cases. For many of these, we can devise characteristic corner case-targeting domains, like the nullity domain, to explore null pointer dereference-related errors or floating point domain to represent all corner-case values like infinities, zeros, or `NaN`.

The sign domain is, in a way, a corner case domain that defines three cases for values: negative, positive, or zero. This approach can be expanded beyond integers to floating point values, where the corner cases can be defined as infinities, two zeros, and `NaN`. One of the benefits of using corner-case domains is their efficient implementation, as operations can be represented as transfer tables. Additionally, these domains can be combined with functor domains, such as the powerset domain, to capture all possible corner cases while maintaining precision. Moreover, their simplicity, as their representation usually resembles the concrete values they represent, allows for abstraction optimization.

4.2 Abstract Semantics

In the design of the domain, it is crucial to consider the objective of the abstract analysis, which is to determine *what should the abstract analysis compute*? Program proving tools aim to overapproximate the program state space to avoid missing any potential errors. The lack of an error

state in the abstracted state space implies that such a state does not exist in the concrete state space. On the other hand, test generators or bug-finding tools aim to underapproximate all program states to simplify the analysis. In case of a reachable error, the abstract analysis identifies a fault-inducing input (a test case).

The choice of abstraction is always a trade-off between precision, soundness, and efficiency. We will now explore multiple abstract domains ranging from the most simple unit domain to the most precise symbolic abstraction. The soundness of domains varies depending on how one models program semantics, whether the domain represents bit-precise numbers, whether it can detect overflows, or how precisely it models memory interactions. Abstractions make even more significant compromises in concurrent environments. Depending on the granularity of the steps, abstraction can omit some program interleavings or weak memory behavior. This all leads to unsound but still valuable analysis.[2]

2: Soundness is already compromised by considering only one program representation. Compilers usually enforce a predetermined order of evaluation that can lead to behavior omissions which would be present if the code is compiled differently or for a different architecture. Further, analysis tools usually operate on Harvard architecture, where the analyzed code cannot be accessed or altered, which presents a limited and potentially flawed view of the program's actual semantics.

4.2.1 Abstract State

The fundamental advantage of scalar abstraction lies in its capacity to operate independently on each program value, regardless of whether it is stored in a register or memory. In practical terms, it is desirable to enable both concrete and abstract values in the program since it is frequently unnecessary to abstract the entire program. Rather, we may only need to abstract those operations that may encounter abstract values and consequently only their results (as discussed in the previous chapter on reaching abstraction). To achieve this, we expand the definition of concrete state to accommodate abstract scalar values alongside concrete abstract values in scalar registers and memory content. It should be noted that scalar abstraction does not interfere with the description of memory block size, despite it being a scalar value, nor does it allow for abstract pointer arithmetic; these aspects will be addressed in the subsequent chapters on aggregate and pointer abstraction.

Recall concrete semantic domains from Figure 3.2. In the following abstract semantic domains, we will designate with a $\hat{\_}$ symbol all domains that have been modified to incorporate abstract values. To accommodate abstract values from domain $\mathcal{A}$, we expand the state definition by permitting their storage in registers and memory content. Likewise, the definition can also be extended for multi-domain analysis to include multiple scalar abstract domains in respective environments.

$$
\begin{aligned}
\hat{\varepsilon}_s \in \widehat{\mathbb{Env}}_{\mathsf{s}} &\equiv \mathsf{Reg}_{\mathsf{s}} \to \mathcal{C}_{\mathsf{s}} \cup \mathcal{A} && \text{(scalar registers environment)}\\
\varepsilon_p \in \mathbb{Env}_{\mathsf{p}} &\equiv \mathsf{Reg}_{\mathsf{p}} \to \mathcal{C}_{\mathsf{p}} && \text{(pointer registers environment)}\\
\hat{v} \in \hat{\mathbb{V}} &\equiv \mathcal{C}_{\mathsf{s}} \to \mathcal{C} \cup \mathcal{A} && \text{(memory block content)}\\
\langle s, \hat{v} \rangle \in \widehat{\mathsf{Blk}} &\equiv \mathcal{C}_{\mathsf{s}} \times \hat{\mathbb{V}} && \text{(memory blocks)}\\
\hat{\mu} \in \widehat{\mathbb{Env}}_{\mu} &\equiv \mathsf{Id} \to \widehat{\mathsf{Blk}} && \text{(memory environment)}\\
\hat{\sigma} \in \widehat{\mathbb{Env}} &\equiv \widehat{\mathbb{Env}}_{\mathsf{s}} \times \mathbb{Env}_{\mathsf{p}} \times \widehat{\mathbb{Env}}_{\mu} && \text{(program states)}
\end{aligned}
$$

Figure 4.3: Abstract scalar semantic domains.

This gives us the definition of a scalar abstract state, which enables a combination of concrete and abstract values. It is advantageous to perceive

program states as a Cartesian product of domains. This can be theoretically accomplished by projecting all registers and memory contents to a flattened n-tuple representing the program's state values.

As a result, the program state can be treated as a direct Cartesian product of its individual elementary value domains. Scalar operations applied to this state gather elementary scalar values and alter only a single element of the state and its corresponding domain, as required.[3]

3: It is important to note that this perspective in compilation-based abstraction is only theoretical, as all values are, in fact, realized concretely, e.g., intervals as two integer values, and operations may modify multiple concrete values in a single abstract transformation step (both interval bounds). We will describe this later in Chapter 8.

By making use of the Galois connection for constituent domains, we can exploit its inherent properties described in Chapter 2. Specifically, we leverage that the Cartesian product of multiple domains also forms a Galois connection. As a result, the properties of the constituent state domains can be extended to the entire program state. For instance, when we possess an abstract state $\hat{\sigma}$, given a Galois connection, we can view it as a collection of concrete states defined as $\gamma(\hat{\sigma})$, as the concretization can be applied element-wise[4].

4: Contretization in the concrete domain is identity.

To keep things concise, we occasionally represent a program state using the Cartesian product of program domains. Only later on, when we discuss state equivalence checking, which is layout dependent, and the use of explicit tools, we must consider the abstract state's actual physical layout.

4.2.2 Abstract Operators

While program abstraction may involve various specificities related to abstract control and data flow, the essence of abstraction is still maintained in abstract transformers. In other words, to perform abstract program analysis, it is necessary to abstract the program-state transitions that correspond to the edges of a control-flow graph. We refer to $\widehat{op}$ as the abstraction of computational statement $op \in \texttt{stmt}_S$.

The abstract transformer $\widehat{op}$ from domain $\mathcal{A}$ is a map that relates abstract operands and result values. Its arity matches the arity of the concrete counterpart op, but all operands are in the domain of transformer.

Consider a transformer for the sign domain

$$\tau_- = \{(\texttt{0}, \texttt{0}), (\texttt{<0}, \texttt{>0}), (\texttt{>0}, \texttt{<0})\}$$

This transformer describes the unary negation operation, which flips the sign of abstract values. In practice, operations will be more complex, for which we will either write a transfer table or step semantic functions that describe how the operation alters the program state.

In the discussion on abstract operators (cf. Section 2.3), we defined best abstraction of an operator op as $\gamma \circ op \circ \alpha$, which serves as a guide for constructing approximate solutions, though the actual best abstraction is rarely constructible in practice. Nevertheless, when designing a domain, our goal is to make its transformers as close as possible to best abstraction.

For instance, the abstraction of a binary operator $op : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is defined as $\widehat{op} : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$. The most precise abstraction of $\widehat{op}(a, b)$ is

given by the Galois connection, where the composition of concretization and abstraction is defined as

$$\widehat{op} \triangleq \alpha_{\mathcal{A}} \circ op \circ \gamma_{\mathcal{A}}$$

Evaluating *op* on all possible combinations of concretized arguments is usually extremely inefficient and yields diminishing gains for program analysis. In abstraction, we aim to give sound and precise abstract operators that efficiently abstract the computation. To ensure soundness, the operation needs to overapproximate the result of the precise abstraction:

$$op \circ \gamma_{\mathcal{A}} \subseteq \gamma_{\mathcal{A}} \circ \widehat{op}$$

In the remainder of this chapter, we will use this rule as a guiding principle for designing sound abstract operations. In addition to soundness, it is crucial to consider the complexity of operations. The scalar domains presented earlier typically have transformers with constant time complexity. However, when the transformer is not given as a transfer table and involves relational reasoning, the operation can easily become a bottleneck in the analysis. Symbolic execution, for example, often requires using an SMT solver to decide the feasibility of program locations.

In a simple case where a program involves only a single abstract domain if concrete and abstract values appear as operands in the evaluation of a statement, we need to bring them to the shared domain following the previous restriction. Since concretizing is often impractical, we lift the concrete operands to the abstract domain using its abstraction function.

Later on, in Chapter 6, we discuss multi-domain programs and interactions between domains. Nevertheless, we still need to ensure that operations are performed within a single domain. Similarly to the interaction between concrete and abstract values, we can rely on domain conversions to handle the interaction between different abstract domains. We require conversion functions to be sound with respect to concretization. Suppose we have a conversion function $\chi_{\mathcal{A}_1 \to \mathcal{A}_2}$ that converts values from domain $\mathcal{A}_1$ to $\mathcal{A}_2$. In order for this conversion to be sound, it must satisfy the following condition:

$$\gamma_{\mathcal{A}_1} \subseteq \gamma_{\mathcal{A}_2} \circ \chi_{\mathcal{A}_1 \to \mathcal{A}_2}$$

Our approach to abstraction involves defining abstract transformers solely for expressions while preserving control flow explicitly. In later chapters, where we cover memory abstraction, we will also introduce the abstraction of memory-manipulating statements. We refer to this approach control-explicit data-abstract approach, akin to the control-explicit data-symbolic approach presented in [Mrá+16].

This implies that control-flow statements need to be realized concretely. Namely, we need to resolve branching in a concrete domain. In this specific case, we employ concretization as a boolean condition implies only two cases to be considered. However, to do so, we introduce an intermediate domain called the tristate domain (`Tr`), which implements three-value logic.

To use the tristate domain, we require the abstract domain to provide a conversion function $\chi_{\mathcal{A} \to \mathtt{Tr}}$ that maps abstract values into three-value

[Mrá+16]: Mrázek et al. (2016), "Sym-DIVINE: Tool for Control-Explicit Data-Symbolic State Space Exploration"

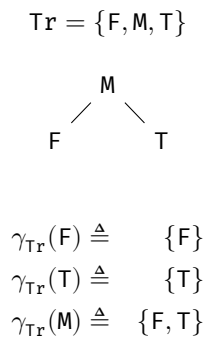


Figure 4.4: The tristate domain `Tr` represents value in a three-value logic: true (`T`), false (`F`), and maybe (`M`).

logic values. This conversion guarantees a uniform interaction with control flow and can be more efficient than direct concretization, which may result in multiple values. Once we have concretized a tristate value for a branch condition, we only need to explore multiple paths if the value is `M` (i.e., if it could be both true and false).

It is important to note that relational operations do not directly return tristate values, as booleans in LLVM IR can still be used as single-bit values in computations. This would lead to unnecessary translations between the tristate and other abstract domains. To differentiate branching from computation, we leverage to-tristate conversion only when an abstract value is used as a branch condition.

Definition 4.2.1 *Conversion to tristate is required to be sound underapproximation regarding the conditioned control flow, i.e., it does not omit a path. That is for $v \in \mathcal{A}$ and conversion $\chi_{\mathcal{A}\to\mathtt{Tr}}(v) = t$ it holds:*

$$\mathtt{T} \in \gamma_{\mathcal{A}}(v) \implies \mathtt{T} \in \gamma_{\mathtt{Tr}}(t)$$
$$\mathtt{F} \in \gamma_{\mathcal{A}}(v) \implies \mathtt{F} \in \gamma_{\mathtt{Tr}}(t)$$

4.3 Scalar Domains

Let us revisit the value domain definition presented in Definition 2.3.6. To expand on this, we introduce an additional set of conversions denoted as χ:

$$(\mathcal{D}, \sqsubseteq, \gamma, \alpha, \bot, \top, \tau, \chi, \cup, \cap)$$

a partial order $\langle \mathcal{D}, \sqsubseteq \rangle$ with lattice operations join $\cup$ and meet $\cap$, an abstraction function α, a concretization function γ, the smallest $\bot$ and the largest element $\top$, and a set of abstract transformers τ. Typically, we do not require abstract transformers for all operations since some operations may not have semantic meaning for a particular domain. Additionally, we may define conversion operators to handle certain cases. For instance, we define a conversion to the tristate domain $\chi_{\mathcal{A}\to\mathtt{Tr}}$ whenever domain may interact with control flow.

Operation	Effect on $\widehat{\sigma}$
$\mathtt{r_s} \longleftarrow \bigstar$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \bigstar]$
$\mathtt{r_s} \longleftarrow$ **amb**	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \bigstar]$
$\mathtt{r_s} \longleftarrow \bigstar \circ \mathtt{v}$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \bigstar]$
$\mathtt{r_s} \longleftarrow \mathtt{v} \circ \bigstar$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \bigstar]$
$\mathtt{r_s} \longleftarrow$ **cast** $\bigstar$ **to** t	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \bigstar]$
$\mathtt{r_b} \longleftarrow \bigstar \diamond \mathtt{v}$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \bigstar]$
$\mathtt{r_b} \longleftarrow \mathtt{v} \diamond \bigstar$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \bigstar]$
$\mathtt{r_b} \longleftarrow \chi_{\mathcal{A}_\bigstar\to\mathtt{Tr}}(\bigstar)$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \mathtt{M}]$
$\mathtt{r_p} \longleftarrow \mathtt{p} + \bigstar$	$\hat{\varepsilon}_p[\mathtt{r_p} \mapsto \bigstar]$
$\mathtt{r_p} \longleftarrow \bigstar + \mathtt{v}$	$\hat{\varepsilon}_p[\mathtt{r_p} \mapsto \bigstar]$
$\mathtt{r_p} \longleftarrow \mathtt{p} - \bigstar$	$\hat{\varepsilon}_p[\mathtt{r_p} \mapsto \bigstar]$
$\mathtt{r_p} \longleftarrow \bigstar - \mathtt{p}$	$\hat{\varepsilon}_p[\mathtt{r_p} \mapsto \bigstar]$

Figure 4.5: Unit domain step semantics, where $\circ$ are arithmetic operations, and $\diamond$ relational operations as described in Definition 3.2.1. Note that this description performs implicit abstraction in the case of mixed computation, as described in previous section. The domain only implements transformations over $\bigstar$ values. For that, we first abstract the concrete values, for example:

$$\bigstar \circ c \triangleq \bigstar \circ \alpha(c) \triangleq \bigstar \circ \bigstar \triangleq \bigstar.$$

4.3.1 Unit Domain

The simplest way to abstract any (non-empty) set of values, is to introduce a single symbol representing all elements of the set, thereby losing all the information contained in the particular values. We call this symbol the unit and denote it by $\bigstar$.

Unit domain $\mathcal{A}_\bigstar$ does not leave space for creativity in transformation design. The most precise and the only abstraction of transformers always takes and returns $\bigstar$ values – see Figure 4.5. The interaction with other domains also does not leave much space. Lowering of $\bigstar$ value needs to yield a $\top$ in a particular domain, e.g., `M` in the tristate domain.

Definition 4.3.1 *Formally, the unit domain $\mathcal{A}_\bigstar$ is defined as:*

$$(\{\bigstar\}, \sqsubseteq_\bigstar, \gamma_\bigstar, \alpha_\bigstar, \bigstar, \bigstar, \tau_\bigstar, \cup_\bigstar, \cap_\bigstar)$$

where order $\sqsubseteq_\bigstar$ and lattice operations $\cup_\bigstar, \cap_\bigstar$ are trivial on single value domain. An abstraction of arbitrary value v is defined as $\alpha(v) \triangleq \bigstar$, whereas concretization yields the whole concrete domain $\gamma_\bigstar(\bigstar) \triangleq \mathcal{C}$. Abstract transformers are given by Figure 4.5.

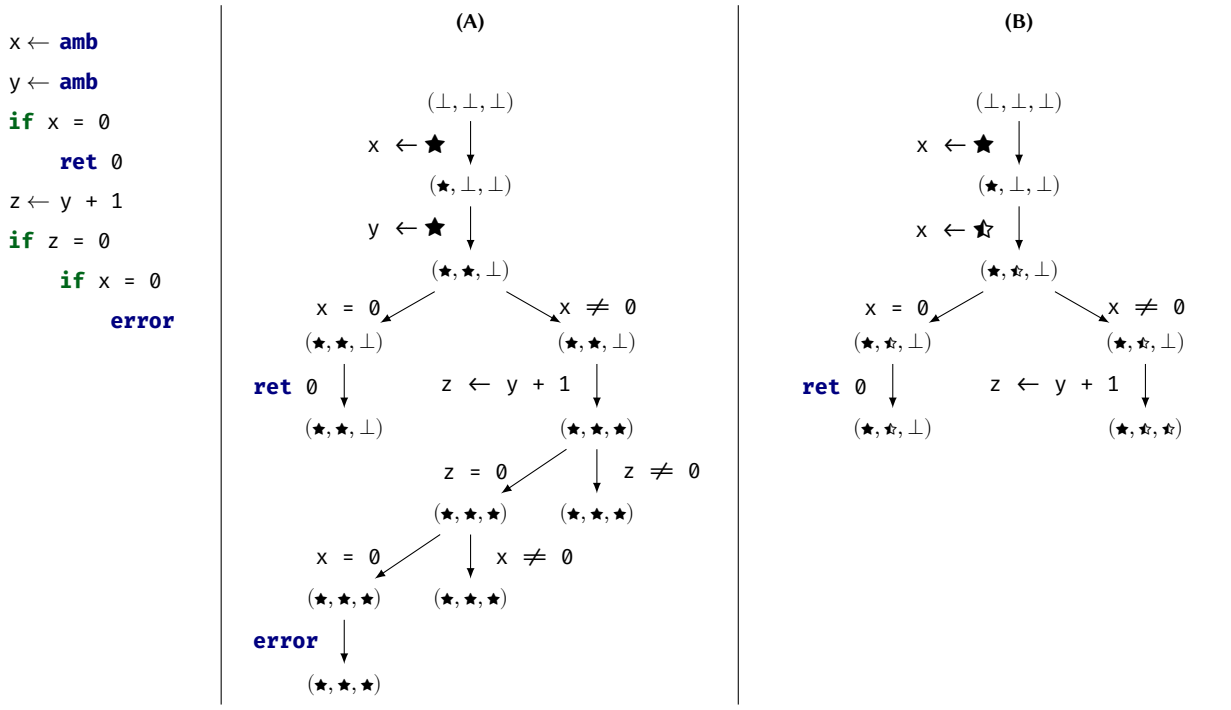


Figure 4.6: Example of unit domain computations for program on the left. Examine two abstractions, the first (A) abstracts both ambiguous values as ★ values, the second abstract x as ★, and y as slicing value ☆. We represent program states as a cartesian product of values x, y, z. The first noteworthy thing is that the unit domain is non-relational. Therefore, even when we decide that `x ≠ 0` on the first branch, we still reach an error behind the second condition `x = 0`. In the second abstraction (B), one can observe how the slicing domain terminates the execution on condition when z is ☆. Nevertheless, arithmetic operations using ☆ continue to be carried out. Note that for simplicity, we abbreviated the branch computation process. This actually includes computing the branch predicate, converting it to a tristate domain, and then concretizing it to a boolean value. Only then do we execute the conditional jump.

Despite its simplicity, the unit domain can be a versatile tool for program analysis. Abstracting to it eliminates variables from the program, allowing us to explore all program paths regardless of the variable value since, when used in conditional statements, both outcomes will be considered. However, when working with variable-size objects, extra caution is necessary. To address this, we can abstract the content of arrays or objects in the same way so that any access to these objects at any index returns a ★ value. Additionally, signaling an out-of-bounds error on every access to these arrays is necessary to ensure the abstraction overapproximates the concrete program.

Further attention needs to be paid to concurrency. In concurrent applications, loads and stores must not be eliminated as a result of this abstraction, and for this reason it may be necessary to mark loads/stores of (possibly) abstract values as volatile. Otherwise, we might mask possibly visible effects, and therefore change the program behavior.

On the contrary, if our goal is to eliminate dependence on a specific variable completely, we can augment the $\mathcal{A}_\bigstar$ domain to terminate on conditional branching on a ★ value. This domain is referred to as a *slicing* unit and denoted as ☆. We realize the termination in the domain conversion to the tristate domain by using an `exit` statement instead of conversion. However, this approach results in underapproximation since it conceals possible program behaviors. This abstraction is analogous to dynamic program slicing, where we can designate unimportant variables, and the abstraction will exclude their dependent computations (refer to Figure 4.6).

As we will demonstrate in Chapter 9, it is possible to perform slicing also during the static analysis phase by optimizing the abstracted program linked with the unit domain through the compiler. In compilation, the compiler performs constant folding on unit values, which allows for efficient slicing. This optimization technique is explored further in our case study on abstraction optimization.

4.3.2 Sign Domain

A sign domain $\mathcal{A}_\pm$ is a bit more complex non-relational value domain that reasons about value signedness properties. We distinguish positive value `>0`, negative value `<0`, zero value `0`, arbitrary value $\top_\pm$, or undefined value $\bot_\pm$. This domain enables the detection of potential division by zero errors or violations of function contracts that mandate positive or negative values.

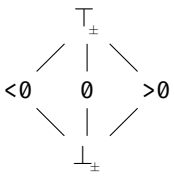


Figure 4.7: A Hasse diagram for a simple *sign domain* $\mathcal{A}^\pm$.

Definition 4.3.2 *Formally, a sign domain* $\mathcal{A}^\pm$ *is defined as:*

$$(\{\bot_\pm, \texttt{0}, \texttt{>0}, \texttt{<0}, \top_\pm\}, \sqsubseteq_\pm, \gamma_\pm, \alpha_\pm, \bot_\pm, \top_\pm, \tau_\pm, \chi_\pm, \cup_\pm, \cap_\pm)$$

The Hasse diagram for the sign domain lattice is presented in Figure 4.7. The diagram implicitly defines abstract order $\sqsubset_\pm$*, the lattice operations join* $\cup_\pm$ *and the meet* $\cap_\pm$*.*

The domain is equipped with Galois connection $\mathcal{C} \underset{\alpha_\pm}{\overset{\gamma_\pm}{\leftrightarrows}} \mathcal{A}_\pm$ *defined in Figure 4.8. We give conversion into tristate domain in Figure 4.9. Additionally,* $\tau_\pm$ *defines the set of abstract transformers, which will be covered in depth in subsequent text.*

$$\alpha_\pm(c) \triangleq \begin{cases} \texttt{<0} & \text{if } c < 0 \\ \texttt{0} & \text{if } c = 0 \\ \texttt{>0} & \text{if } c > 0 \end{cases}$$

$$\begin{aligned} \gamma_\pm(\bot_\pm) &\triangleq \emptyset \\ \gamma_\pm(\texttt{>0}) &\triangleq \{n \mid n > 0\} \\ \gamma_\pm(\texttt{0}) &\triangleq \{0\} \\ \gamma_\pm(\texttt{<0}) &\triangleq \{n \mid n > 0\} \\ \gamma_\pm(\top_\pm) &\triangleq \mathcal{C} \end{aligned}$$

Figure 4.8: Sign domain $\mathcal{A}^\pm$ abstraction & concretization. Note that in implementation, we define specific functions for each bitwidth and type.

Abstract union and intersection are given by lattice `lub` $\cup_\pm \overset{\text{def}}{=} \sqcup_\pm$ and `glb` $\cap_\pm \overset{\text{def}}{=} \sqcap_\pm$. Due to galois connection, it is given that $\cap_\pm$ is exact. The $\cup_\pm$ is also exact, which is actually quite rare – the union of the sets represented by two abstract elements is seldom exactly representable.

$$\chi_{\pm\to\texttt{Tr}}(\hat{v}) \triangleq \begin{cases} \texttt{T} & \text{if } \hat{v} \in \{\texttt{>0}, \texttt{<0}\} \\ \texttt{F} & \text{if } \hat{v} = \texttt{0} \\ \texttt{M} & \text{if } \hat{v} = \top_\pm \end{cases}$$

Figure 4.9: To-tristate conversion from sign domain. Note it is not complete, as we do not define computation for $\bot$ values.

One common approach to defining abstract operations is through the use of table schemas, which outline abstract transfer functions. Specifically, for binary operations $\circ$, a two-dimensional table is employed, where the value $v_\pm$ at the intersection of row $r_\pm \in \mathcal{A}^\pm$ and column $c_\pm \in \mathcal{A}^\pm$ represents the transfer $r_\pm \circ c_\pm = v_\pm$. Luckily, the sign abstraction being perfect, the transfer functions can be derived almost automatically from the definition, i.e., $\hat{a}_1 \circ \hat{a}_2 = \alpha_\pm(\gamma_\pm(\hat{a}_1) \circ \gamma_\pm(\hat{a}_1))$. We describe few interesting transformers in Figure 4.10.

+	$\bot_\pm$	>0	0	<0	$\top_\pm$
$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$
>0	$\bot_\pm$	>0	>0	$\top_\pm$	$\top_\pm$
0	$\bot_\pm$	>0	0	<0	$\top_\pm$
<0	$\bot_\pm$	$\top_\pm$	<0	<0	$\top_\pm$
$\top_\pm$	$\bot_\pm$	$\top_\pm$	$\top_\pm$	$\top_\pm$	$\top_\pm$

-	$\bot_\pm$	>0	0	<0	$\top_\pm$
$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$
>0	$\bot_\pm$	$\top_\pm$	>0	>0	$\top_\pm$
0	$\bot_\pm$	>0	0	<0	$\top_\pm$
<0	$\bot_\pm$	<0	<0	$\top_\pm$	$\top_\pm$
$\top_\pm$	$\bot_\pm$	$\top_\pm$	$\top_\pm$	$\top_\pm$	$\top_\pm$

*	$\bot_\pm$	>0	0	<0	$\top_\pm$
$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$
>0	$\bot_\pm$	>0	0	<0	$\top_\pm$
0	$\bot_\pm$	0	0	0	0
<0	$\bot_\pm$	<0	0	>0	$\top_\pm$
$\top_\pm$	$\bot_\pm$	$\top_\pm$	0	$\top_\pm$	$\top_\pm$

/	$\bot_\pm$	>0	0	<0	$\top_\pm$
$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$
>0	$\bot_\pm$	>0	$\bot_\pm$	<0	$\top_\pm$
0	$\bot_\pm$	0	$\bot_\pm$	0	$\top_\pm$
<0	$\bot_\pm$	<0	$\bot_\pm$	>0	$\top_\pm$
$\top_\pm$	$\bot_\pm$	$\top_\pm$	$\bot_\pm$	$\top_\pm$	$\top_\pm$

%	$\bot_\pm$	>0	0	<0	$\top_\pm$
$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$
>0	$\bot_\pm$	$\top_\pm$	$\bot_\pm$	$\top_\pm$	$\top_\pm$
0	$\bot_\pm$	0	$\bot_\pm$	0	$\top_\pm$
<0	$\bot_\pm$	$\top_\pm$	$\bot_\pm$	$\top_\pm$	$\top_\pm$
$\top_\pm$	$\bot_\pm$	$\top_\pm$	$\bot_\pm$	$\top_\pm$	$\top_\pm$

<<	$\bot_\pm$	>0	0	<0	$\top_\pm$
$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$
>0	$\bot_\pm$	$\top_\pm$	>0	$\top_\pm$	$\top_\pm$
0	$\bot_\pm$	0	0	0	$\top_\pm$
<0	$\bot_\pm$	$\top_\pm$	<0	$\top_\pm$	$\top_\pm$
$\top_\pm$	$\bot_\pm$	$\top_\pm$	$\top_\pm$	$\top_\pm$	$\top_\pm$

=	$\bot_\pm$	>0	0	<0	$\top_\pm$
$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$
>0	$\bot_\pm$	$\top_\pm$	0	0	$\top_\pm$
0	$\bot_\pm$	0	>0	0	$\top_\pm$
<0	$\bot_\pm$	0	0	$\top_\pm$	$\top_\pm$
$\top_\pm$	$\bot_\pm$	$\bot_\pm$	$\top_\pm$	$\top_\pm$	$\top_\pm$

≠	$\bot_\pm$	>0	0	<0	$\top_\pm$
$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$
>0	$\bot_\pm$	$\top_\pm$	>0	>0	$\top_\pm$
0	$\bot_\pm$	>0	0	>0	$\top_\pm$
<0	$\bot_\pm$	>0	>0	$\top_\pm$	$\top_\pm$
$\top_\pm$	$\bot_\pm$	$\top_\pm$	$\top_\pm$	$\top_\pm$	$\top_\pm$

<	$\bot_\pm$	>0	0	<0	$\top_\pm$
$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$	$\bot_\pm$
>0	$\bot_\pm$	$\top_\pm$	0	0	$\top_\pm$
0	$\bot_\pm$	>0	0	0	$\top_\pm$
<0	$\bot_\pm$	0	>0	$\top_\pm$	$\top_\pm$
$\top_\pm$	$\bot_\pm$	$\top_\pm$	$\top_\pm$	$\top_\pm$	$\top_\pm$

Figure 4.10: An example of sign domain abstract operation table schemas. The definition of operation schemas is contingent on whether overflows are considered. However, given the high level of imprecision in the sign domain with overflows, we demonstrate the abstraction in an underapproximative domain that accounts for unbounded values.

4.3.3 Powerset Domain

Maintaining fine-grained information is often desirable in abstractions. To achieve this, one can compute in a powerset domain [FR99], where operations are performed over a set of abstract values instead of elementary values. For example, `(<0) * (>0)` would yield the $\top_\pm$ value in the elementary sign domain, whereas, in the powerset sign domain, the result would be $\{$`<0`, `>0`$\}$, excluding zero from the result. This is crucial in reducing false positive cases.

[FR99]: Filé et al. (1999), "The powerset operator on abstract interpretations"

The powerset domain is not a domain by itself but rather a domain adaptor, also known as a *functor domain*, which transforms an elementary domain to perform computations on sets of its elementary values. To denote the powerset domain created from the elementary value domain $\mathcal{A}$, we will write $\wp(\mathcal{A})$. The concept behind the powerset domain is straightforward. Given an abstract domain $\mathcal{A}$, any subset S of $\mathcal{A}$ represents the union of its elements in the concrete domain $\mathcal{C}$.

Functor domains generally serve as templates for creating domains. They are parametrized by other domains, and their operations are defined in terms of the operations from these domains. For instance, the powerset domain defines its operations on set elements using the operations of the parameter domain.

Example 4.3.1 Consider $\hat{S}_1, \hat{S}_2 \in \wp(\mathcal{A})$. The addition in $\wp(\mathcal{A})$ is defined as the elementary addition from $\mathcal{A}$ on all elements of the sets:

$$\hat{S}_1 \mathbin{+_{\wp(\mathcal{A})}} \hat{S}_2 \triangleq \{\hat{v}_1 \mathbin{+_{\mathcal{A}}} \hat{v}_2 \mid \hat{v}_1 \in \hat{S}_1 \wedge \hat{v}_2 \in \hat{S}_2\}$$

Given an abstract domain $\mathcal{A}$ with functions $\alpha_{\mathcal{A}}$, $\gamma_{\mathcal{A}}$, and set of transformers $\tau_{\mathcal{A}}$, we will now construct a general powerset domain $\wp(\mathcal{A})$ without compromising the soundness of the elementary abstract domain $\mathcal{A}$ and being at least as precise as the domain $\mathcal{A}$.

Definition 4.3.3 *Elements of powerset domain $\wp(\mathcal{A})$ are defined as:*

- $\bot_{\wp(\mathcal{A})} \triangleq \emptyset$ *and* $\top_{\wp(\mathcal{A})} \triangleq \mathcal{A}$
- *A concretization of $\hat{S} \in \wp(\mathcal{A})$ is defined as:*

$$\gamma_{\wp(\mathcal{A})}(\hat{S}) \triangleq \bigcup \{\gamma_{\mathcal{A}}(\hat{v}) \mid \hat{v} \in \hat{S}\}$$

- *An abstraction of $v \in \mathcal{C}$ is defined as singleton set:*

$$\alpha_{\wp(\mathcal{A})}(v) \triangleq \{\alpha_{\mathcal{A}}(v)\}$$

- *Binary operations $\circ : \wp(\mathcal{A}) \times \wp(\mathcal{A}) \to \wp(\mathcal{A})$ are induced from $\circ_{\mathcal{A}} : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ as:*

$$\hat{S}_1 \mathbin{\circ_{\wp(\mathcal{A})}} \hat{S}_2 \triangleq \{\hat{v}_1 \mathbin{\circ_{\mathcal{A}}} \hat{v}_2 \mid \hat{v}_1 \in \hat{S}_1 \wedge \hat{v}_2 \in \hat{S}_2\}$$

- *Likewise unary operations and casts as $\hat{f}_{\wp(\mathcal{A})} : \wp(\mathcal{A}) \to \wp(\mathcal{A})$ are induced from $\hat{f}_{\mathcal{A}} : \mathcal{A} \to \mathcal{A}$ as:*

$$\hat{f}_{\wp(\mathcal{A})}(\hat{S}) \triangleq \{\hat{f}_{\mathcal{A}}(\hat{v}) \mid \hat{v} \in \hat{S}\}$$

- *Tristate conversion is defined as a join over all constituent tristate conversions:*

$$\chi_{\wp(\mathcal{A}) \to \mathtt{Tr}}(\hat{S}) \triangleq \bigsqcup \{\chi_{\mathcal{A} \to \mathtt{Tr}}(\hat{v}) \mid \hat{v} \in \hat{S}\}$$

For a comprehensive understanding of the powerset domain, refer to the foundational publication [FR99]. In this thesis, the powerset domain serves as an example of a functor domain to introduce the concept. To showcase its usefulness, constructing a powerset domain $\wp(\mathcal{A}_{\pm})$ from the sign domain results in the domain illustrated in Figure 4.11. This transformation enables us to consider non-strict inequalities by grouping `0` with either positive or negative values. Additionally, it gives us the capability to express all non-null values as $\{\mathtt{<0}, \mathtt{>0}\}$.

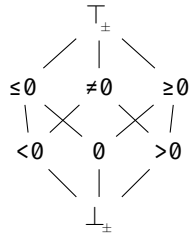


Figure 4.11: A Hasse diagram for $\wp(\mathcal{A}_{\pm})$ domain with strict inequalities. For brevity, we denote elements: $\{\mathtt{>0}, \mathtt{<0}\}$ as $\mathtt{\neq 0}$, $\{\mathtt{>0}, \mathtt{0}\}$ as $\mathtt{\geq 0}$, and $\{\mathtt{<0}, \mathtt{0}\}$ as $\mathtt{\leq 0}$.

4.3.4 Product Domain

As one can observe, various domains can reason about diverse program properties, which yields different precision of the analysis. Abstraction through multiple domains can effectively address imprecision. Instead

of performing program analysis with each domain individually, it is advantageous to use multiple domains simultaneously. This not only saves time by avoiding repetitive analysis but also allows the domains to support each other in eliminating unreachable executions and reducing false positives.

A commonly used approach to multi-domain analysis is the direct product of two domains, $\mathcal{A}$ and $\mathcal{B}$, represented as $\mathcal{A} \times \mathcal{B}$. This is a set of all ordered pairs $\langle a, b \rangle$ where a belongs to $\mathcal{A}$ and b belongs to $\mathcal{B}$. One can see that the direct product is a functor domain parametrized by its element domains. This idea can be easily extended to include more than two domains.

Note the difference to the cartesian product of domains, which is used to represent multiple values in an abstract state. In contrast, the product domain expresses properties of a single value in multiple domains.

The operations in the product domain are derived from the constituent domains and applied element-wise, as given in the following definition.

Definition 4.3.4 *Formally, all arithmetic and relational operations on a product domain $\mathcal{A} \times \mathcal{B}$ are performed element-wise:*

$$\langle a_1, b_1 \rangle \circ \langle a_2, b_2 \rangle \stackrel{\text{def}}{=} \langle a_1 \circ a_2, b_1 \circ b_2 \rangle$$

When it comes to operations that involve the concrete domain, special consideration needs to be taken. It's crucial to specify how to concretize pairs of values or which value to use to direct control-flow branching. A set intersection of two concretizations $\gamma_{\mathcal{A}}(a) \cap \gamma_{\mathcal{B}}(b)$ can be employed, but it is often too restrictive or difficult to compute. Sometimes, one domain in the product can only be supplementary for certain operations, so we do not want to treat it always in cost-demanding operations such as concretization. To address this, we can parametrize the product domain with a strategy for handling domain interactions.

In practice, a more accurate variation of the direct product, known as *reduced product*, is commonly used [Cou+07; TCR13]. This is because it removes duplicit domain elements, making the domain smaller and more efficient. In this thesis, we will suffice with the simple direct product.

[Cou+07]: Cousot et al. (2007), "Combination of Abstractions in the ASTRÉE Static Analyzer"
[TCR13]: Toubhans et al. (2013), "Reduced Product Combination of Abstract Domains for Shapes"

4.3.5 Interval Domain

The interval abstraction, denoted by $\mathcal{A}_i$, is a way to express the smallest and largest values that a variable (register value) can take. It summarizes a range of values as a lower bound ($\underline{v}$), and an upper bound ($\overline{v}$). If the bounds are ambiguous, they are considered to be infinite, which is represented as the lowest, respectively greatest, machine-representable integer r of a given bitwidth. This representation, however, can not reason about overflow errors and may not soundly reflect overflow arithmetic. Some adaptations of the interval domain take into account wrapped arithmetic, such as [Nav+12]. However, for our demonstration of domains for compilation-based abstraction is sufficient to use a more basic form of the interval domain, as it can easily be expanded to more advanced

[Nav+12]: Navas et al. (2012), "Signedness-Agnostic Program Analysis: Precise Integer Bounds for Low-Level Code"

forms later on. Despite this, interval abstraction is useful because it provides a compact representation and efficient constant-time operations for arithmetic and relational calculations.

[Cou21]: Cousot (2021), *Principles of Abstract Interpretation*

We define the interval domain as in [Cou21], the values of the interval domain are:

$$\begin{aligned}\mathcal{A}_i \triangleq \{\emptyset\} \cup \{[\underline{v}, \overline{v}] \mid \underline{v}, \overline{v} \in \mathcal{C} \wedge \underline{v} \leq \overline{v}\} \\ \cup \{[-\infty, \overline{v}] \mid \overline{v} \in \mathcal{C}\} \cup \{[\underline{v}, \infty] \mid \underline{v} \in \mathcal{C}\} \cup \{[\infty, \infty]\}\end{aligned}$$

where an empty interval $\bot_i = \emptyset$ is encoded by any invalid interval $[\underline{v}, \overline{v}]$ with $\overline{v} < \underline{v}$. Intervals are accompanied by a partial order $\sqsubseteq_i$ on $\mathcal{A}_i$ defined as inclusion:

$$\emptyset \sqsubseteq_i [\underline{a}, \overline{a}] \sqsubseteq_i [\underline{b}, \overline{b}] \text{ iff } \underline{b} \leq \underline{a} \leq \overline{a} \leq \overline{b}$$

For that, the maximal interval is $\top_i = [-\infty, \infty]$. Notice we do not allow, $[-\infty, -\infty]$ and $[\infty, \infty]$. In the computation with infinities we presume, $\infty \circ \infty = \infty$, and **`cast`** ∞ **`to`** `ty` $= \infty$, similarly for $-\infty$. All operations with $\bot_i$ result in $\bot_i$ as the computation with an empty set are not defined. Converting between the abstract, concrete and tristate domain is given as:

$$\alpha_i(V) \triangleq [\mathit{min}(V), \mathit{max}(V)]$$
$$\gamma_i([\underline{v}, \overline{v}]) \triangleq \{v \in \mathcal{C} \mid \underline{v} \leq v \leq \overline{v}\}$$
$$\chi_{i \to \mathtt{Tr}}([\underline{v}, \overline{v}]) \triangleq \begin{cases} \mathtt{T} & \text{if } \overline{v} < 0 \vee \underline{v} > 0 \\ \mathtt{F} & \text{if } \underline{v} = 0 = \overline{v} \\ \mathtt{M} & \text{otherwise} \end{cases}$$

The effects of interval operations are summarized in Figure 4.12.

In Figure 4.12, we provide only a definition for the relational operation `<`. The remaining relational operations can be defined in a similar manner. During abstract execution, if a bottom value is obtained, the process terminates. Thus, there is no need to specify the implementation of relational operations on $\bot_i$.

Similarly to other domains, we let the underlying interpreter yield an error in cases like division by zero. For simplicity, the semantics have been kept as concise as possible, despite the availability of more accurate implementations. For example, a more precise implementation is proposed in [HJV01]. These checks can always be performed as an additional assertion, such as adding an assertion for non-zero values before division.

[HJV01]: Hickey et al. (2001), "Interval Arithmetic: From Principles to Implementation"

In the context of compilation-based abstraction, the interval domain is a noteworthy domain due to its simple scalar representation. This property enables a compiler to perform easily constant folding of interval computations, what simplifies the subsequent dynamic analysis.

Moreover, we have implemented the interval domain as a functor domain that is parametrized by the domain of bounds. This design enables us to specialize the bounds to be bit-precise or even relational, using relational-bound domain, such as terms presented later in this chapter. This is similar to approach presented in [Sha17].

[Sha17]: Sharma (2017), *Abstract Interpretation Over Bitvectors*

Note that the implementation of the interval domain and other domains can vary in their approach to error handling. For example, two viable

Operation	Effect on $\widehat{\sigma}$	
$\mathtt{r_s} \longleftarrow [\underline{a}, \overline{a}]$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto [\underline{a}, \overline{a}]]$	
$\mathtt{r_s} \longleftarrow$ **amb**	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto [-\infty, \infty]]$	
$\mathtt{r_s} \longleftarrow$ **amb**$(v_1, \ldots, v_n)$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \alpha_i(\{v_1, \ldots, v_n\})]$	
$\mathtt{r_s} \longleftarrow \bot_i \circ [\underline{b}, \overline{b}]$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \bot_i]$	
$\mathtt{r_s} \longleftarrow [\underline{a}, \overline{a}] \circ \bot_i$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \bot_i]$	
$\mathtt{r_s} \longleftarrow [\underline{a}, \overline{a}]$ + $[\underline{b}, \overline{b}]$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto [\underline{a} + \underline{b}, \overline{a} + \overline{b}]]$	
$\mathtt{r_s} \longleftarrow [\underline{a}, \overline{a}]$ - $[\underline{b}, \overline{b}]$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto [\underline{a} - \overline{b}, \overline{a} - \underline{b}]]$	
$\mathtt{r_s} \longleftarrow [\underline{a}, \overline{a}] * [\underline{b}, \overline{b}]$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto [min(V), max(V)]]$	
	where $V = \{\underline{a} * \underline{b}, \underline{a} * \overline{b}, \overline{a} * \underline{b}, \overline{a} * \overline{b}\}$	
$\mathtt{r_s} \longleftarrow [\underline{a}, \overline{a}]$ / $[\underline{b}, \overline{b}]$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto [\underline{a}, \overline{a}] * [1/\overline{b}, 1/\underline{b}]$ provided $0 \notin [\underline{b}, \overline{b}]$	
$\mathtt{r_s} \longleftarrow [\underline{a}, \overline{a}]$ % $[\underline{b}, \overline{b}]$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto [min(V), max(V)]]$	
	where $V = \{\underline{a} \,\%\, \underline{b}, \underline{a} \,\%\, \overline{b}, \overline{a} \,\%\, \underline{b}, \overline{a} \,\%\, \overline{b}\}$	
$\mathtt{r_s} \longleftarrow$ **cast** $\bot_i$ **to** ty	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \bot_i]$	
$\mathtt{r_s} \longleftarrow$ **cast** $[\underline{a}, \overline{a}]$ **to** ty	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto [min(V), max(V)]]$	
	where $V = \{$**cast** $\underline{a}$ **to** ty, **cast** $\overline{a}$ **to** ty$\}$	
$\mathtt{r_b} \longleftarrow [\underline{a}, \overline{a}] = [\underline{b}, \overline{b}]$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto [1, 1]]$	if $\underline{a} = \overline{a} = \underline{b} = \overline{b}$
	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto [0, 0]]$	if $\overline{a} < \underline{b} \vee \overline{b} < \overline{a}$
	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto [0, 1]]$	otherwise
$\mathtt{r_b} \longleftarrow [\underline{a}, \overline{a}] \neq [\underline{b}, \overline{b}]$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \neg([\underline{a}, \overline{a}] = [\underline{b}, \overline{b}])$	
$\mathtt{r_b} \longleftarrow [\underline{a}, \overline{a}] < [\underline{b}, \overline{b}]$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto [1, 1]]$	if $\overline{a} < \underline{b}$
	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto [0, 0]]$	if $\underline{a} \geq \overline{b}$
	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto [0, 1]]$	otherwise

Figure 4.12: Interval domain step semantics, where $\circ$ are arithmetic operations, and $\diamond$ relational operations as described in Definition 3.2.1. Note that this description performs only abstract computation, for mixed computation, we first abstract all arguments to the interval domain.

domain implementations may consider integer overflow arithmetic or use unbounded abstractions. Similarly, in the implementation, we can make other optimization-driven design choices. For instance, when a value is undefined in abstract execution, i.e., $\bot$, we do not need to continue in the interpretation, but the run can be canceled. This allows for optimization in the representation of values and their operations, such choices also influence syntactic abstraction, as compiler can leverage the fact we terminate on $\bot$ value.

4.3.6 Term Domain

In program analysis, it is advantageous to have a clear understanding of relationships and data dependencies between program values. To achieve this, a common approach is to use logical formalisms. One such representation is ground terms, first introduced by Jacques Herbrand [Her30]. Ground terms can describe mathematical objects such as constants, as well as symbolic terms to represent variables and function symbols to describe operations. The models of these symbolic terms describe sets of values, and it has been proven that they form a complete lattice with a partial order $\sqsubseteq_{\mathcal{T}}$ with a meaning of *less general* [Plo71; Rey70]. For example, the following terms are ordered by their generality:

$$op(1, 2) \sqsubseteq_{\mathcal{T}} op(a, 2) \sqsubseteq_{\mathcal{T}} op(a, b)$$

[Her30]: Herbrand (1930), "Recherches sur la théorie de la démonstration"

[Plo71]: Plotkin (1971), "A further note on inductive generalization"
[Rey70]: Reynolds (1970), "Transformational systems and algebraic structure of atomic formulas"

[Gan+16]: Gange et al. (2016), "An Abstract Domain of Uninterpreted Functions"

Terms. The set of terms $\mathcal{T}$, is defined recursively [Gan+16]. Every term t is either a variable $v \in \mathit{vars}_{\mathcal{T}}$ or an application of a function symbol $f(t_1, \dots, t_n)$, where $f \in \mathcal{T}_F$ is a function symbol with arity $n \geq 0$ and $t_1, \dots, t_n$ are terms.

A substitution is a mapping, ϱ, from variables $\mathcal{T}_V$ to terms $\mathcal{T}$. This mapping is naturally extended to $\mathcal{T} \to \mathcal{T}$. We use standard notation for substitutions, such as $[x \mapsto t]$ representing the substitution ϱ where $\varrho(x) = t$ and $\varrho(v) = v$ for all $v \neq x$. An instance of a term t is any term $\varrho(t)$. We say that term $t \sqsubseteq_{\mathcal{T}} t'$ if there exists a substitution ϱ such that $t = \varrho(t')$. Moreover, $t \equiv t'$ iff $t \sqsubseteq_{\mathcal{T}} t \wedge t' \sqsubseteq_{\mathcal{T}} t$.

Notably, two terms t and s are structuraly equivalent:

- if t and s are identical constants,
- if $t = f(t_1, \dots, t_n)$ and $s = f(s_1, \dots, s_n)$, and for all i, $t_i \equiv s_i$.

Symbolic terms, in combination with a decision procedure, usually a dedicated SMT solver, are broadly adopted by tools implementing theorem proving, symbolic execution, or bounded model checking. As a most basic instance, for abstract interpretation, we can consider a domain of uninterpreted functions, also called subterm domain [Gan+16].

We adopt the definition of subterm domain from [Gan+16], which matches our implementation of term abstraction. Values in the subterm domain are given as mapping from program values (registers and memory content) to terms. The structure of this domain is based on uninterpreted functions. However, to make a meaningful analysis of concrete computations, each function symbol f must be assigned a semantic function, denoted as $S(f)$, that maps an n-tuple of scalars to a scalar value. Given an assignment, denoted as ϱ, which maps term variables to scalar values, the evaluation of a term under ϱ can be recursively defined as $\mathit{eval}_{\mathcal{T}}(t, \varrho)$.

$$\mathit{eval}_{\mathcal{T}}(x, \varrho) \stackrel{\text{def}}{=} \varrho(x)$$
$$\mathit{eval}_{\mathcal{T}}(f(t_1, \dots, t_n), \varrho) \stackrel{\text{def}}{=} S(f)(\mathit{eval}_{\mathcal{T}}(t_1), \dots, \mathit{eval}_{\mathcal{T}}(t_n))$$

The question now arises whether this domain is relational and exact. This depends on the interpretation of free variables: first consider a single global assignment of values to free variables for the entire program – that is, variables are represented as a single term, which has structure of a tuple, and each variable corresponds to an element of this tuple. In this case, the answer to both is in the affirmative: the term domain is exact and relational. However, if the assignment is done separately – each variable is represented by its own independent term – then it is neither. We will only consider the former (exact) case.

For instance, consider state with two symbolic variables:

$\varphi(x, y) \equiv x \geq 0 \wedge y = x + 1 \wedge y < 10.$

The set of states $[\![\varphi]\!]$ contains all pairs that satisfy φ:

$[\![\varphi]\!] = \{(x, y) \mid x \leq 0 \wedge 1 \leq y < 10\}.$

Figure 4.13: An example of symbolic state.

In fact, the term domain encodes concrete semantics. If we let term variables obtain values from the concrete domain and give concrete semantics to function symbols, we obtain a description of a set of states in the form of a logic formula. We say that $[\![\varphi]\!]$ is a set of states that satisfies the formula φ. These are all substitutions of variables for constants that satisfy the formula φ (see Figure 4.13). Notably, we

can change the domain of term variables and the precision of function symbols, which results in the abstraction of the set of states.

It is important to note that in our abstraction, the formula in the term domain does not represent the whole program state but rather only the abstract part. To generalize the notion of symbolic states that satisfies $[\![\varphi]\!]$, we can think of the concrete part of the state as a fixed part, where each concrete register and memory content is assigned a constant so that each state in $[\![\varphi]\!]$ has the same concrete part.

Our implementation of the term domain in LART serves the purpose of performing symbolic computation. The term domain allows the program to construct terms during its execution. Initially, these terms lack meaning, and it is the responsibility of the analysis tool or runtime to assign meaning to them during evaluation. To facilitate this, we have defined a set of common function symbols for arithmetic and relational operations, e.t.c., that are allowed to appear in analyzed programs and can be used to build terms.

Programs usually work with bit strings of finite length that represent numbers rather than whole numbers, as commonly encountered in mathematical practice. As a result, conventional logical formalisms, such as first-order logic with integers, cannot be soundly utilized in this context. The appropriate formalism to reflect the behavior of binary-represented finite numbers is the logic of fixed-size bit-vectors. We leverage their semantics to assign meaning to function symbols within the term domain. Often, function symbols from bit-vector theory correspond to instructions in LLVM IR or require only minor adaptations, such as encoding of implicit conversions between booleans and bit-vector values.

In the following, we will briefly introduce the theory of fixed-size bit-vectors. For a more comprehensive description, the reader is referred to the work by Hadarean monography [Had15].

[Had15]: Hadarean (2015), "An efficient and trustworthy theory solver for bit-vectors in satisfiability modulo theories"

The fixed-size bit-vector theory is a many-sorted first-order theory that consists of an infinite number of sorts denoted as $[n]$ representing bit-vectors of length n. It defines common arithmetic, bitwise and relational function symbols with natural semantics. For example:

$$(x^{[32]} - y^{[32]} = (z^{[16]} \circ 0^{[16]}) \ll 1^{[32]}) \wedge x^{[32]} \neq y^{[32]}$$

consists of bit-vector variables x and y having a bit-width of 32 bits, and z having a bit-width of 16 bits. The formula also incorporates subtraction, concatenation, bitshift operations, as well as comparisons. In practice, we will employ a more verbose definition from SMT-LIB standard, summarized in Table 4.1. In abstraction, we have the option to either compute terms without assigning meaning to them right away or construct terms directly in a particular theory, like the theory of bit-vectors.

One of our primary goals for the abstraction, was to allow the program to perform symbolic analysis. Now that we have described a suitable term representation, the remaining task is to re-cast symbolic computation as an abstract domain. Fortunately, this is not very hard: the abstract values in the domain are terms, while the abstract instructions simply construct corresponding terms from their operands. In other words, symbolic computation is realized by free algebra (that is, the term algebra).

Table 4.1: Function and predicate symbols of the bit-vector, adapted from [Jon19].

Symbol	Sort	Semantics description
`bX`	$[m]$	bitvector constant `X`
`x`	$[m]$	bitvector variable
`bvnot`	$[n] \to [n]$	bitwise negation
`bvneg`		arithmetic negation
`bvand`	$[n] \times [n] \to [n]$	bitwise and
`bvor`		bitwise or
`bvxor`		bitwise exclusive or
`bvadd`	$[n] \times [n] \to [n]$	addition
`bvsub`		subtraction
`bvmul`		multiplication
`bvudiv`, `bvurem`		unsigned division, remainder
`bvsdiv`, `bvsrem`		signed division, remainder
`bvshl`	$[n] \times [n] \to [n]$	logical shift left
`bvlshr`		logical shift right
`bvashr`		arithmetic shift right
`bveq`, `bvne`	$[n] \times [n] \to \mathit{Bool}$	equality, inequality
`bvule`, `bvult`		unsigned less or equal, less than
`bvsle`, `bvslt`		signed less or equal, less than
`bvuge`, `bvugt`		unsigned greater or equal, greater than
`bvsge`, `bvsgt`		signed greater or equal, greater than
`concat`	$[m] \times [n] \to [m+n]$	bit concatenation
$\texttt{zext}_m$	$[n] \to [n+m]$	zero extension
$\texttt{sext}_m$	$[n] \to [n+m]$	sign extension
$\texttt{extract}_{i,j}$	$[m] \to [i-j+1]$	slice from i-th to j-th bit

The input values of the program correspond to nullary symbols – in practice, a unique nullary symbol is created each time the program obtains a value from its input (`amb`). All the remaining values are built up as terms of applications of function symbols and constants.

It is not hard to see that a program abstracted this way will perform part of its computation symbolically in the usual sense. Furthermore, for complete symbolic analysis, we need to impose constraints on nullary symbols. This is realized on interpretation of guard edges of CFG. Each constraint takes the form of a term with a relational symbol in the root position. These constraints become part of the abstract state, effectively ensuring that the term domain is fully relational. One can perceive these constraints as a path condition induced by program execution.[5] In this case, we expand the definition of the abstract state by adding a new component, π, which describes the constraints collected during the execution of the program along the path – also known as the path condition. The guard edge in the control flow graph serves to conjunct the specified constraint with the path condition π. Moreover, in the analysis we need to check its feasibility.

It is a requirement of abstract interpretation that it is possible to construct an abstract state from a set of concrete states. In the term domain this can be achieved by assigning, to each memory location that differs in some of the concrete states[6], a fresh nullary symbol. We then impose constraints that ensure that exactly the input set of concrete states is

5: An abstract domain is called relational when it is capable of preserving information about relationships among various abstract values that appear in the program.

6: At the moment, we only deal with abstract (symbolic) values. The structure of the program state, that is, the arrangement of the program memory, is taken to be always represented explicitly, i.e., it belongs squarely to the concrete domain.

represented by the resulting abstract state. For instance, if the input set of concrete states differs by the value of a single variable a, and this variable takes values 1, 2, 3 and 4 in the 4 input states, a suitable constraint would be $a \geq 1 \wedge a \leq 4$.

In some cases, it is impossible to construct the requisite constraints using only conjunction and relational operators. To ensure that the term domain forms a lattice (in particular that a least upper bound always exists), it is necessary to allow the constraints to use logical disjunction.

An important aspect of abstraction is its effect on control flow of the program. It is often the case that control flow depends on specific values of variables via conditional branching. The condition on the branch is typically a predicate on some value, or a relationship among multiple values that appear in the program. If the involved values are, in fact, abstract values, it is quite possible that both results of the predicate or comparison are admissible and that the conditional branch can therefore go both ways. The choice of direction provides additional information – constraints – on the possible values of variables (cf. Example 4.3.4).

While the above considerations regarding constraints are an important part of the theoretical underpinnings of the approach, it is almost always entirely impractical to shift back and forth between concrete and abstract states. In practice, therefore, the constraints described in this section simply arise through guard conditions. As such, the constraints that appear in a given state form a *path condition*. Finally, we note that the least upper bound of abstract states defined above corresponds to path conditions which arise from *path merging* in symbolic execution.

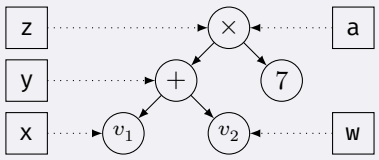
A symbolic representation of data is typically partitioned into two distinct components, namely the path condition of the program and the definitions of data. The path condition is a combination of formulas representing the restriction of data collected during branching along the path leading to the current location. Meanwhile, definitions consist of a set of formulas in the form of "*variable = expression*" that describe the relationships between variables. These definitions are created as a result of assignments or arithmetic instructions. This structured symbolic data representation provides a precise description of what is needed for program analysis but lacks a canonical representation. As a matter of fact, the feasibility and quality of symbolic states cannot be determined purely syntactically. To overcome this, we need to employ a specialized decision procedure, usually an SMT solver, to evaluate the path condition's satisfiability and determine the equality of multiple states [Mrá+16].

[Mrá+16]: Mrázek et al. (2016), "Sym-DIVINE: Tool for Control-Explicit Data-Symbolic State Space Exploration"

In the following examples, we represent terms as tree data structure. The abstract instructions that correspond to operations on values construct a tree representation of the requisite term by joining their operands to a new root node, where only the operation in the root node depends on the specific abstract instruction.

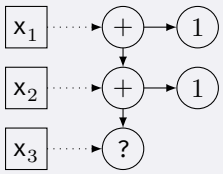
Example 4.3.2 A formula tree as generated by the term domain. The boxes correspond to abstract variables, while the circles are the concrete representation of terms. Nodes with vs denote unconstrained nullary symbols.

```
a ← malloc sizeof(i32)
w ← amb
x ← amb
y ← w ∔̂ x
z ← y ×̂ 7
store z → a
```



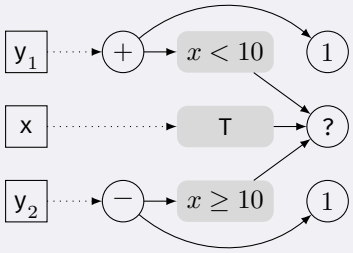
Example 4.3.3 An example of a formula tree arising from a loop. Versions of the variable x which exist in different iterations of the loop are distinguished by an index in the picture.

```
x ← amb
i ← 1
while i < 2
    x ← x ∔̂ 1
    i ← i + 1
```



Example 4.3.4 The formula tree on the right includes constraints arising from branching. Note that on any given path through the program, only one of the subtrees rooted in y_1 or y_2 can exist.

```
x ← amb
if x <̂ 10
    y ← x ∔̂ 1
else
    y ← x ∸̂ 1
```



The primary purpose of the term domain was to facilitate symbolic computation and augment model checking with symbolic capabilities. We will delve deeper into these concepts in Chapter 8. These approaches have been effectively employed, both individually and in combination with other scalar domains, in software verification competition (SV-COMP) for several years, further described in Appendix B, as well as in our own evaluations described in Appendix A.

4.4 Abstraction Refinement

The abstraction-driven analysis goes hand-in-hand with various refinement techniques. In the context of abstract interpretation, refinement is employed to address the convergence issue that arises from the fixpoint-

based approach. Additionally, refinement can help reduce imprecision caused by abstraction composition, enable the use of coarser abstractions, and allow for lazy refinement of the domain to achieve greater precision and scalability in the analysis. In this section, we will discuss a few related techniques regarding our approach. Moreover, we will design a domain-agnostic refinement for abstract execution.

When it comes to fine-tuning the precision of program analysis, we can differentiate between two types of refinement. The first, which is more conventional, involves enhancing the precision of a single-domain computation. This type of refinement is referred to as *intradomain refinement*. On the other hand, techniques that improve precision by leveraging multiple domains are known as interdomain refinements. In this section, we will focus on the former, and we will delve into the latter, interdomain refinement, in more detail in Chapter 7, once we have access to more complex domains.

4.4.1 State of the Art

So far, we have seen abstract domains with various expressiveness, cost, and precision. However, in many cases, we prefer to commence the abstraction with the simplest domains and refine them automatically on the fly when a spurious counterexample occurs. One approach of refinement is based on the observation that abstract domains also form a lattice. Within this lattice structure, we can refine two domains through a meet operation, also recognized as the reduced product construction [CCM11; TCR13]. A comprehensive analysis of large-scale reduced products is further presented in [Cou+07]. Later in the chapter dedicated to interdomain refinement, we will delve more into the subject of domain lattices.

[CCM11]: Cousot et al. (2011), "The Reduced Product of Abstract Domains and the Combination of Decision Procedures"
[TCR13]: Toubhans et al. (2013), "Reduced Product Combination of Abstract Domains for Shapes"

[Cou+07]: Cousot et al. (2007), "Combination of Abstractions in the ASTRÉE Static Analyzer"

Another refinement technique is the *powerset construction* [CC79], which creates a powerset lattice from the abstract domain's elements, also known as the powerset functor domain. The powerset domain offers a more precise abstraction, enabling us to describe values through a subset of properties. For instance, in the *sign* domain, the powerset domain enables us to abstract a value to a subset such as $\{\texttt{0}, \texttt{>0}\}$, which was not previously possible. This approach was further improved in [FR99] and expanded with efficient widening operators [BHZ04] and exact joins of numerical elements [BHZ10].

[CC79]: Cousot et al. (1979), "Systematic Design of Program Analysis Frameworks"
[FR99]: Filé et al. (1999), "The powerset operator on abstract interpretations"
[BHZ04]: Bagnara et al. (2004), "Widening Operators for Powerset Domains"
[BHZ10]: Bagnara et al. (2010), "Exact join detection for convex polyhedra and other numerical abstractions"

A major source of imprecision in abstraction is an approximation that arises from abstract unions. The common use-case of abstract disjunctions (unions) is to combine multiple program paths. However, many abstract domains are not closed under union operation. To address this issue, *disjunctive completion methods* have been studied in [FR99]. The completion process includes all concrete disjunctions of domain elements that were initially absent from the abstraction. The resulting domain is the most comprehensive domain that exactly represents disjunctions of abstract properties in the original domain. Similarly, the technique of *complementation* introduces the missing complements to the abstraction [Cor+95]. Another refinement technique, known as *fixpoint completion*, enhances the precision of abstract domains by incorporating program-specific properties that are necessary for achieving more accurate fixpoint iteration results. This method has been discussed in [GQ01; GRS00].

[FR99]: Filé et al. (1999), "The powerset operator on abstract interpretations"

[Cor+95]: Cortesi et al. (1995), "Complementation in abstract interpretation"
[GQ01]: Giacobazzi et al. (2001), "Incompleteness, Counterexamples, and Refinements in Abstract Model-Checking"
[GRS00]: Giacobazzi et al. (2000), "Making Abstract Interpretations Complete"

Relational abstract domains typically face scalability issues. One way to address this problem and maintain linear cost is to utilize a technique called "variable packing", which involves dividing variables into smaller, independent sets [Min06c]. The effectiveness of the variable packing technique can be further enhanced by preserving relations between the different sets of variables, as suggested in [Bou12].

[Min06c]: Miné (2006), "The octagon abstract domain"

[Bou12]: Bouaziz (2012), "TreeKs: A Functor to Make Numerical Abstract Domains Scalable"

[MJ81]: Muchnick et al. (1981), *Program Flow Analysis: Theory and Application*

State partitioning techniques, introduced in [MJ81], work similarly to variable packing. These techniques use decision-tree data structures based on binary decision diagrams to break program states into smaller parts, allowing for relational reasoning about value properties within each part. State partitioning is frequently utilized in modern analysis tools, for instance, in [Ber+10; Ber+15]. Similarly, there are abstract domains that incorporate decision trees to handle disjunctions in the programs [CCM10] or to prove conditional termination [UM14].

In a pursuit to achieve greater precision, some techniques move away from the traditional lattice-based approach and instead develop weaker formalisms for abstract domains that use the quasi-join operation, as described in [Gan+13]. The relaxed restrictions on domain definition enable the creation of new domains, such as non-convex numeric domains, that increase precision without sacrificing efficiency. We pay for this by losing certain properties of abstraction, such as the monotonicity or associativity of join operations, that we need to consider in applied analysis.

[Gan+13]: Gange et al. (2013), "Abstract Interpretation over Non-lattice Abstract Domains"

4.4.2 Intradomain Refinement

In abstract program execution, we often observe loss of precision which arises from non-relational nature of domains in use, and also to certain degree on weakly-relational domains if they can not capture specific relations. For instance, whenever we make a branch decision on an abstract value we inherently overapproximate values, as the relational operation only computes the comparison result, not constraining its operands.

Let us consider the interval domain as an abstraction of values from Figure 4.14. When we encounter a conditional statement like `w < z`, which can result in an ambiguous *maybe* value, the operands are not constrained in a straightforward execution if we make a branch decision based on this value. This lack of constraints can lead to unnecessary overapproximations and, in turn, false alarms.

```
x ← amb[1, 2]
y ← amb[0, 1]
z ← amb[2, 5]
w ← x + y
if w < z
    a ← x + 1
```

Figure 4.14: Example of unconstrained values.

Cousot's fixed-point approach constrain variables if a specific path is taken. However, this approach only considers two specific variables from the guard condition, which is often insufficient in the case of LLVM, where SSA values often form long chains of dependencies and can be used in many places. For instance, in the Figure 4.14, the sole constraint `w < z` would not apply to the dependent values of `x` and `y`, resulting in an overapproximated value of `a`.

It is possible to perform a chain of assumptions (constraints), but doing so would require computing all dependencies, which quickly becomes

intractable when values are context-sensitive, i.e., arise from other constrained computations, come from memory, or function parameters. Consider the example in Figure 4.15.

```
a ← amb
b ← amb
c ← amb
if b > 0
  x ← a
else
  x ← c
if x = 0
    /* ... */
```

Figure 4.15: Example of context-sensitive dependencies. After the conditional statement `x = 0`, one of the variables `a` or `c` should also be constrained. However, which value to constrain is unclear because it depends on the previous branch taken.

As one can observe, constraining values statically is often insufficient. Since our main target is dynamic abstraction (in abstract execution or model checking), we can employ constraint propagation dynamically, leveraging the fact that dynamic execution has more accessible context-sensitive data dependencies.

For instance, let us revisit the program Figure 4.15, but now from the dynamic point of view. Suppose the value `x` retains its dependency, i.e., whether it was initialized with `a` or `c`. In that case, we can use this information in the second conditional statement to constrain the respective dependency, which was not doable in static analysis.

Generally, when we impose a constraint function c on the value $\hat{v}$, we must ensure that the constrained value $\hat{v}'$ does not give rise to any new values to preserve analysis soundness:

$$\hat{v}' \sqsubseteq \hat{v} \implies \gamma(\hat{v}') \subseteq \gamma(\hat{v})$$

We can achieve this by simply mandating that any constrained value $\hat{v}'$ computes as $\hat{v}' \triangleq \hat{v} \sqcap c(\hat{v})$, where c represents some constraining function.

There are already known methods for solving similar problems. For example, in [Min06b] author enhances non-relational values with symbolic reasoning. In this approach, author proposes to construct expressions during computation and evaluates expressions in a specified abstract domain. This approach allows specific constraints related to values at evaluation points, similar to the approach of term abstraction.

[Min06b]: Miné (2006), "Symbolic Methods to Enhance the Precision of Numerical Abstract Domains"

We aim to perform similar dynamic value refinement. Our approach involves constraining values and their dependencies whenever an ambiguous branch is taken. To achieve this, we will propagate constrained information in the opposite direction of the dataflow. We will call this process *backward dataflow refinement*.

To be specific, for a statement like `x ← a + b`, if `x` is later in the computation constrained to be positive, we may adjust the values of `a` and `b` used in the addition to ensure that the resulting value is positive. One approach to achieving this is to use inverse computation to derive new values for `a` and `b`, denoted as a' and b' respectively, such that $x' = a' + b'$ where x' is the constrained value of `x`. However, it is important to note that this technique may not always yield perfectly constrained parameters, particularly if the domain is not sufficiently expressive. In some cases, the result of adding constrained values may not produce a strictly positive value. Further details on specific scenarios are outlined in Figure 4.16 and Figure 4.17.

A sequence of data dependencies can be regarded as a single compound expression (term) that produces the given value. For instance, the expression $r = (a + b) - (a + (c + 10))$ corresponds to a chain of several atomic arithmetic statements that perform individual arithmetic operations. This is similar to the approach proposed by [Min06b]. However, in

our approach, we will not directly evaluate these expressions; instead, we will use them to propagate constraints along the expression's structure. That is, we propagate constraints to operation operands from constrained results.

Definition 4.4.1 *We say $\hat{r}'$ is a **constrained result** of $\widehat{op}(\hat{v}_1, \ldots, \hat{v}_n)$ if*

$$\hat{r}' \sqsubseteq \widehat{op}(\hat{v}_1, \ldots, \hat{v}_n)$$

```
a ← amb[-1, 1]
b ← amb[1, 2]
c ← a - b
assume c = 0
```

Consider the above program execution in the non-relational sign domain $\mathcal{A}_{\pm}$. We depict effects of statements in $\wr \ldots \wr$

```
a ← amb[-1, 1]   ⟅a = ⊤±⟆
b ← amb[1, 2]    ⟅b = >0⟆
c ← a - b        ⟅c = ⊤±⟆
assume(c = 0)    ⟅c = 0⟆
```

In general, evaluating the `assume` statement does not constrain other values in non-relational domains. However, for precise analysis, after evaluating the `assume` statement, we may want to constrain value $\wr a = \texttt{>0} \wr$ in order to be more precise with respect to $c = 0$ as it implies that the first argument of subtractions had to be positive (considering non-overflow semantics).
To accomplish this, we can use backward (inverse) data flow to perform an addition operation, which is the inverse of subtraction. Specifically, we can compute a new value of a, denoted as a', using the new constrained values $a' = c + b = \texttt{0} + \texttt{>0}$.

Figure 4.16: Example of the backward refinement process with sign domain.

We constrain operands using backward operations, which define how constraints propagate backward along data dependencies. As each forward operation (operation in usual sense of computation) constraints operands differently, we define for each forward operation $\widehat{op}$ a dedicated backward operation $\overleftarrow{\widehat{op}}$. Its purpose is to constrain the operands of the forward operation $\widehat{op}$ according to the constrained result of the operation. For a forward n-ary operation $\widehat{op} : \mathcal{D}_1 \times \cdots \times \mathcal{D}_n \to \mathcal{D}_r$, the backward operation is defined as follows:

$$\overleftarrow{\widehat{op}} : \mathcal{D}_r \times \mathcal{D}_1 \times \cdots \times \mathcal{D}_n \to (\mathcal{D}_1 \times \cdots \times \mathcal{D}_n)$$

It takes an additional first operand for the new constrained result, which is used to constrain the rest of the operands. The operation returns an n-tuple of constrained operands. To simplify notation, we will often define the backward operation piecewise as a constraining function for a single operand with index i:

$$\overleftarrow{\widehat{op}}_i : \mathcal{D}_r \times \mathcal{D}_1 \times \cdots \times \mathcal{D}_n \to \mathcal{D}_i$$

In the case of binary operations, we use the notation

$$\overleftarrow{\widehat{op}}_l : \mathcal{D}_r \times \mathcal{D}_1 \times \mathcal{D}_2 \to \mathcal{D}_1$$
$$\overleftarrow{\widehat{op}}_r : \mathcal{D}_r \times \mathcal{D}_1 \times \mathcal{D}_2 \to \mathcal{D}_2$$

to denote constraints on the left and right operands, respectively.

To prevent the analysis from producing new infeasible false alarms, it is essential that operand refinement does not introduce any new behavior. This requires that backward operations can only further restrict abstract values, resulting in a smaller set of concrete values (which satisfies the *reduction property*). In addition, to ensure the soundness of the computation (as per the *soundness property*), the operation $\widehat{op}$ with constrained operands cannot underapproximate the given result.

Definition 4.4.2 ***Reduction property.*** *Let $\hat{r}'$ be a constrained result of $\widehat{op}(\hat{v}_1, \ldots, \hat{v}_n)$. Then, the backward operation $\overleftarrow{\widehat{op}}$ is said to satisfy the reduction property if for all $1 \leq i \leq n$ it holds:*

$$\overleftarrow{\widehat{op}}_i(\hat{r}', \hat{v}_1, \ldots, \hat{v}_n) = \hat{v}_i' \sqsubseteq \hat{v}_i$$

In other words, a refined operand value $\hat{v}_i'$ is at most as abstract as the original operand $\hat{v}_i$.

Definition 4.4.3 ***Soundness property.*** *Let $\hat{v}'_i = \overleftarrow{op}_i(\hat{r}', \hat{v}_1, \ldots, \hat{v}_n)$ for $1 \leq i \leq n$ be constrained values of n-ary operation op. Then, the backward operation $\overleftarrow{op}$ satisfies the soundness property if overapproximate the constrained result, but does not introduce more imprecision in comparison to original result:*

$$\hat{r}' \sqsubseteq \widehat{op}(\hat{v}'_1, \ldots, \hat{v}'_n) \sqsubseteq \widehat{op}(\hat{v}_1, \ldots, \hat{v}_n)$$

Precise (best) refinement would result in $\hat{r}' = op(\hat{v}'_1, \ldots, \hat{v}'_n)$ to hold; however, this is not always possible as the domain may not be expressive enough to differentiate specific scenarios. To illustrate, in Figure 4.16, if `b` were to represent $\top_{\pm}$, it would not be possible to infer anything about any of the operands.

In order to perform a single backward propagation step from the constrained value $\hat{r}'$, we must determine how this value was initially derived, including the operation that produced the value $\hat{r}$ and the operands that were used to generate it. To achieve this, we have introduced two new specialized domains: an operation domain that records information about the source operation that produced the value and a dependency domain that maintains references to the values that were involved in creation of the value $\hat{r}$. The operation domain is, in fact, a term domain without given meaning, i.e., formed from uninterpreted symbols, as we need it only to infer dependency structure. With these domains in place, we can formulate an algorithm that recursively performs backward propagation steps on the dependency structure, utilizing terms to apply the appropriate backward operation.

Given backward operations, which are domain-specific, we can devise a universal method for propagating constraints in dynamic abstract executions for non-relational domains. This aligns with our objective of integrating abstraction into the program. To facilitate this, we must integrate the entire backward refinement process into the program semantics, allowing the program to handle refinement autonomously. To accomplish this, we leverage the abovementioned domains that are agnostic to any particular domain and can be used to retarget non-relation abstraction to perform runtime value refinement using its backward operations.

In summary, our proposed *backward dataflow refinement* comprises three fundamental components:

1. The *operation domain*, which captures information about the operations involved in generating values.
2. The *dependency domain*, which keeps track of its runtime dependencies.
3. The *backward refinement algorithm*, which propagates constraints backwards through the dataflow based on the computed dependencies.

In the following sections, we will formally define each of these components separately.

Similarly, let us examine the execution of the program in the non-relational interval domain $\mathcal{A}_i$:

```
a ← amb[-1, 1]   ⦇a = [−1, 1]⦈
b ← amb[1, 2]    ⦇b = [1, 2]⦈
c ← a - b        ⦇c = [−3, 0]⦈
assume(c = 0)    ⦇c = [0, 0]⦈
```

As with the previous case, we can also perform the inverse dataflow computation in the interval domain. However, we cannot simply compute the inverse operation, as this may introduce new values and result in imprecise analysis. Therefore, we must constrain the inverse computation to intersect with the original value. As we can see, these are the exact values that result in a constrained value of $c = 0$:

$$\begin{aligned} a' &= a \sqcap (c + b) \\ &= [-1, 1] \sqcap ([0, 0] + [1, 2]) \\ &= [1, 1] \end{aligned}$$

$$\begin{aligned} b' &= b \sqcap (a - c) \\ &= [1, 2] \sqcap ([-1, 1] - [0, 0]) \\ &= [1, 1] \end{aligned}$$

Figure 4.17: Example of the backward refinement process with interval domain.

Operation Domain

The operation domain Op, records the operation responsible for creating a value. By doing so, we can execute dynamic reflection on the value, query what operation produced the value, and use this information to propagate backward in the data flow.

This domain is rather simple as it merely preserves the syntactic information about the value's producing operation. The expression semantics are presented in Figure 4.18. Evaluating an expression in the operation domain Op results in a syntactic representation of the expression, identified by lowercase names written in small caps.

As an example, when we evaluate the statement `x ← a + b`, we assign an operation value ADD to `x`, which reflects the syntactic information about its origin (i.e., the addition operation). This enables us to conduct backward refinement by establishing a correspondence between operation values and backward transformers. In this case, we determine that we need to use $\overleftarrow{+}$ as it corresponds to the ADD value.

Expression	Op value
amb	AMB
c	CONST
r	REG
v + v	ADD
v − v	SUB
s * s	MUL
s / s	DIV
s % s	REM
s << s	SHL
s >> s	SHR
v = v	EQ
s ≠ s	NEQ
s < s	LT
s > s	GT
s ⩽ s	LTE
s ⩾ s	GTE
zext s **to** ty	ZEXT
sext s **to** ty	SEXT
trunc s **to** ty	TRUNC

Figure 4.18: Expression semantics of the operation domain Op.

Definition 4.4.4 *We define a map* $\text{bop}_{\mathcal{A}} : \text{Op} \to \overleftarrow{\text{Op}}$*, which maps operation domain values to corresponding backward functions in the domain* $\mathcal{A}$*.*

Dependency Domain

The objective of the dependency domain Dep is to preserve immediate dataflow dependencies. By doing so, we can associate each value with the data dependencies that contributed to its creation. The purpose of this domain is to keep track of runtime data dependencies so that we can refine them later.

So far, we have treated abstract values as having value semantics. However, in order to update dependencies in the backward refinement algorithm, the dependency domain considers abstract values to have reference semantics. This means that the elements of the dependency domain are references (pointers in the implementation) to abstract values.

The elements of the dependency domain are sets of referenced values. For example, in the assignment `x ⟵ a ∘ b`, the dependency domain assigns to `x` a value of `{a, b}`, indicating that `x` is data-dependent on both `a` and `b`.

An elementary dependency is a single operand. In our analysis, we focus solely on non-constant dependencies (i.e., values stored in registers) since these are the values that require further constraint. We define $\delta(o)$ to represent the dependency for a single operand o: if o is constant, then $\delta(o) \triangleq \emptyset$ otherwise $\delta(o)$ is a set containing reference to o. Using δ, we define dependency domain semantics in Definition 4.4.5.

Definition 4.4.5 *Dependency domain semantics computes sets of register dependencies:*

Operation	*Effect on* $\hat{\sigma}$
$\mathtt{r_s} \longleftarrow \mathbf{amb}$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \emptyset]$
$\mathtt{r_s} \longleftarrow \mathtt{s}$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \delta(\mathtt{s})]$
$\mathtt{r_s} \longleftarrow \mathtt{s_1} \circ \mathtt{s_2}$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \delta(\mathtt{s_1}) \cup \delta(\mathtt{s_2})]$
$\mathtt{r_s} \longleftarrow \mathbf{cast}\ \mathtt{s}\ \mathbf{to}\ \mathtt{ty}$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \delta(\mathtt{s})]$
$\mathtt{r_p} \longleftarrow \mathtt{p + v}$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \delta(\mathtt{p}) \cup \delta(\mathtt{v})]$
$\mathtt{r_p} \longleftarrow \mathtt{p_1 - p_2}$	$\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \delta(\mathtt{p_1}) \cup \delta(\mathtt{p_2})]$

where $\mathtt{s}$ *is scalar constant or register operand, and* $\mathtt{p}$ *is pointer operand. Operation* $\circ$ *represent all arithmetic, boolean and relational operations in this case.*

When executing a program with the Deps domain, it results in a context and field-sensitive analysis. This domain propagates through memory like other value domains, and whenever we call a function with an abstract value, it is passed as an argument and propagated into the function body execution. This allows the Deps domain to maintain dependencies across both memory and function call (context) boundaries.

4.4.3 Backward Constraint Propagation

Combining both the syntactic and semantic information from the operation (Op) and dependency (Deps) domains, we are able to propagate constraints along the dependency structure. The representation of this structure is a product domain $\mathcal{A} \times \text{Deps} \times \text{Op}$, which we refer to as the functor domain $\widehat{\text{BCP}}(\mathcal{A})$. The abstract domain $\mathcal{A}$ serves as a parameter for the backward constraint propagation and is responsible for performing actual value abstraction and navigating control flow. However, when constraining an abstract value, we use the latter two domains to constrain dependencies appropriately.

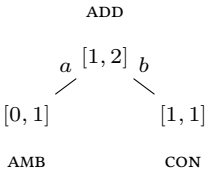
One can view program values in $\widehat{\text{BCP}}(\mathcal{A})$ as a term-like structure, where nodes represent operations that produce their values and edges represent dependencies (cf. Figure 4.19). This information about dependencies and their originating operations allows us to propagate constraint information backward in the dataflow, from the root of the term to its leaves – data dependencies.

Just like in the term domain, in backward constraint propagation, the constraint is applied based on the guard condition in the control flow graph. Each domain determines how it responds to newly generated constraints. For example, it may expand the path condition or initiate backward refinement in the case of backward constraint propagation.

The general process of backward constraint propagation can be outlined as follows. When a guard condition, such as $a < b$, produces an ambiguous result (i.e., `M` value), we constrain it to be only true on the current path. Subsequently, using the latter two components of the domain, we infer how operands of the guard condition were generated (via a lookup table

```
a ← amb[0, 1]
⟅a = ⟨[0, 1], ∅, amb⟩⟆
b ← 1
⟅b = ⟨[1, 1], ∅, con⟩⟆
c ← a + b
⟅c = ⟨[1, 2], {a, b}, add⟩⟆
```

The tree below depicts the "$\widehat{\text{BCP}}$-term" corresponding to c:



If we were to constrain c to the value of 2, we leverage backward constraint propagation to restrict a to the range of $[1, 1]$.

Figure 4.19: Example of $\widehat{\text{BCP}}(\mathcal{A}_i)$.

that maps from forward to backward operations) and apply the obtained backward operation to the value dependencies. This procedure is repeated recursively for the dependencies until we do not constrain the value or reach a leaf of the dependency tree (cf. algorithm 2).

Algorithm 2: Backward Constraint Propagation

Input: value $v \equiv \langle \hat{v}, op, deps \rangle \in \widehat{\mathsf{BCP}}(\mathcal{A})$, where $\hat{v} \in \mathcal{A}$ is an abstract value, $op \in \mathrm{Op}$ is v-producing operation, and $deps \in \mathrm{Deps}$ are its data dependencies

```
1  Function BCP(⟨v̂, op, deps⟩)
2      op̂p ← bop_A(op)
3      foreach ⟨v̂_i, op_i, deps_i⟩ ∈ deps do
4          v̂'_i ← op̂p_i(v̂, values(deps))              // Backward refinement.
5      end
6      foreach ⟨v̂_i, op_i, deps_i⟩ ∈ deps do
7          if v̂_i ≠ v̂'_i then
8              v̂_i ← v̂'_i
9              BCP(⟨v̂_i, op_i, deps_i⟩)          // Recurse to all dependencies.
10         end
11     end
```

In algorithm 2, we present an algorithm for backward constraint propagation, which is invoked when constraining the value of a guard condition. The algorithm includes a function called values : $\wp(\widehat{\mathsf{BCP}}(\mathcal{A})) \to \wp(\mathcal{A})$, which returns the actual abstract values (i.e., the first elements) of the $\widehat{\mathsf{BCP}}(\mathcal{A})$ product, which we want to constrain.

Abstract domains must define their own backward operations because the implementation often relies on the existence of inverses and is tailored to specific cases. Here, we provide examples of backward interval relational operations (cf. Figure 4.20) and arithmetic operations in Figure 4.21. Backward operations are defined in terms of interval arithmetic operations. Note that the backward operations presented are simplified examples. In actual implementation, it is necessary to account for the potential occurrence of division by zero. In such cases, we may need to split the backward refinement into two paths, bypassing the zero divisors.

guard	$\overleftarrow{op}_l$	$\overleftarrow{op}_r$
$[\underline{a}, \overline{a}] = [\underline{b}, \overline{b}]$	$[\underline{a}, \overline{a}] \sqcap [\underline{b}, \overline{b}]$	$[\underline{a}, \overline{a}] \sqcap [\underline{b}, \overline{b}]$
$[\underline{a}, \overline{a}] \neq [\underline{b}, \overline{b}]$	$\{\overleftarrow{<}_l(\mathsf{T}, a, b), \overleftarrow{>}_l(\mathsf{T}, a, b)\}$	$\{\overleftarrow{<}_r(\mathsf{T}, a, b), \overleftarrow{>}_r(\mathsf{T}, a, b)\}$
$[\underline{a}, \overline{a}] < [\underline{b}, \overline{b}]$	$[\underline{a}, \mathit{min}(\overline{a}, \overline{b} - 1)]$	$[\mathit{max}(\underline{b}, \underline{a} - 1), \overline{b}]$
$[\underline{a}, \overline{a}] \leq [\underline{b}, \overline{b}]$	$[\mathit{max}(\underline{a}, \underline{b}), \overline{a}]$	$[\underline{b}, \mathit{min}(\overline{a}, \overline{b})]$

Figure 4.20: Backward relational operations of interval domain.

When faced with inequality in the interval domain, we define two possible refinements that result in the program continuing down two distinct paths. One path considers a smaller than b, while the other assumes a is greater than b.

INTERVAL	BACKWARD	ARITHMETIC
$\widehat{op}$	$\overleftarrow{op}_l$	$\overleftarrow{op}_r$
$r \leftarrow a + b$	$a \sqcap r - b$	$b \sqcap r - a$
$r \leftarrow a - b$	$a \sqcap r + b$	$b \sqcap a - r$
$r \leftarrow a * b$	$a \sqcap r / b$	$b \sqcap r / a$
$r \leftarrow a / b$	$a \sqcap r * b$	$b \sqcap a / r$

Figure 4.21: Examples of backward arithmetic operations.

Example 4.4.1 Consider following backward operations:

- $\overleftarrow{<}(\mathsf{T}, [1,2], [3,3]) = ([1,2], [3,3])$ because $a \in [1,2]$ is guaranteed to be smaller than $b \in [3,3]$.

- $\overleftarrow{<}(\mathsf{T}, [3,4], [2,2]) = (\bot_i, \bot_i)$ because $a \in [3,4]$ is never smaller than $b \in [2,2]$.
- $\overleftarrow{<}(\mathsf{T}, [2,5], [1,4]) = ([2,3], [3,4])$ because $a \in [2,5]$ must be smaller than maximal value of $b \in [1,4] = 4$ and $b \in [3,4]$ must be greater than minimal value of $a \in [2,4] = 2$.

Example 4.4.2 Let us examine the execution of the following program using the $\widehat{\mathrm{BCP}}(\mathcal{A}_i)$ domain:

```
a ← amb[1, 10]
b ← amb[1, 10]
c ← a + 5
⟅c = [6, 15]⟆
if c > 10 (1)
  d ← b - a (2)
  ⟅d = [−9, 4]⟆
  if d = 2 (3)
    /* ... */
```

1. After ambiguous branch we constrain the condition to be true. This invokes backward constraint propagation from the guard result:
   1. Initially, we restrict the guard operands:
      $$\overleftarrow{>}(\mathsf{T}, [6,15], [10,10]) = ([11,15], [10,10])$$
   2. Afterward, we propagate from the constrained value c to its dependencies:
      $$\overleftarrow{+}([11,15], [1,10], [5,5]) = ([6,10], [5,5])$$
2. We use constrained value to get $d = [1,10] - [6,10]$
3. After, the second condition, we sequentially perform following backward operations:
   $$\overleftarrow{=}(\mathsf{T}, d, 2) = \overleftarrow{=}(\mathsf{T}, [-9,4], [2,2]) = ([2,2], [2,2])$$
   $$\overleftarrow{-}(d, b, a) = \overleftarrow{-}([2,2], [1,10], [6,10]) = ([8,10], [6,8])$$

One can observe that simple backward constraint propagation may not yield accurate results. For instance, consider the following backward addition:

$$\overleftarrow{+}([2,5], [0,5], [2,4]) = ([0,3], [2,5])$$

Even with refined operands, the addition produces overapproximated result:

$$[0,3] + [2,5] = [2,8] \sqsupseteq [2,5]$$

For such cases, we explored the possibilities of further refinement. We propose an easy extension to backward operations to split the operands into smaller, more granular cases to achieve greater precision. When applying this approach to addition from the previous example, the result might be as depicted in Figure 4.22.

At this point, two possible approaches: 1) either the computation is split into distinct cases, or 2) a powerset domain is employed to compute with sets of values. However, in more complex scenarios, it may not be feasible to split the computation into all conceivable cases. In these situations, it may be necessary to limit the number of splits to acquire at least a certain level of precision. While these approaches sound promising, we still need to evaluate their effectiveness thoroughly.

In situations where it is unclear how to define a backward operation generally, we can still define the operation to refine only specific cases.

Dividing the calculation into the following four cases leads to best refinement in each of the program's computations:

1. $a = [0,3], b = [2,2]$
2. $a = [0,2], b = [2,3]$
3. $a = [0,1], b = [2,4]$
4. $a = [0,0], b = [2,5]$

While divisions that produce the desired result can be selected freely, omitting certain combinations of values from the original computation would lead to an underapproximation of the analysis.

Figure 4.22: Example of split refinement.

For example, refinement may be limited to corner case values or situations where it can be proven that division by zero will not occur. The only requirement is that operations adhere to soundness and reduction properties.

The most basic form of backward propagation analysis propagates up to the leaves of dependence trees, but it can become impractical for programs with lengthy dependence chains. Therefore, another possible approach is to apply bounded backward propagation as the refinement gains diminish with greater depth. Furthermore, it might be worthwhile to annotate locations with live variables and use this information to prevent unnecessary refinements of no longer necessary variables, consequently avoiding execution splits or further backward constraint propagation evaluation.

Summary

In this chapter, we have expanded the concrete semantics to accommodate scalar abstract domains. We have discussed the relationships between concrete and abstract domains, outlining the requirements for abstract operators and defining how abstraction interacts with control flow using a tristate domain as an intermediate translation layer.

We have developed a control-explicit data-abstract approach, allowing for abstract data treatment while preserving the concrete control flow. Additionally, we have provided a broad introduction to the state-of-the-art of scalar abstraction, defining a number of common scalar domains in the context of our semantics, including the sign and interval domains.

Moreover, we have covered various scalar domains we have designed, such as the unit domain and term domain. We have discussed how these domains are used for program slicing and symbolic computation. Additionally, we have introduced the concept of the functor domain (parametric domain), and have demonstrated its application using common functor domains such as powerset or product domain, which can enhance abstraction precision.

Finally, we have examined domain refinement techniques, with a particular focus on intradomain refinement. We have proposed a novel approach to backward constraint propagation, a refinement technique that enhances non-relational domains by relational reasoning. The main innovation of this approach is that we recast the entire computation as a parametric functor domain, which tracks dependencies and performs refinement autonomously.

5 Aggregates Abstraction

The content of this chapter is based on the following publication:

- Petr Ročkai, Henrich Lauko, Martina Olliaro, and Agostino Cortesi. "String Abstraction for Model Checking of C Programs." In: *Model Checking Software. 26th International Symposium, SPIN 2019, Beijing, China, July 15–16, 2019, Proceedings*. Ed. by Axel Legay Fabrizio Biondi Thomas Given-Wilson. Cham: Springer, 2019. ISBN: 978-3-030-30922-0 [Roč+19]
- Henrich Lauko, Martina Olliaro, Agostino Cortesi, and Petr Ročkai. "Abstracting Strings for Model Checking of C Programs." In: *Applied Sciences. Special Issue Static Analysis Techniques: Recent Advances and New Horizons* 10.21 (2020) [Lau+20]

In the following two chapters we explore non-scalar abstract domains. They abstract infinite sets of in/finite functions [CC14], graphs and other constructs of programming languages like arrays [CCL11b], trees [Mau00], heaps (non-sharing [CC77b; CC77c] or shape analysis [CR08; RSW02]).

RQ1 What are differences between scalar and aggregate abstraction, and how do they interact?

RQ2 How do domain-specific operations contribute to improving the precision of abstraction?

RQ3 What domains are suitable for the abstraction of C strings and arrays?

In a general setting, analyses of programming languages require more specialized abstractions to reason about complex constructs as aggregates or memory, than just the scalar ones. In LLVM IR, we recognize two types of aggregates: homogeneous, like arrays or vectors, or heterogeneous structures. Although it may seem reasonable to represent aggregates as products of scalar abstract domains, in practice such representation is cost-inefficient. Moreover, product representation is restricted only to the exact size of aggregate, hence does not allow an analysis of arbitrary large objects. Therefore, many tools utilize various domain specific abstractions to reason about aggregates efficiently.

Depending on aggregates' nature, we can classify them as aggregates which contain a variable number of items (*arrays*), records that contain a fixed number of items in a fixed layout, where each of these can be of a different type. The items in such aggregates can be (and often are) scalars. However, more complex aggregates are also possible: arrays of records, records which in turn contain other records, and so on.

While scalar domains only deal with elementary values, we consider composite data and memory blocks the fundamental unit of abstraction in aggregate domains. Similarly to scalar domains, abstract aggregate domains approximate a set of concrete aggregates by describing their

[CCL11b]: Cousot et al. (2011), "A parametric segmentation functor for fully automatic and scalable array content analysis"

[Mau00]: Mauborgne (2000), "An incremental unique representation for regular trees"

[CC77b]: Cousot et al. (1977), "Static determination of dynamic properties of generalized type unions"

[CC77c]: Cousot et al. (1977), "Static determination of dynamic properties of recursive procedures"

[CR08]: Chang et al. (2008), "Relational inductive shape analysis"

[RSW02]: Reps et al. (2002), "Shape analysis and applications"

properties. For instance, homogenous aggregates (arrays) are defined by two essential properties: length and content, which are the usual target of aggregate abstraction.

We can equip aggregate domains with arbitrary operations, however, two operations are, in some sense, universal, being elementary memory manipulation operations, namely: byte-wise access and update of the aggregate. The universality of these operations originates from the fact that all aggregate operations can be represented as accesses and updates. In a low-level representation of a program (assembly), they are usually presented in this form. LLVM allows a slightly higher level of manipulation to access and update individual scalars present in the aggregates (as opposed to bytes).[1] All other operations are present in the form of sequences of elementary instructions—possibly encapsulated in functions. Moreover, as in concrete programs, the access and update represents an interface between scalars and memory, in the abstraction, they form an interface between scalar and aggregate domains (even in the case of byte-oriented access since bytes are also scalars).

1: For string abstraction we will perform, though, this distinction is not essential because the scalars stored in C strings are individual bytes (characters).

All memory accesses and, consequently, aggregate manipulations are performed through pointers to aggregates. In LLVM IR, pointers provide two pieces of information regarding a memory location: they identify the *memory block* and an *offset* within that block.[2] In explicit programs, these two components are typically combined into a single value and are often indistinguishable. However, when working with abstract aggregate values, the distinction between the block identifier and offset becomes relevant. In the case of aggregate abstraction, the base address component of the pointer is concrete, as it identifies a specific abstract aggregate. In contrast, depending on the specific abstract aggregate domain, the offset component may or may not be concrete. Representing the offset abstractly, such as with a 32-bit abstract scalar value, may offer advantages in certain cases. Importantly, accessing memory through such a pointer must be regarded as an abstract access or update operation in both cases.

2: In our implementation in DIVINE, the first 32 bits of a pointer are treated as a block identifier, while the last 32 bits are treated as the offset.

In LLVM IR, and likewise in our language, two basic memory access operations are defined — `load` and `store`, corresponding to the access and update operations. It is important to notice that memory access is always explicit: memory is never used in a computation directly. This observation is used in the design of aggregate abstraction, where we can assume that the access to the content of an aggregate will always go through a pointer associated with the abstract aggregate.

Recall abstract domain functors, which are functions that take abstract parameter domains $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n$ and return a new abstract domain $\mathcal{A}(\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n)$. They compose the abstract domain properties of the parameter domains to build a new set of abstract properties and operations. Functor domains are frequently encountered in aggregate abstraction, where aggregates are typically parametrized by the domain of their content or a domain used to reason about aggregate structure, including size. The functor's role is to coordinate these underlying domains and efficiently implement operations on aggregates.

5.1 Abstract Aggregates Memory Semantics

To facilitate reasoning about aggregates, we expand our abstract state definition. Aggregates, at their core, are simply memory blocks, each of them identified by a base address $a \in \mathsf{Id}$. Each aggregate's content can be identified by a cell address $\langle a, o \rangle \in \mathsf{Cell} \equiv \mathsf{Id} \times \mathcal{C}_\mathsf{s}$, which acts like a pointer with base a and offset o. Aggregates abstraction focuses on abstracting the memory blocks Blk, stored in the memory map μ.

Recall semantic domains from Figure 3.2. Each memory block $\mathsf{blk} \equiv \langle s, v \rangle \in \mathsf{Blk} \equiv \mathcal{C} \times \mathbb{V}$ constitutes two components: size s, which represents the number of valid bytes in the allocated block, and a content value function $v : \mathcal{C}_\mathsf{s} \to \mathcal{C}$. The content value function v maps valid offsets to the stored values in the block.

In aggregate abstraction, we reason about memory block properties. We say that an abstract memory block $\widehat{\mathsf{blk}} \in \widehat{\mathsf{Blk}}$ describes a set of concrete memory blocks $\gamma(\widehat{\mathsf{blk}}) \subseteq \mathsf{Blk}$. The abstraction can take two different forms:

1. a size (offsets) abstraction using scalar domain $\widehat{\mathsf{Off}}$ in $\widehat{\mathsf{Blk}} \stackrel{\text{def}}{=} \widehat{\mathsf{Off}} \times \hat{\mathbb{V}}$, where $\hat{\mathbb{V}} : \widehat{\mathsf{Off}} \to \mathcal{C}$ allows concrete values at abstract offsets.
2. a content (value) abstraction using domain $\widehat{\mathsf{Val}}$ to represent aggregate content in $\widehat{\mathsf{Blk}} \stackrel{\text{def}}{=} \mathcal{C}_\mathsf{s} \times \hat{\mathbb{V}}$, where $\hat{\mathbb{V}} : \mathcal{C}_\mathsf{s} \to \widehat{\mathsf{Val}}$ allows abstract values at concrete offsets.

We can apply both forms of aggregate abstraction simultaneously. A general aggregate domain can be expressed as a functor domain $\mathcal{A}(\widehat{\mathsf{Off}}, \widehat{\mathsf{Val}})$, where $\widehat{\mathsf{Off}}$ is a scalar domain used to represent offsets, and $\widehat{\mathsf{Val}}$ is used to abstract memory content. Unlike scalar abstraction, which allows abstract values to be stored in scalar registers and memory, we also need to update memory-dependent parts of the abstract state $\hat{\sigma} \equiv (\hat{\varepsilon}_s, \hat{\varepsilon}_p, \hat{\mu})$, described in Figure 5.1.

$$
\begin{array}{rcll}
\langle a, \hat{o} \rangle \in \widehat{\mathsf{Cell}} & \equiv & \mathsf{Id} \times \widehat{\mathsf{Off}} & \text{(memory cells)} \\
\langle \hat{s}, \hat{v} \rangle \in \widehat{\mathsf{Blk}} & \equiv & (\mathcal{C}_\mathsf{s} \cup \widehat{\mathsf{Off}}) \to \hat{\mathbb{V}} & \text{(memory blocks)} \\
\hat{v} \in \hat{\mathbb{V}} & \equiv & (\mathcal{C}_\mathsf{s} \cup \widehat{\mathsf{Off}}) \to (\mathcal{C} \cup \widehat{\mathsf{Val}}) & \text{(memory block content)} \\
\hat{\varepsilon}_s \in \widehat{\mathbb{E}\mathbb{v}_\mathsf{s}} & \equiv & \mathsf{Reg}_\mathsf{s} \to \mathcal{C}_\mathsf{s} \cup \widehat{\mathsf{Val}} \cup \widehat{\mathsf{Off}} & \text{(scalar registers env.)} \\
\hat{\varepsilon}_p \in \widehat{\mathbb{E}\mathbb{v}_\mathsf{p}} & \equiv & \mathsf{Reg}_\mathsf{p} \to \mathcal{C}_\mathsf{p} \cup \widehat{\mathsf{Cell}} & \text{(pointer registers env.)} \\
\hat{\mu} \in \widehat{\mathbb{E}\mathbb{v}_\mu} & \equiv & \mathsf{Id} \to \widehat{\mathsf{Blk}} & \text{(memory environment)}
\end{array}
$$

Figure 5.1: Aggregate semantic domains.

In summary, we permit pointer registers to contain mixed pointers with abstract offsets, and we allow abstract memory cells to hold abstract values. Note that if $\widehat{\mathsf{Val}}$ was a pointer domain, it would belong to $\hat{\varepsilon}_p$ instead of $\hat{\varepsilon}_s$. For simplicity's sake, we will only consider the scalar content of memory blocks.

Another perspective on aggregate abstraction is that the domain value represents a single allocated memory block, which may also have an

abstract size. This is distinct from more general memory abstraction, which may consider multiple allocations and may also abstract block identifiers – in such cases, we typically refer to it as shape abstraction.

In dynamic memory allocation, unlike in the concrete case, we allow allocations of abstract size $\hat{s} \in \widehat{\mathsf{Off}}$. It is worth noting that allocated blocks have arbitrary (uninitialized) content. Therefore, many aggregate domains are typically equipped with special initializers to address this issue. This is because it may not be practical or even possible to sequentially initialize blocks with abstract sizes, which may be unbounded.

Another difference to scalar abstraction is that aggregates have reference semantics and can be referenced from multiple locations, as multiple pointers can point to the same memory block. Therefore, when making modifications to an aggregate, we need to make them visible from references to the same aggregate. Specifically, the abstract representation of aggregate needs to be shared between abstract pointers.

As is evident from the semantic domains, we assign a block identifier to each abstract aggregate, analogous to how we assign identifiers to concrete aggregates. Thus, abstract pointers consist of "concrete" identifiers, while only offsets can be abstracted. When we modify an aggregate, we access a shared abstract block at the location designated by the identifier. This approach differs from the scalar domain, in which modifications have value semantics and always result in the creation of a new value.

When working with abstract memory blocks, viewing them as elementary values instead of aggregates of bytes can be advantageous. The memory access can retrieve a handle to abstract aggregate and let the domain perform the access given a specific offset. For that, we represent a pointer into an abstracted block as a pair: a handle to single shared storage of the aggregate and an offset, rather than in the concrete case, where the pointer contained the cell address in a single numerical representation. This allows aggregates to be abstracted as a whole rather than on a per-byte basis. Similarly to concrete memory blocks, we store the abstract blocks at a single shared location, which makes it easier to access and soundly modify aggregate from multiple places, including those of possibly concurrent executions.

An aggregate domain $\mathcal{A}(\widehat{\mathsf{Off}}, \widehat{\mathsf{Val}})$ must define the following functions, where $\widehat{\mathsf{Blk}}$ denotes the set of abstract memory blocks, i.e., elements of the aggregate domain.

$$
\begin{aligned}
\mathit{alloc}_{\mathcal{A}} &: \widehat{\mathsf{Off}} \to \widehat{\mathsf{Blk}} \\
\mathit{sizeof}_{\mathcal{A}} &: \widehat{\mathsf{Blk}} \to \widehat{\mathsf{Off}} \\
\mathit{access}_{\mathcal{A}} &: \widehat{\mathsf{Blk}} \times \widehat{\mathsf{Off}} \to \widehat{\mathsf{Val}} \\
\mathit{update}_{\mathcal{A}} &: \widehat{\mathsf{Blk}} \times \widehat{\mathsf{Off}} \times \widehat{\mathsf{Val}} \to \widehat{\mathsf{Blk}}
\end{aligned}
$$

Note that update returns a new abstract block, however, we often perform the update in-place.

The memory operations are defined in terms of elementary operations:

Definition 5.1.1 *(Heap allocation) We define abstract memory allocation of abstract or concrete size* $[\![\hat{\mathtt{s}}]\!]_{\hat{\sigma}} \in \widehat{\mathsf{Off}} \cup \mathcal{C}_{\mathtt{s}}$ *as:*

$$\hat{\sigma} \xrightarrow{\mathtt{r_p} \longleftarrow \textbf{malloc}\ \hat{\mathtt{s}}} \{(\hat{\varepsilon}_s, \hat{\varepsilon}_p[\mathtt{r_p} \mapsto \langle q, 0\rangle], \hat{\mu}[q \mapsto \mathit{alloc}_{\mathcal{A}}([\![\hat{\mathtt{s}}]\!]_{\hat{\sigma}})]) \mid q \in \mathsf{Id}\}$$

and q *is a handle to the newly allocated abstract block. In practice, it is usually a pointer to the storage of abstract aggregate.*

Definition 5.1.2 *(Memory update) To update abstract memory content we extend* `store` *operation to transfers values from registers into abstract memory blocks:*

$$\hat{\sigma} \xrightarrow{\textbf{store}\ \hat{\mathtt{v}} \longrightarrow \mathtt{r_p}} \{(\hat{\varepsilon}_s, \hat{\varepsilon}_p, \hat{\mu}[a \mapsto \mathit{store}_{\hat{\sigma}}(a, \hat{o}, [\![\hat{\mathtt{v}}]\!]_{\hat{\sigma}})]) \mid \langle a, \hat{o}\rangle = [\![\mathtt{r_p}]\!]_{\hat{\sigma}}\}$$

where $\mathit{store}_{\hat{\sigma}}(a, \hat{o}, \hat{v}) \stackrel{\text{def}}{=} \mathit{update}_{\mathcal{A}}(\hat{\mu}(a), \hat{o}, \hat{v})$*. The store performs an abstract update in the pointed abstract memory block* $\hat{\mu}(a) = \widehat{\mathsf{blk}}$ *starting from offset* $\hat{o} \in \widehat{\mathsf{Off}}$*:*

Definition 5.1.3 *(Memory access) To read from memory we extend* `load` *operation to transfer values from abstract memory location* $[\![\mathtt{r}_a]\!]_{\hat{\sigma}}$ *into registers:*

$$\hat{\sigma} \xrightarrow{\mathtt{r_s} \longleftarrow \textbf{load}\ \mathtt{r}_a\ \textbf{of}\ \mathtt{ty}} \{(\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \mathit{load}_{\hat{\sigma}}([\![\mathtt{r_p}]\!]_{\hat{\sigma}}, \mathsf{sizeof}(\mathtt{ty}))], \hat{\varepsilon}_p, \hat{\mu})\}$$
$$\hat{\sigma} \xrightarrow{\mathtt{r_p} \longleftarrow \textbf{load}\ \mathtt{r}_a\ \textbf{of}\ \mathtt{ty}} \{(\hat{\varepsilon}_s, \hat{\varepsilon}_p[\mathtt{r_s} \mapsto \mathit{load}_{\hat{\sigma}}([\![\mathtt{r_p}]\!]_{\hat{\sigma}}, \mathsf{sizeof}(\mathtt{ty}))], \hat{\mu})\}$$

where $\mathit{load}_{\hat{\sigma}}(\langle a, \hat{o}\rangle, n) \stackrel{\text{def}}{=} \mathit{concat}(\{\mathit{access}_{\mathcal{A}}(\hat{\mu}(a), \hat{o} + i) \mid 0 \leq i < n\})$ *reconstructs single multibyte value of size* n *by concatenating bytes from abstract memory block* a *from offset* $\hat{o}$*.*

Similar to the concrete semantics, if the aggregate is homogeneous and the access operations are aligned with the stored values, reconstructing values from bytes may not be necessary. Instead, we can abstract larger objects, such as integers, and omit concatenation, which may lead to the imprecision of abstraction.

It is important to note that these are just templates, and specific domains can override the semantics of memory operations. For example, it may be advantageous for a domain to load multibyte values at once rather than concatenating abstract values.

5.2 Array Abstraction

The simplest and heavily used aggregate – an *array* receive a lot of specific abstract domains for efficient reasoning about array manipulating programs. In the analysis of arrays, the analyzer needs to be able to

discover relationships among values of array elements, as well as their relationships to scalar variables and constants located in the program.

Generally, it is problematic to statically reason about arrays, because of their unbounded nature. The problem is that the array operations do not consider a particular fixed array size. Instead, the array size typically depends on the scalar variables. The precise verification of such program requires being able to reason about relationships between array size, array content and program variables. Hence, analyses tend to utilize symbolic methods to represent constraints on arrays [GRS05].

[GRS05]: Gopan et al. (2005), "A framework for numeric analysis of array operations"

Depending on the interested properties, we recognize two types of array-related techniques. The analysis either focuses on the reasoning about array bounds and related out of bounds errors, or on the content of the array [HP08].

[HP08]: Halbwachs et al. (2008), "Discovering properties about arrays in simple programs"

To reason about array content, abstract domains are usually parametrizable by numerical abstract domains by which the abstraction represents the internal elements of an abstract array [CCL11b]. Symbolic methods are also used to represent or generalize facts about array manipulations – for example, *universally quantified abstract domains*, enable us to quantify over array elements with the use of simpler abstract domains [GMT08]. Predicate abstraction is also used to describe properties about arrays in a form of update predicates [KV09].

[CCL11b]: Cousot et al. (2011), "A parametric segmentation functor for fully automatic and scalable array content analysis"

[GMT08]: Gulwani et al. (2008), "Lifting abstract interpreters to quantified logical domains"

[KV09]: Kovács et al. (2009), "Finding loop invariants for programs over arrays using a theorem prover"

The symbolic techniques naturally request for a refinement of abstractions. The refinement of universally quantified abstract domains is presented in [SPW09]. As universal quantifiers represent hard task for decision procedures, the refinement tries to eliminate their occurrences using ghost variables. Furthermore, the refinement techniques leverage that predicates have a particular form when they reason about array indices. Other refinement approaches utilize lazy-abstraction [Alb+12b] and interpolants [Alb+12c].

[SPW09]: Seghir et al. (2009), "Abstraction refinement for quantified array assertions"

[Alb+12b]: Alberti et al. (2012), "Lazy abstraction with interpolants for arrays"

[Alb+12c]: Alberti et al. (2012), "SAFARI: SMT-Based Abstraction for Arrays with Interpolants"

The recent work [PS18] summarizes how weakly-relational domains for array indices does not scale in the analysis of object-oriented languages. However, in combination with symbolic expressions, traditional domains can scale to real-world codebases.

[PS18]: Payet et al. (2018), "Checking Array Bounds by Abstract Interpretation and Symbolic Expressions"

An alternative method involves program instrumentation [Cor+15] that converts program arrays into code that manipulates scalars. This results in array-free programs that can take advantage of finely-tuned numerical analysis. Moreover, the instrumentation can more easily slice the program only for interested array properties.

[Cor+15]: Cornish et al. (2015), "Analyzing Array Manipulating Programs by Program Transformation"

5.2.1 Array Concrete Semantics

Let $\mu_{\mathsf{A}} \subseteq \mu$ be a set of concrete array environments and $\mathsf{Blk}_{\mathsf{A}} \subseteq \mathsf{Blk}$ set of concrete arrays. A concrete array environment μ_{A} maps pointers to arrays a to their content, such that:

$$\mu_{\mathsf{A}}(a)_\sigma \stackrel{\text{def}}{=} \langle \mathit{size}, v_a \rangle$$

where *size* specifies size of an array a in bytes in state σ, and $v_a \in \mathcal{C}_{\mathsf{s}} \mapsto \mathcal{C}$ maps indices to their corresponding array values. A concrete array is an instance of a parametric aggregate domain. We denote concrete arrays

Let a be a `C` integer array initialized as follows: `a[4] = {5,7,9,11}`. The concrete value of a is given by the tuple

$$\mu_{\mathsf{A}}(a)_\sigma = \langle 4, v_a \rangle,$$

where the size of a is clear from the context and the content map is

$$v_a \triangleq \{0 \mapsto 5, 1 \mapsto 7, 2 \mapsto 9, 3 \mapsto 11\}$$

Figure 5.2: Examples of concrete arrays.

domain as $\mathsf{A}(\mathcal{C}_\mathsf{s}, \mathcal{C})$, where the first parameter is the concrete set of offsets, and the second is the concrete set of content values. Observe that this array representation allows reasoning about the correspondence between shape components of an array and actual values of the array elements.

5.2.2 Smashed Array Domain

The most basic instance of abstract domains for array content is the smashed array domain, which "smashes" (summarizes) all properties of an abstract memory block into a single content variable. Essentially, this means that offsets are disregarded in the analysis. The smashed array can be viewed as a form of an aggregate domain, in which offsets are abstracted within a unit domain ★. This is because every update of an aggregate a is performed on a single element with address $\langle a, ★ \rangle$. In practice, we abstract away from offsets completely (cf. Figure 5.3).

Smashing domain is mostly usable in case we presume absence of undefined behaviors, specifically concerning memory accesses, This allows our analysis to use the domain to assume, for instance, that a memory read can only access to initialized data [GN21].

[GN21]: Gurfinkel et al. (2021), "Abstract interpretation of LLVM with a region-based memory model"

We represent the smashing array as a functor domain $\mathcal{A}_\square(\widehat{\mathsf{Val}})$. Since we ignore the offset part of the array, the elements of the smashing array domain are, in fact, values from the value domain $\widehat{\mathsf{Val}}$. The straightforward approach is to represent memory blocks as summarization scalar values of $\mathcal{A}_\square(\widehat{\mathsf{Val}})$, meaning that we permit scalar values to be in the memory block map $\hat{\mu} : \mathsf{Id} \to \mathsf{Blk} \cup \widehat{\mathsf{Val}}$, i.e., $\widehat{\mathsf{blk}} \in \widehat{\mathsf{Val}}$. Once this is established, memory access operations become operations on scalar values – see Figure 5.3.

$$\mathsf{alloc}_{\mathcal{A}_\square}(o) \stackrel{\text{def}}{=} \bot_{\widehat{\mathsf{Val}}}$$
$$\mathsf{sizeof}_{\mathcal{A}_\square}(\widehat{\mathsf{blk}}) \stackrel{\text{def}}{=} \top$$
$$\mathsf{update}_{\mathcal{A}_\square}(\widehat{\mathsf{blk}}, o, \hat{x}) \stackrel{\text{def}}{=} \widehat{\mathsf{blk}} \sqcup \hat{x}$$
$$\mathsf{access}_{\mathcal{A}_\square}(\widehat{\mathsf{blk}}, o) \stackrel{\text{def}}{=} \widehat{\mathsf{blk}}$$

Figure 5.3: Smashed array operations. When allocating, we create an unused summarization variable. The size of smashed arrays is ambiguous. To update a block $\widehat{\mathsf{blk}}$ with a value $\hat{v}$ at a given offset o, we disregard the offset and join the value $\hat{v}$ to the summarization variable. Likewise, in access, we return the summarization variable.

In Figure 5.4, we provide an example of a computation using the smashed array domain $\mathcal{A}_\square(\mathcal{A}_i)$, which employs intervals as the content domain. We use an abbreviation `a[x]` to carry out pointer arithmetic, which denotes a common element addressing of a cell at offset x of an array with base address a. In abstract allocation ❶, a reference to the smashed value content $\hat{v}$ is created. In the following operations ❷ and ❸, regardless of the offset, the content is joined with the smashed content. For instance, storing values at offsets 1 and 2 results in a join of both stored values: $[4,4] \sqcup [1,2] = [1,4]$. Similarly, loading from an offset that has not been written to, such as 2, results in the smashed content $[1,4]$ at ❹. It is evident that the smashed array overapproximates offsets significantly. To mitigate this, we can use a bounded smashed array that expands to multiple smashed cells representing the prefix and tail of an array. Despite its simplicity, this domain is useful for illustrating the interaction between scalar and aggregate domains.

```
s ← amb      ⦃s ↦ ⊤⦄
a ← malloc s ⦃v̂ ↦ ⊥⦄ ❶
store 4 → a[0] ⦃v̂ ↦ [4,4]⦄ ❷
x ← amb[1, 2]  ⦃x ↦ [1,2]⦄
store x → a[1] ⦃v̂ ↦ [1,4]⦄ ❸
y ← load a[2]  ⦃y ↦ [1,4]⦄ ❹
```

Figure 5.4: Example of smashed array computation instantiated with interval content domain. For brevity we depict only the content map $\hat{v}$ of the array.

5.2.3 Array Segmentation Domains

In the latter part of this chapter, we introduce a string segmentation domain that called M-String is inspired by the general array segmentation abstraction defined in [CCL11a]. We give its brief introduction in this section. Notice that we slightly modified the notation to be consistent with the whole work. For more details, we invite the reader to refer directly to the original paper [Lau+20].

[CCL11a]: Cousot et al. (2011), "A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis"

[Lau+20]: Lauko et al. (2020), "Abstracting Strings for Model Checking of C Programs"

Array Segmentation Abstract Domain Functor.

The functor array abstract domain $\widehat{\mathsf{Af}}$ (shortcut for $\widehat{\mathsf{Af}}(\widehat{\mathsf{Off}}, \widehat{\mathsf{Val}})$), as described in [CCL11a], allows to represent a sequence of consecutive, non-overlapping and possibly empty segments that over-approximate a set of concrete array values in $\wp(\mathsf{Blk})$. Each segment represents a sub-array whose elements share a common property, such as being positive integer values, and is enclosed by segment bounds, which are abstractions of its lower and upper bounds. The elements of functor array belong to the set

$$\widehat{\mathsf{Af}} \stackrel{\text{def}}{=} \{(\widehat{\mathsf{Off}} \times \widehat{\mathsf{Val}}) \times (\widehat{\mathsf{Off}} \times \widehat{\mathsf{Val}} \times \{\text{␣}, ?\})^k \times (\widehat{\mathsf{Off}} \times \{\text{␣}, ?\}) \mid k \geqslant 0\} \cup \{\bot_{\widehat{\mathsf{Af}}}\},$$

which have the form

$$\hat{b}_1 \hat{s}_1 \hat{b}_2 [?_2] \hat{s}_2 \ldots \hat{s}_{n-1} \hat{b}_n [?_n]$$

where:

1. $\widehat{\mathsf{Off}}$ is the segment bound abstract domain approximating array indexes with abstract properties $\hat{b}_i \in \widehat{\mathsf{Off}}$ such that $i \in [1, n]$.
2. $\widehat{\mathsf{Val}}$ is the array element abstract domain representing possible segment values as abstract segment values $\hat{s}_i \in \widehat{\mathsf{Val}}$.
3. the question mark, if present, expresses the possibility that the segment that precedes it may be empty. The question mark can never precede $\hat{b}_1$. The space symbol ␣ represents a non-empty segment.

Example 5.2.1 Let $\mathcal{A}_{\pm}$ be the sign abstraction of numerical values. The segmentation abstract predicate `{0} >0 {2}? <0 {4}` represents arrays of length 4, having either 0 or 2 positive elements followed by either 4 or 2 negative elements, respectively. For instance, it represents: `[7,10,-11,-9]`, `[1, 2, 3, -1]`, or `[-2,-6,-3,-1]`. Note that, in the last case, the lack of positive values is justified by the presence of the question mark that says that the first segment is optional.

Consider the integer array:

```
arr[5] = {5, 7, 9, 10, 12}
```

As an abstraction of `arr` we may consider

```
{0} odd {3} even {5}
```

saying that the array contains odd numbers in the first three elements (indexed from 0 to 2) and two even elements (indexed from 3 to 4).

Figure 5.5: Example of segmentation abstract domain functor.

The *unification algorithm*, described in [CCL11a], aligns two compatible segmentations[3] by modifying them to have the same lists of bounds. While the result of the unification algorithm may not be maximal, it is always well-defined, terminates, and is deterministic. The partial order $\sqsubseteq_{\widehat{\mathsf{Af}}}$ over $\widehat{\mathsf{Af}}$ is defined over unified segmentations, as well as the join $\sqcup_{\widehat{\mathsf{Af}}}$ and the meet $\sqcap_{\widehat{\mathsf{Af}}}$ operators. Note that $\widehat{\mathsf{Af}}$ is not necessarily a lattice [Gan+13]. To ensure the convergence of the analysis, $\widehat{\mathsf{Af}}$ is equipped with a widening operator $\nabla_{\widehat{\mathsf{Af}}}$. A narrowing operator $\Delta_{\widehat{\mathsf{Af}}}$, which improves the precision of the widening result, is also defined. The $\nabla_{\widehat{\mathsf{Af}}}$ and $\Delta_{\widehat{\mathsf{Af}}}$ operators operate on unified segmentations.

Such an abstract array representation is effective for analyzing the content of arrays, but in the case of the `C` programming language, where a string is defined as a null-terminating character array, it is not powerful enough to detect common string manipulation errors.

3: Two array segmentations:

$$\hat{b}_1^1 \ldots \hat{b}_n^1 [?_n^1] \text{ and } \hat{b}_1^2 \ldots \hat{b}_n^2 [?_n^2],$$

are compatible if

$$\hat{b}_1^1 \cap \hat{b}_1^2 \neq \emptyset \text{ and } \hat{b}_{\mathrm{n}}^1 \cap \hat{b}_{\mathrm{n}}^2 \neq \emptyset.$$

[Gan+13]: Gange et al. (2013), "Abstract Interpretation over Non-lattice Abstract Domains"

5.3 String Abstraction

The C programming language continues to be widely used [Cas19], particularly in critical systems such as server-side software and embedded systems. However, C programs are prone to bugs due to the way they manage memory, which can be exploited by malicious actors to carry out security attacks. Therefore, ensuring the correctness of C software is crucial. In particular, we are interested in ensuring the correctness of C programs that manipulate strings, as incorrect string handling can lead to catastrophic events such as data loss or exposure, as well as crashes in critical software components.

[Cas19]: Cass (2019), *The Top Programming Languages 2019. IEEE Spectrum Magazine*

In C, strings are not a basic data type but are instead represented by zero-terminated arrays of characters. Various libraries, in particular, C standard library, provide functions to operate with the such representation of strings [Bul+17]. However, programs that manipulate strings can suffer from buffer overflows and related issues due to discrepancies between the string's size and the array buffer's size. A buffer overflow is a bug that occurs when a buffer is accessed out of its bounds, with out-of-bounds writes being particularly dangerous. Although out-of-bounds reads are less critical, it is still important to design methods for automatic correctness verification of string management in C programs due to the possibility of arbitrary code execution through exploitable buffer overflows [One96]. Code analysis tools can help identify existing bugs and limit the introduction of new bugs, thereby reducing the risk of costly security incidents.

[Bul+17]: Bultan et al. (2017), *String Analysis for Software Verification and Security*

[One96]: One (1996), *Smashing The Stack For Fun And Profit*

Even though array abstraction might be used to handle strings in programs, it is more precise to treat them separately. In C language as well as in LLVM IR, strings are represented as null-terminated arrays of characters. Hence, to detect string related errors, many abstractions extends the array abstractions to keep record about null characters [CO18].

[CO18]: Cortesi et al. (2018), "M-String Segmentation: A Refined Abstract Domain for String Analysis in C Programs"

Since string domains are more restricted, they utilize other formalisms to represent an abstract set of strings like finite automata [CMS03] or trie data structures. The recent work of Constantini [CFC11; CFC15] proposes a unifying approach to string analysis. In his work, presented domains can be tuned to a different level of precision and address only specific properties of strings.

[CMS03]: Christensen et al. (2003), "Precise analysis of string expressions"
[CFC11]: Costantini et al. (2011), "Static analysis of string values"
[CFC15]: Costantini et al. (2015), "A Suite of Abstract Domains for Static Analysis of String Values"

Static methods tailored to identify buffer overflows automatically have been extensively studied in the literature, enclosing tainted dataflow analysis, constraint solvers for string theories, and techniques based on them (e.g., symbolic execution), annotation analysis or string pattern matching analysis [SZ10]. These techniques and their adaptations have been implemented in numerous bug-hunting tools [DRS03; EL02; Hol02; Wag+00; XCE03].

[SZ10]: Shahriar et al. (2010), "Classification of Static Analysis-Based Buffer Overflow Detectors"
[DRS03]: Dor et al. (2003), "CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C"
[EL02]: Evans et al. (2002), "Improving Security Using Extensible Lightweight Static Analysis"
[Hol02]: Holzmann (2002), "UNO: Static Source Code Checking for UserDefined Properties"
[Wag+00]: Wagner et al. (2000), "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities"
[XCE03]: Xie et al. (2003), "ARCHER: using symbolic, path-sensitive analysis to detect memory access errors"

Several publications have demonstrated the usability of abstract interpretation [CC77a] in approximating the semantics of string operations. The most straightforward, well-known domain is a *string set* domain, which simply keeps track of a set of strings – this is a specific instance of the general (bounded) powerset domain. Other domains include, for instance, the *character inclusion* domain, tracking the characters present in a string but not their order or frequency, and the *prefix-suffix* domain, tracking the first and last characters of a string. Another general-purpose string

[CC77a]: Cousot et al. (1977), "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints"

domain is the *string hash* domain proposed in [MA14], based on a distributive hash function. A more complete review of general-purpose string domains is readily available in the literature, e.g. [Ama+17; CFC15].

[MA14]: Madsen et al. (2014), "String Analysis for Dynamic Field Access"

[Ama+17]: Amadini et al. (2017), "Combining String Abstract Domains for JavaScript Analysis: An Evaluation"

[CFC15]: Costantini et al. (2015), "A Suite of Abstract Domains for Static Analysis of String Values"

Such general-purpose domains focus on the generic aspects of strings, without accounting for the specifics of string handling in different programming languages. It is, however, often beneficial to consider such specific aspects of string representation when designing abstract domains for program analysis: indeed, M-String, the domain explored in foundational publications for this chapter, is a domain tailored specifically for the representation of strings used in C programs. With regard to the C programming language, [JMO18] proposed an abstract domain for C strings which tracks both their length and the buffer allocated size into which their are contained. Then, the latter domain, together with the cell abstraction [Min06a], describes relations between length of variables and offsets of pointers. A number of abstract string domains (and their combinations) for analysis of JavaScript programs have been evaluated in [Ama+17]. Another domain that was conceived for JavaScript analysis is the simplified regular expression domain defined in [PIR16]. While dynamic languages heavily rely on strings and their analysis benefits greatly from tailored abstract domains, the specifics of the C approach to strings also deserves attention: the M-String domain, tailored for modeling zero-terminated strings stored in character buffers in C programs has first been described in [CO18]. The domain was initially designed for static analysis, but in collaboration with the original authors, we have refined it for dynamic analysis and compilation-based abstraction as presented in this thesis. This refinement enables the automatic abstraction of C programs, which was not previously considered in the original publication.

[JMO18]: Journault et al. (2018), "Modular Static Analysis of String Manipulations in C Programs"

[Min06a]: Miné (2006), "Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics"

[Ama+17]: Amadini et al. (2017), "Combining String Abstract Domains for JavaScript Analysis: An Evaluation"

[PIR16]: Park et al. (2016), "Precise and Scalable Static Analysis of jQuery Using a Regular Expression Domain"

[CO18]: Cortesi et al. (2018), "M-String Segmentation: A Refined Abstract Domain for String Analysis in C Programs"

We introduce a special purpose string array segmentation domain (M-String) to abstract C strings and adapt it to compilation-based abstraction in publication on string abstraction for model checking of C programs [Roč+19] and its extended journal version [Lau+20]. It is a domain tailored for the analysis of strings in C, whose elements

- approximate sets of C character arrays;
- allow the abstraction of both shape information on the array structure and value information on the contained characters;
- highlight the presence of well-formed strings in the approximated character arrays.

The M-String domain refines the segmentation approach to array representation introduced in [CCL11a]. Its goal is to detect the presence of common string management errors that may lead to undefined behaviors or, more specifically, that may result in buffer overflows. Moreover, keeping track of the content of the characters occurring after the first null character allows us to reduce the number of false positives. In fact, rewriting the first null character in the array is not always an error, as further occurrences of the null character may follow. The M-String, like the array segmentation-based representation introduced in [CCL11a], is parametric in two ways: both with respect to the representation of the indices of the array and with respect to the abstraction of the element values.

[CCL11a]: Cousot et al. (2011), "A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis"

5.3.1 Character Array Concrete Domain

Our goal is to prevent security vulnerabilities that may arise from poorly managed strings in C character arrays. To achieve this, we introduce a concrete value for character arrays that highlights the presence of null characters and define the concept of a *string of interest* – the string that precedes the first occurrence of a null character – for an array of characters. In this thesis, we will provide a brief overview of the domain, as it is not the main topic of focus. Nonetheless, we will showcase the reasoning capabilities of our approach about string manipulations.

Let $\mathsf{Ch} \subseteq \mathcal{C}_\mathsf{s}$ be a finite set of characters representable by the character encoding in use equipped with a top element $\top_{\mathsf{Ch}}$ representing an unknown value. A concrete character array domain is an instance of a concrete array, where content are characters from Ch. We denote the set of concrete character arrays as ACh, which is short for a concrete array with concrete offsets and character values $\mathsf{A}(\mathcal{C}_\mathsf{s}, \mathsf{Ch})$. Moreover, we define a projection $\mathsf{nulls} : \mathsf{Blk}_{\mathsf{ACh}} \mapsto \wp(\mathcal{C}_\mathsf{s})$ that returns a set of terminating characters present in a given character array:

$$\mathsf{nulls}(\langle s, v\rangle) \stackrel{\text{def}}{=} \{i \mid v(i) = \texttt{'\textbackslash 0'}\}$$

For a well-formed string, it holds that it contains at least one terminating character, which means that the projection nullsis not empty. Moreover, character array elements that have not been initialized are mapped to the top value $\top_{\mathsf{Ch}}$, as one can read garbage values from the allocated array.

We formally define the string of interest of a character array as the sequence of its elements up to the first terminating one (included).

Definition 5.3.1 (String of Interest) *Let $a \in \mathsf{ACh}$ be an array of characters with concrete value $a = \langle s, v\rangle$ and let n be the minimum element of $\mathsf{nulls}(a)$ (if it is non-empty). The string of interest of the character array a is defined as follows:*

$$\mathit{string}(a) = \begin{cases} \langle n+1, \{i \mapsto c \mid 0 < i \leq n \wedge v(i) = c\}\rangle & \text{if } \mathsf{nulls}(a) \neq \emptyset \\ \bot & \text{otherwise} \end{cases}$$

Our definition of the string of interest of character arrays allows us to distinguish well-formed strings from the wrong usage of character arrays. If the null character appears at the first index of a character array, we say its string of interest is empty. In general, we refer to character arrays that contain a well-defined or empty string of interest as character arrays that contain a *well-formed string*.

Moreover, when the allocated memory capacity is not sufficient for a declared character array, the system writes a null character outside the array, occupying memory that is not intended for it and causing a buffer overflow. We do not represent this system behavior because it leads to an undefined one. Therefore, we consider the string of interest of such character arrays as undefined ($\bot$).

Let `m` be a C character array initialized as follows:

```
m[6] = {'b','e','e','\0','b'}.
```

The concrete value of `m` is given by the tuple $\mu(m) = \langle 6, v\rangle$, where the content map is

$$v_m \triangleq \{0 \mapsto \texttt{'b'}, 1 \mapsto \texttt{'e'}, 2 \mapsto \texttt{'e'}, 3 \mapsto \texttt{'\textbackslash 0'}, 4 \mapsto \texttt{'b'}, 5 \mapsto \top_{\mathsf{Ch}}\}$$

and $\mathsf{nulls}(\mu(m))$ is the singleton $\{3\}$, being the the only array cell that certainly contains a null character. The string of interest is the sequence `"bee\0"`.

Figure 5.6: Example concrete character array.

In the original publication, we presented the complete formal concrete and abstract semantics of the character array domain. Additionally, we provided definitions for all requested operations for the abstract domain, including string subsumption, join, meet, and elementary access and update operations. Although abstract interpretation definitions are relevant to our work, our primary focus is not on the technicalities of this domain. Therefore, we only provide an overview of the domain's intuition and its integration with our model of computation in this thesis. We invite readers to refer to our publications [Lau+20; Roč+19] for further inspection of domain semantics and proofs of its soundness.

[Lau+20]: Lauko et al. (2020), "Abstracting Strings for Model Checking of C Programs"
[Roč+19]: Ročkai et al. (2019), "String Abstraction for Model Checking of C Programs"

In brief, string access is defined in the same manner as on generic arrays, while string updates also modify the nulls projection if a new terminating character is introduced or if an existing one is overwritten. However, the most intriguing aspects of string abstraction pertain to the other operations that can be performed on the domain. Given that `C` includes several strings-related functions in the standard library `string.h`, abstracting them allows for more accurate program analysis.

We focus on the most significant functions in the `string.h` header (described in the remainder of this section), manipulating null-terminated sequences of characters, plus the array elements access and update operations. Recall that `char`, `int` and `size_t` are data types in `C`, `const` is a qualifier applied to the declaration of any variable which specifies the immutability of its value, and `*str` denotes that `str` is a pointer variable.

```
char* strcat( char *dst, const char *src )
```

The function `strcat` appends the null-terminated string pointed to by `src` to the end of the string pointed to by `dst` to concatenate the two. The null-terminator of `dst` is replaced by the first character of `src`. It is important to note that the two strings should not overlap, meaning that `src` and `dst` should not point to the same memory location. Upon completion of the concatenation, the function returns a pointer to the modified `dst` string.

Formally, the `strcat` operation embeds the string of interest from `src` array into the content map of the `dst` array, beginning from the offset of the first occurrence of the null-terminator.

```
char* strchr( char *str, int ch )
```

The function `strchr` locates for the first occurrence of the character `ch` (converted to a `char`) in the string pointed to by `str`. The terminating null character is included in the search. If the character is found, the function returns a pointer to the located character; otherwise, it returns a null pointer.

```
int strcmp( const char *lhs, const char *rhs )
```

The function `strcmp` performs a lexicographic comparison of the string pointed to by `lhs` to the string pointed to by `rhs`. It returns an integer

greater than, equal to, or less than zero, depending on whether the string pointed to by `lhs` is greater than, equal to, or less than the string pointed to by `rhs`, respectively.

```
char* strcpy( char *dst, const char *src )
```

The function `strcpy` copies the null-terminated string pointed to by `src` to the memory pointed to by `dst`. The strings pointed to by `src` and `dst` should not overlap. The function returns the pointer `dst`.

```
size_t strlen( const char *str )
```

The function `strlen` computes the number of bytes in the string to which `str` points, not including the terminating null byte. The string length function returns the length of string of interest located in `str`.

It may also be worthwhile to perform abstraction of other memory-manipulating functions of generic array aggregates listed in Figure 5.7. In fact, authors in the recent work [BCD22] investigate the application of symbolic abstraction to operations such as `memcpy` or `memset`. However, in our work, we focus only on string-manipulating functions.

[BCD22]: Borzacchiello et al. (2022), "Handling Memory-Intensive Operations in Symbolic Execution"

```
int   memcmp( const void* lhs, const void* rhs, size_t n )
void* memset( void *dst, int ch, size_t n )
void* memcpy( void *dst, const void *src, size_t n )
void* memchr( const void *ptr, int ch, size_t n )
void* memmove( void* dst, const void* src, size_t n )
```

Figure 5.7: Character array manipulation functions in C from `<string.h>`.

As previously mentioned, C does not provide guarantees for bounds checking on array accesses. When it comes to strings, the language also does not ensure that they are null-terminated. This lack of protection results in multiple vulnerabilities and exploits [Sea13]. For example, passing non-null-terminated strings to the aforementioned functions can lead to misleading results or reading beyond the array boundary. Additionally, since `strcat` and `strcpy` do not permit the specification of the destination array's `dst` size, they are common sources of buffer overflows.

[Sea13]: Seacord (2013), *Secure Coding in C and C++*

5.3.2 Character Array Abstract Domain

In the previous section, we established the concrete character array values that indicate the existence of a valid string within them. Additionally, we introduced our concrete domain ACh, which is composed of sets of character array values, as well as the concrete projection of null-terminating characters. In the next section, we will define the $\widehat{\mathsf{Ch}}$ abstract domain, which approximates the elements within ACh. This domain is called the M-String domain ($\widehat{\mathsf{M}}$ for short).

The M-String abstract domain approximates sets of concrete character array values using a pair of segmentations that highlights the nature of their strings of interest. The domain's elements are split segmentation abstract predicates. Segments represent sequences of characters that share the same property and are bounded by segment bounds. Similar to the array segmentation domain (recalled in Section 5.2.3), the M-String abstract domain is a functor denoted as $\widehat{\mathsf{M}}(\widehat{\mathsf{Off}}, \widehat{\mathsf{Ch}})$, where:

1. $\widehat{\mathsf{Off}}$ denotes the scalar abstraction of segment bounds, equipped with the addition and subtraction operations,
2. $\widehat{\mathsf{Val}}$ is the scalar abstraction of the character array elements.

Elements of M-String belong to the set $\widehat{\mathsf{M}} \triangleq (\widehat{\mathsf{M}}_s, \widehat{\mathsf{M}}_t) \cup \{\bot_{\widehat{\mathsf{M}}}, \top_{\widehat{\mathsf{M}}}\}$ such that:

1. $\widehat{\mathsf{M}}_s$ represents the segmentation of the strings of *interest* of a set of character arrays as value from the set:

$$\widehat{\mathsf{M}}_s \stackrel{\text{def}}{=} \{(\widehat{\mathsf{Off}} \times \widehat{\mathsf{Ch}}) \times (\widehat{\mathsf{Off}} \times \widehat{\mathsf{Ch}})^k \mid k \geqslant 0\} \cup \{\widehat{\mathsf{Off}}\} \cup \{\emptyset\}$$

2. $\widehat{\mathsf{M}}_t$ represents the segmentation of the content of character arrays after their string of interests, or character arrays that do not contain the null terminating character:

$$\widehat{\mathsf{M}}_t \stackrel{\text{def}}{=} \{(\widehat{\mathsf{Off}} \times \widehat{\mathsf{Ch}}) \times (\widehat{\mathsf{Off}} \times \widehat{\mathsf{Ch}})^k \mid k \geqslant 0\} \cup \{\emptyset\}$$

3. $\bot_{\widehat{\mathsf{M}}}, \top_{\widehat{\mathsf{M}}}$ are bottom/top elements of $\widehat{\mathsf{M}}$.

Consider the split segmentation abstract predicate

$$\widehat{m} = ([0,0]\texttt{'a'}[2,5], \emptyset)$$

where $\widehat{\mathsf{Ch}}$ is the concrete domain for characters and $\widehat{\mathsf{Off}}$ the interval domain. The segmentation $\widehat{m}$ approximates character arrays certainly containing a string of interest which is a sequence of `'a'`, whose length goes from 2 to 5, followed by a null character, e.g., `"aa\0"` or `"aaaaa\0"`.

Figure 5.8: Example of an abstract string segmentation.

The set $\widehat{\mathsf{M}}$ can be expressed as a collection of split segmentation abstract predicates in the form of $\widehat{m} = (s, t)$. For instance, if $\widehat{m}$ takes the value $(\hat{b}_1 \hat{c}_1 \hat{b}_2, \emptyset)$, it represents concrete character array values of length $\hat{b}_2$ byte that contain single segment of characters $\hat{c}_1$ in the string of interest. Conversely, if $\widehat{m}$ is equal to $(\hat{b}_1, \hat{b}_2 \hat{c}_2 \hat{b}_3 \ldots \hat{b}_n)$, it approximates concrete character array values that have a length of at least one cell and include an empty string of interest. In particular:

1. $\hat{b}_i \in \widehat{\mathsf{Off}}$ denotes the segment bounds, such that $i \in [1, n]$ and $n > 1$. Note that $\hat{b}_1$ and $\hat{b}_n$ respectively represent the segmentation lower and upper bound.
2. $\hat{c}_i \in \widehat{\mathsf{Ch}}$ are scalar abstractions of segment content.

By employing this approach, it is possible to represent segments that may be empty in cases where bounds overlap such that $\gamma(\hat{b}_i) \cap \gamma(\hat{b}_{i+1}) \neq \emptyset$. If bounds are arranged in a strictly ordered manner such that $\mathit{min}(\gamma(\hat{b}_{i+1})) > \mathit{max}(\gamma(\hat{b}_i))$, they represent a nonempty segment.

Similar to array segmentation domain, M-String has provides $\sqcup_{\widehat{\mathsf{M}}}$, meet $\sqcap_{\widehat{\mathsf{M}}}$, widening $\nabla_{\widehat{\mathsf{M}}}$, and narrowing $\Delta_{\widehat{\mathsf{M}}}$ operators. We give their exhaustive description in [Lau+20].

Abstraction

Let M be a set of concrete character array values. The abstraction function on the M-String abstract domain $\alpha_{\widehat{\mathsf{M}}}(M)$ maps M to $\bot_{\widehat{\mathsf{M}}}$ in the case in which M is empty, otherwise to the pair of segmentations that optimally over-approximates values in M.

In program abstraction, it is common to have cases where the set M contains only one element, in which case the abstraction eagerly creates segments. For example, for the character array $c_1 \dots c_n$, the first segment represents characters $c_1 \dots c_i$ as an abstract character $\hat{c} \in \widehat{\mathsf{Ch}}$, where:

$$\forall_{j=0}^{i} \alpha_{\widehat{\mathsf{Ch}}}(c_j) = \hat{c} \wedge (\alpha_{\widehat{\mathsf{Ch}}}(c_{i+1}) \neq \hat{c} \vee i + 1 > n),$$

and the bound of the segment is $\hat{b} = \alpha_{\widehat{\mathsf{Off}}}(i+1)$. This process is repeated until all segments are built, and if a null terminator is encountered, the segmentation is split. Finally, all abstractions of character arrays from M are joined. Additionally, our implementation, allow programs to create segmentation directly without defining the set of concrete strings to be abstracted.

Concretization

The concretization function on the M-String abstract domain $\gamma_{\widehat{\mathsf{M}}}$ maps an abstract element to a set of concrete character array values as follows: $\gamma_{\widehat{\mathsf{M}}}(\bot_{\widehat{\mathsf{M}}}) = \emptyset$, otherwise $\gamma_{\widehat{\mathsf{M}}}(\widehat{m})$ is the set of all possible character array values represented by a split segmentation abstract predicate $\widehat{m}$.

The concretization can be understood in terms of individual segments. For a segment $\langle \hat{b}_1, \hat{c}, \hat{b}_2 \rangle$, it maps all possible concrete values $c \in \gamma_{\widehat{\mathsf{Ch}}}(\hat{c})$ to the concrete array content within the range of bounds

$$\mathit{min}(\gamma(\hat{b}_1)) \leq k < \mathit{max}(\gamma(\hat{b}_2))$$

The concretization of a string of interest is then the concatenation of all its concretized segments:

$$\gamma_{\widehat{\mathsf{M}}}(\widehat{m}) \stackrel{\text{def}}{=} \{c_1^{b_2} \cdot c_2^{b_3} \cdot \dots \cdot c_n^{b_{n+1}} \mid c_i = \gamma_{\widehat{\mathsf{Ch}}}(\hat{c}_i) \wedge b_i = \gamma_{\widehat{\mathsf{Off}}}(\hat{b}_i)\}$$

Likewise, we can convert the tail of the split segmentation (i.e., the part following the first null-terminator) into a concrete representation, which can then be appended to the desired concrete string.

Abstract Character Array Operations

We define the essential abstract operations of M-String domain more formally in [Lau+20]. In brief, the `access` at position $\hat{i} \in \widehat{\mathsf{Off}}$ returns the abstract character of the segment $\langle \hat{b}_1, \hat{c}, \hat{b}_2 \rangle$ for which $\hat{b}_1 \leq \hat{i} < \hat{b}_2$ holds. If $\hat{i}$ falls into multiple segments (see Figure 5.9), we can either split the computation into multiple executions or join all possible results. Similarly, updating a particular offset may either split a region or have no effect if it is already included in the abstraction of segment content.

The main benefit of representing strings in a dedicated domain is the ability to utilize "domain-specific" abstractions of operations. As described in the previous section, these abstractions include the standard `C` library string operations. Although it would be possible to abstract these operations solely through abstract updates and accesses, for possibly infinite (abstractly bounded) strings, such approaches would typically not terminate as they generally iterate up to the null-terminating character.

Consider the split segmentation abstract predicate

$$\widehat{m} = ([0,0]\texttt{'a'}[2,5]\texttt{'b'}[3,6], \emptyset)$$

Accessing the index 1, i.e., $[1,1]$ in the interval domain, corresponds to accessing the first segment only:

$$[0,0] < [1,1] < [2,5]$$

Therefore, it yields only the value `a`. However, accessing the index 3 can fall into both segments. In that case, we can either yield both possible results `a`, `b` or fork the computation to consider both results.

Figure 5.9: Example of M-String `access` operation.

```
size_t strlen(const char *str)
{
    const char *s;
    for (s = str; *s; ++s);
    return (s - str);
}
```

Figure 5.10: GNU C `strlen` function.

An example of this is the string length operation `strlen` from the GNU C standard library implementation depicted in Figure 5.10.

Consider a split segmentation of infinitely many characters `x`:

$$\widehat{m} = ([0,0]\texttt{'x'}[1,\infty], \emptyset)$$

If we were to abstract the `strlen` operation purely in terms of abstract updates and accesses, each iteration of the for loop would satisfy the condition that the character is not a null-terminating character, and the loop would iterate infinitely without reaching the ∞ bound. However, in the M-String abstraction, we know the size of the string of interest, which is the last bound of the string of interest. Therefore, we can abstract the `strlen` operation as a constant-time operation that returns the particular bound.

Likewise, when performing abstract operations, we do not need to iterate through characters but rather through segments. For example, the operation `strchr`, which finds the first occurrence of a given character $\hat{c}$, can be implemented in linear time to the number of segments. We can iterate through the segments of the string of interest and return the beginning bound of the segment that satisfies $\hat{c} \sqsubseteq \hat{c}_s$, where $\hat{c}_s$ is the segment character. This operation may be unfeasible or much more computationally intensive in terms of pure accesses and updates. In string comparison operation `strcmp`, we iterate over segments and compare their bounds and content, again the operation being linear in the number of segments.

In the case of string-modifying operations such as `strcat`, we iterate over segments and concatenate them to the end of the string of interest while dropping segments from the tail of the split segmentation. Similarly, in the case of `strcpy`, we overwrite the string of interest. However, the tricky part of these operations is that they can arbitrarily interleave with present regions. Since we have abstract bounds, we need to consider all possible rewrites.

Example 5.3.1 Consider two abstract strings in $\widehat{\mathsf{M}}$:

$$([0,0]\ \texttt{'a'}\ [5,7], [6,8]\ \texttt{'b'}\ [13,14]) \text{ and } ([0,0]\ \texttt{'a'}\ [3,3], \emptyset)$$

Here, $\widehat{\mathsf{Off}}$ is the interval domain over array indexes and $\widehat{\mathsf{Ch}}$ is the concrete domain. The first element abstracts all character arrays whose string of interest contains character 'a', with a length ranging from 5 to 7, followed by the null character, and string formed from 'b' characters, with a length ranging from 5 to 8. The second element abstracts all character arrays whose string of interest is a string of length 3, containing only 'a' character.

Now, let us examine the concatenation of these two abstract strings. To begin, we must ensure that the destination string has enough space to hold the concatenated string. This condition is met, as the capacity of the destination split segmentation is 13, which is greater than the maximum size of the concatenated string of interest: $7 + 3 + 1$. This sum corresponds to the maximum length of the destination string of

interest, the maximum length of the source string of interest, and an additional null character.

The concatenation results in the following abstract string:

$$([0,0]\ \texttt{'a'}\ [5,7]\ \texttt{'a'}\ [8,10], [9,11]\ \texttt{'b'}\ [13,14])$$

which is equivalent to:

$$([0,0]\ \texttt{'a'}\ [8,10], [9,11]\ \texttt{'b'}\ [13,14]).$$

We have provided an intuitive explanation of string operations in our discussion so far. In our original publication [Lau+20], we provide a formal semantic description of these operations and prove that they are sound approximations of the concrete semantics of string operations. While I acknowledge their importance, I will not delve further into the domain's technical details here as they are not directly related to my contribution and might impede readability without adding much value.

5.3.3 Parametrization of M-String Domain

As an aggregate domain, M-String is a parametrizable by scalar domains of characters and indices (bounds). This allows us to tailor the abstraction to the needs of the analysis of string values. Depending on the precision of chosen domains, the instance of the M-String domain will inherit their properties. With more precise domains, the M-String values will maintain higher granularity of segmentation. On the other hand, simpler character representation will decrease the segmentation granularity for the cost of a higher rate of false alarms.

A particular instance of M-String is automatically derived from a parametric description given in [Lau+20], provided a suitable scalar domain $\widehat{\mathsf{Ch}}$ for characters and scalar domain $\widehat{\mathsf{Off}}$ to represent segment bounds. The instantiation demands that both scalar domains $\widehat{\mathsf{Ch}}$ and $\widehat{\mathsf{Off}}$ are equipped with operations that appear in the operations with the segmentation. These are mainly elementary arithmetic and relational operations. In the implementation, we provide an M-String domain template that automatically derives all the operations from provided scalar domains.

Concrete Characters, Symbolic Bounds

We have experimented with two instantiations of the M-String domain. The first simpler instantiation sets the domain of characters $\widehat{\mathsf{Ch}}$ to be the concrete domain $\mathcal{C}$ (i.e., we let the characters be represented by themselves). We let the domain of segment bounds $\widehat{\mathsf{Off}}$ to be a symbolic 32-bit integers. This instantiation balances between simplicity on the one hand and the ability to describe strings with undetermined length and structure.

At the implementation level the domain remains generic. The particular domains we selected can be easily replaced by other scalar domains. Compared to the theoretical description of M-String, the implementation uses a slightly simplified representation of segmentation using a pair of arrays (as shown in Figure 5.11). The elements of these arrays are characters

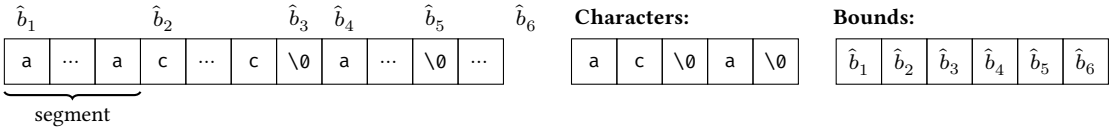


Figure 5.11: M-String value with symbolic bounds, where string of interest is from $\hat{b}_1$ to $\hat{b}_3$.

and bounds, and their type is derived from parametrization, i.e., from the scalar domains $\widehat{\mathsf{Ch}}$ and $\widehat{\mathsf{Off}}$. This modification to the representation is just an optimization for the implementation and does not impact the semantics of the operations. The analysis using this representation is presented in Example 5.3.2.

This instantiation of M-String is well-suited for representing strings that consist of variable-length sequences of a single character, such as strings of the form $a^k b^l c^m$, where the relationships between k, l, m can be specified using standard arithmetic and relational operators, and each of a, b, c is a concrete letter. This, in turn, allows M-String to be used for the analysis of program behavior on broad classes of input strings described this way. A more in-depth discussion of this approach can be found in the experimental section in Appendix C.

Example 5.3.2 Consider program abstraction with symbolic bounds and concrete characters:

```
str ← mstring(x, b̂1, \0, b̂2, y,  b̂3, \0, b̂4)
i ← amb
if i < b̂4
   store 'y' → str[i]
   len ← strlen(str)
```

Consider symbolic bounds $\hat{b}_1 < \hat{b}_2 < \hat{b}_3 < \hat{b}_4$, the abstract program creates an instance of `mstring` with characters $[\mathtt{x}, \mathtt{\backslash 0}, \mathtt{y}, \mathtt{\backslash 0}]$ and bounds $[0, \hat{b}_1, \hat{b}_2, \hat{b}_3, \hat{b}_4]$. Hereafter, we represent the `mstring` value as a pair of these two arrays. An abstract index is created on the second line and constrained on line 3 to be smaller than the maximum string length to avoid failure of the subsequent update operation. The character `y` is assigned to the position indicated by the symbolic index, resulting in multiple possible strings $\widehat{str}_x$, depending on the value of i. To obtain the final string, all possible strings are joined, which are the following:

1. if $i < \hat{b}_1$: i falls to the first segment:

$$\widehat{str}_1 = ([\mathtt{x}, \mathtt{y}, \mathtt{x}, \mathtt{\backslash 0}, \mathtt{y}, \mathtt{\backslash 0}], [0, i, i+1, \hat{b}_1, \hat{b}_2, \hat{b}_3, \hat{b}_4])$$

   and creates a new segment between i and $i{+}1$ containing character `y`. Notice that if $i = 0$ the first segment is empty, similarly the third segment for $\mathtt{i} + 1 = \hat{b}_1$. The string of interest for $\widehat{str}_1$ is of form

$$string(\widehat{str}_1) \triangleq x^i y^1 x^{\hat{b}_1 - i - 1}$$

2. if $\hat{b}_1 \leq i < \hat{b}_2$ than

$$\widehat{str}_2 = ([\mathtt{x}, \mathtt{\backslash 0}, \mathtt{y}, \mathtt{\backslash 0}, \mathtt{y}, \mathtt{\backslash 0}], [0, \hat{b}_1, i, i+1, \hat{b}_2, \hat{b}_3, \hat{b}_4]),$$

where string of interest is a join of following forms:

- $string(\widehat{str}_2) \triangleq x^{\hat{b}_1} y$ when the update is performed right after the first segment ($i = \hat{b}_1$) and the segment of zeros contains more elements $\left|\hat{b}_1 - \hat{b}_2\right| > 1$,
- $string(\widehat{str}_2) \triangleq x^{\hat{b}_1} y^{\hat{b}_3 - \hat{b}_1}$ when the update is performed right after the first segment ($i = \hat{b}_1$), and the segment of zeros has a single character $\left|\hat{b}_1 - \hat{b}_2\right| = 1$,
- $string(\widehat{str}_2) \triangleq x^{\hat{b}_1}$ otherwise between first segment and i is a terminating zero, hence the string of interest remains unchanged.

3. if $\hat{b}_2 \leq i < \hat{b}_3$ than $\widehat{str}_3 \equiv \widehat{str}$, because update stores the same character as is already present in the segment.
4. if $\hat{b}_3 \leq i < \hat{b}_4$ than update creates a new segment inside of sequence of last zeros:

$$\widehat{str}_4 = ([\mathtt{x}, \backslash\mathtt{0}, \mathtt{y}, \backslash\mathtt{0}, \mathtt{y}, \backslash\mathtt{0}], [0, \hat{b}_1, \hat{b}_2, \hat{b}_3, i, i+1, \hat{b}_4])$$

Thus, the `strlen` operation on the last line of the program computes the union of all possible lengths of strings of interest, which can be expressed as $\hat{b}_1 \cup (\hat{b}_1 + 1) \cup \hat{b}_3$.

Symbolic Characters, Symbolic Bounds

The second instantiation is used in benchmarks, where the computation with M-String values encountered abstract scalars (characters). This occurs when the program obtains some character as input from the environment and tries to store it into the M-String value. Therefore, we instantiated the M-String domain with an abstract representation of characters by setting the domain C to be the term domain, which keeps track of symbolic 8b bit-vectors (characters in C language). In this way, we do not need to lower abstract characters before storing them to the M-Strings, what was needed for the concrete domain used in the previous instantiation. However, we pay the price for more expensive computation with symbolic characters.

In order to provide evidence of the effectiveness of M-String, alongside related publications, we extended LART [LRB18], a tool that performs automatic abstraction of programs, making it also support aggregate (non-scalar) domains like M-String.

We extended LART along with DIVINE 4 [Bar+17], an explicit state model checker based on LLVM. This way, we can verify the correctness of operations on strings in C programs automatically. The experimental evaluation is performed by analyzing a number of C programs, ranging from quite simple to moderately complex, including parsers generated by `bison`, a tool that translates context-free grammars into C parsers. The evaluation results demonstrate the benefits and correctness of abstracting more complex functions from the string library. For further details, including evaluation artifacts, please refer to Appendix C.

Summary

This chapter presents an extension of abstract semantics to aggregate domains. Unlike scalar abstraction, aggregate domains can abstract possibly infinitely large content of memory blocks and are usually defined by two domains: the offset domain and the content domain.

The techniques were demonstrated on elementary array abstractions and applied to a non-trivial string segmentation domain known as M-String. Notably, the designed abstraction includes domain-specific operations, for which we will incorporate generic support into the compilation-based abstraction approach in the following chapters.

A novel segmentation-based abstract domain has been introduced for approximating C strings. This new approach abstracts both index bounds and substrings while managing strings as a pair of two string buffers: the string of interest and a tail of allocated but possibly unused memory. This method allows for a more precise modeling of the standard C library functions for strings, while also addressing known weaknesses in managing terminating null characters and buffer bounds. The M-String domain is effective at identifying security leaks resulting from string manipulation errors, such as buffer overflows.

Besides presenting the theoretical framework and basic operations on strings, we implemented the abstract semantics using C++ language and combined them with a tool that lifts string-manipulating C codes to the M-String domain. Our experimental results focused on tuning the parameters of M-String, including the domains for both segment content and segment bounds. We instantiated these parameters using both concrete and symbolic characters and symbolic (bit-vector) bounds. The presented approach combined with compilation-based abstraction allowed us to extend the explicit-state model checker DIVINE to compute with abstract stings without any further modifications.

Dynamic Memory Abstraction | 6

The content of this chapter is based on the following publication:

▶ Henrich Lauko, Lukáš Korenčik, and Petr Ročkai. "Verification of Programs Sensitive to Heap Layout." In: *ACM Transactions on Software Engineering and Methodology* 31 (4 2022) [LKR22]

In this chapter, we extend our abstraction by introducing the concept of dynamic memory abstraction, which is the last category of abstraction we will discuss in this thesis. The preceding scalar and aggregate abstractions focused on the contents of registers and memory blocks. In contrast, dynamic memory abstraction is concerned with modeling dynamic data structures that consist of multiple blocks referencing each other. The goal is to reason about these blocks' relationships and related abstract pointers. Unlike aggregate abstraction, dynamic memory abstraction also involves abstracting memory identifiers, enabling the abstraction of a set of blocks that a pointer can reference.

The forthcoming discussion will delve into the diverse approaches used for dynamic memory abstraction. In verification, memory representation techniques can be broadly classified into two categories: symbolic techniques, which employ SAT or SMT formulas, and abstract techniques. As we proceed with this chapter, we will primarily focus on a particular functor domain, namely, the pointer arithmetic domain, which allows us to reason about the layout of the dynamic memory. This domain was the primary focus of the associated publication with this chapter.

RQ1 How do dynamic memory domains interact with scalar and aggregate analyses?

RQ2 What is the relationship between pointer values and the real dynamic memory layout?

RQ3 How can we describe the ambiguity of memory layout without abstracting its content?

6.1 Symbolic Memory Models

While a fully symbolic model of a dynamic memory is not hard to implement, it is computationally expensive [Bal+18]. For this reason, some symbolic tools choose to trade soundness for performance by using under-approximations, e.g., address concretization.

[Bal+18]: Baldoni et al. (2018), "A survey of symbolic execution techniques"

A few sound approaches exist: the simplest is *state forking*, which considers all possible states that may result from a memory operation. More sophisticated, logic-based techniques, encode possibilities with `if-then-else` formulae or use the SMT theory of arrays [Bal+18]. Coarser representations restrict pointers to be either null or point to previously heap-allocated objects. Concolic executors DART [GKS05] or CUTE [SMA05]

[Bal+18]: Baldoni et al. (2018), "A survey of symbolic execution techniques"

[GKS05]: Godefroid et al. (2005), "DART: Directed Automated Random Testing"

[SMA05]: Sen et al. (2005), "CUTE: A Concolic Unit Testing Engine for C"

further under-approximate pointers to a single specific address or null, making constraint solving much more efficient: for instance, CUTE only reasons about pointer equality and hence can use a simple equivalence graph, instead of costly SMT queries. Such simplified models are incapable of relational reasoning about addresses.

To mitigate the scalability problems of fully symbolic memory and the loss of soundness due to memory concretization, *partial memory models* strike a balance between concretization and fully symbolic representation.

[Cha+12]: Cha et al. (2012), "Unleashing Mayhem on Binary Code"

For instance, Mayhem [Cha+12] concretizes addresses when they are used in writes, but otherwise keeps them symbolic as long as the contiguous interval of possible values they may access is sufficiently small. Mayhem additionally performs several optimizations like value-set analysis and caching to refine ranges efficiently. This technique is the most similar to the one we will present in the second half of this chapter, in the sense that it performs complicated memory operations in concrete form but retains symbolic relational reasoning.

[KPV03]: Khurshid et al. (2003), "Generalized Symbolic Execution for Model Checking and Testing"

[KK16]: Kanvar et al. (2016), "Heap Abstractions for Static Analysis"

A related approach is a *lazy initialization* of memory locations [KPV03]. In this technique, memory locations are initialized only after they are first accessed. Lazy initialization is usually used together with pre-generated heap configurations that model data-structures like trees or lists.

6.2 Abstract Memory Models

Because heap memory is potentially unbounded and seemingly arbitrary, we cannot employ the same techniques as with stack and static memory. Another key difference of memory abstractions to previous techniques is that memory permits self-referential values, i.e., abstraction needs to manage possibly infinitely recursive structures. Hence, a general heap abstraction consists of two components:

1. *heap modeling*, which takes care of heap memory representation as a memory model (i.e., an unbounded set of concrete locations as an unbounded set of abstract locations), and
2. *summarization*, which bounds the memory by merging multiple abstract locations into summary locations.

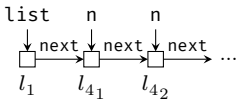
In the literature, we recognize multiple types of heap models: a *store-based model*, which represents memory as a graph with symbolic addresses, where edge $p \to l$ represents that a pointer p may hold the address of concrete location l (see Figure 6.1); on the contrary, a *storeless model* abstracts locations in terms of access paths and capture the structure of relations between heap objects [KK16].

In the memory-aware analysis, we may perform a separate abstraction of pointers. The simplest of them are non-relational pointer abstractions. A non-relational pointer abstraction is defined in the same way as a scalar abstraction; it is given as lattice of abstract pointer values. An example of simple non-relational pointer abstract domain is a *null pointer* domain illustrated in Figure 6.2.

Non-Relational pointer domains have a disadvantage because of their imprecision – as soon as a points-to set of pointers contain several elements,

```
Node *list = malloc(Node);
Node *n = list;
while (true) {
  n↠next = malloc(Node);
  n = x↠next;
}
```

Store based model of concrete heap of C previous program, where l_1, l_{4_1} and l_{4_2} denote allocation sites:



Allocation site based summarization:

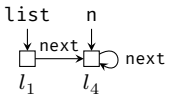


Figure 6.1: Heap summarization example.

Figure 6.2: The null pointer domain keeps information whether a pointer might be *null*:

$$\gamma_{ptr}(\top) \triangleq \mathcal{C}_{\mathsf{p}}$$
$$\gamma_{ptr}(\textit{non-null}) \triangleq \mathcal{C}_{\mathsf{p}} \backslash \{0\}$$

the abstract interpretation performs imprecise updates. Moreover, they are not able to express well-structured invariants about complex memory objects.

For more precise updates, pointer abstraction leverage summarization techniques, also known as *shape analysis*. Shape graphs are used to generalize abstract data structures (e.g., lists or trees). This technique is used in tools like Predator [DPV11] or CPAchecker [BK11]. To reason about unbounded data structures, memory graphs are usually combined with shape abstraction [DPV11]. Additionally, configurable analyses in CPAchecker may abstract memory and scalar values simultaneously. Hence, in theory, it should be able to reason about pointer values in a layout-sensitive fashion. Unfortunately, it treats each allocation separately and does not apply sufficient constraints to reason about heap re-orderings. The shape graph can summarize inductive data structures into simple shapes [DOY06]. Similarly to scalar analysis, shape analysis can be refined by widening or by keeping shape relations [CR08]. Reduced products to shape analysis was introduced in [TCR13].

[DPV11]: Dudka et al. (2011), "Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic"

[DOY06]: Distefano et al. (2006), "A local shape analysis based on separation logic"

[CR08]: Chang et al. (2008), "Relational inductive shape analysis"

[TCR13]: Toubhans et al. (2013), "Reduced Product Combination of Abstract Domains for Shapes"

More recent approaches leverage SMT solvers and reduce problems from separational logic, that is used in shape analysis, to propositional logics [Itz+14a; Itz+14b; PWZ13]. Also, template-based domains [Mal+18] delegates semantic reasoning to SMT solvers and focuses on the design of appropriate shape templates. In comparison to traditional predicate analysis where a user needs to supply a set of predicates, template-based domains use a set of parametrized predicates which are further gradually refined via SMT. Moreover, the SMT representation enables a straightforward combination with value analysis – hence allows reasoning about the shape and their contents at the same time.

[Itz+14a]: Itzhaky et al. (2014), "Modular reasoning about heap paths via effectively propositional formulas"

[Itz+14b]: Itzhaky et al. (2014), "Property-Directed Shape Analysis"

[PWZ13]: Piskac et al. (2013), "Automating separation logic using SMT"

[Mal+18]: Malík et al. (2018), "Template-Based Verification of Heap-Manipulating Programs"

To reason only about numeric properties of heap manipulating programs, we can omit a lot of shape analysis. Magill et al. present an abstraction [Mag+10] that instruments only numeric abstraction of shape properties into the program, hence allow cheaper reasoning about properties like a length of linked-lists or distance between two elements of the list. Other domains extend the relational numeric analysis of the content of particular shapes, for example, trees [JMO19].

[Mag+10]: Magill et al. (2010), "Automatic numeric abstractions for heap-manipulating programs"

[JMO19]: Journault et al. (2019), "An Abstract Domain for Trees with Numeric Relations"

Many tools specialize in the analysis of a single type of data structures. To name a few: Predator [Dud+12] is a tool for automated verification of programs with dynamic linked data structures using a separation logic and shape graphs; Forester [Hol+15] leverage tree automata in the combination with shape analysis to verify programs with various types of lists or trees; whereas Space Invader [Yan+08] use shape analysis in combination with abduction. For an extended survey of heap abstractions, we refer the reader to [KK16].

[Dud+12]: Dudka et al. (2012), "PREDATOR: A verification tool for programs with dynamic linked data structures"

[Hol+15]: Holík et al. (2015), "Forester: Shape Analysis Using Tree Automata"

[Yan+08]: Yang et al. (2008), "Scalable Shape Analysis for Systems Code"

[KK16]: Kanvar et al. (2016), "Heap Abstractions for Static Analysis"

6.3 Programs sensitive to Heap Layout

Most C and C++ programs, ranging from operating system kernels to web browsers, use dynamically organized memory to store data. The dynamic memory is usually maintained by the lowest levels of the runtime support built into the operating system, i.e. the standard C library. The abstraction is so pervasive and useful that almost all verification tools make use of

this additional structure, instead of treating memory strictly as a flat byte array. A common approach is to model memory using a graph, where the individual objects (pieces of data) stored in heap are the vertices and pointers stored in those objects are the edges. We will use this model in the rest of the chapter, since it most clearly demonstrates the issue which we are addressing.

Since the CPU still only provides a flat address space to programs, each memory object must be stored at some specific numeric address, and each pointer must be represented as a number. In normal execution, then, the mapping of memory objects (vertices) to concrete numeric addresses is performed by the `C` runtime and is not part of the program in question. Depending on the particulars of the execution platform, the mapping may end up being different between executions on different machines, or even between multiple executions on the same machine. Under normal circumstances, the different mappings are simply different representations of the same dynamic memory graph.

This abstraction is, however, imperfect, since the program may observe the specific numeric addresses assigned to individual pieces of data. We will refer to the specific operations involved in such observation as *ambiguous*[1] (a more formal definition will be given in Definition 6.4.2).

1: Remark the difference to ambiguity `amb` operator, which only serves to represent external input and is not related to ambiguity arising from memory layout.

The most common form of such an observation is sensitivity to the relative ordering of the numeric addresses of individual memory objects. Of course, under normal circumstances, this should not affect the high-level behavior of the program. Low-level details are, however, more likely to be affected: for instance, a number of programs compare pointers when storing them in some ordered data structure (sorted arrays, search trees and the like). In those cases, though, the intention is usually that the specific ordering of the pointers in the container does not affect higher layers of abstraction. Unfortunately, this is not always the case and it might be a source of bugs which are hard to track down and fix.

The particular effects that appear in real code-bases are modelled by programs shown in Figure 6.7. Specific `C` code which performs pointer comparisons, without being able to guarantee that the pointers involved point to the same memory object, is shown in Figure 6.3, and Figure 6.4 (Chromium), Figure 6.6 (Linux) and Figure 6.5 (LLVM).

Note that many operations on unrelated pointers, according to `C` and `C++` standards, yield undefined behavior. It is, however, allowed to cast pointers to their integer representation (`uintptr_t`, `intptr_t`) on which arithmetic and relational operations are well-defined. Similarly, the `C++` standard library defines relational functions (e.g., `std::less`) that guarantee a consistent total order on pointers for a particular execution.

Mapping C/C++ to LLVM

While pointer arithmetic is well defined in level of LLVM bitcode, where we perform our analysis, this is not the case for the intended high-level input languages, `C` and `C++`. According to the section 6.5.8 of the `C11` standard, the relational comparison of pointers is only defined if pointers point to the same array, structure, or object. In these cases,

the operations are unambiguous because only offset parts of pointers are compared. The same rules apply to arithmetic operations on pointers. Further, the addition and subtraction of pointer to an array and an integer is well defined only in the cases when a result points into or just after the array (section 6.5.6 of C11 standard). However, it is allowed to cast pointers to pointer integer types (`uintptr_t`, `intptr_t`) and perform integer arithmetic and relational operations on the cast values. In this case, compilers emit the aforementioned operations with corresponding semantics. Similarly, even though it is undefined to use the value of a pointer to an object whose lifetime has ended, one can capture the pointer value to `uintptr_t` before its lifetime ends and use it afterward.

Furthermore, according to the section 6.5.9 of C11 standard, equality comparison of pointers is well defined for compatible types, comparison to a null pointer constant, and comparisons to void pointers.

Additionally, the C++17 standard states in the section 23.14.7, *A specialization of* `std::less` *for any pointer type yields a strict total order, even if the built-in* `operator<` *does not. The strict total order is consistent among specializations of* `std::less`*,* `std::greater`*,* `std::less_equal`*, and* `std::greater_equal` *for that pointer type.* In the evaluation, we adhere only to the defined operations and explore the ambiguous behavior yielded by them.

In this chapter, we will focus on the defined cases, since in the undefined case, the compiler is free to alter the behavior of the program. Nonetheless, programs with undefined behavior are usually still handled correctly.

```
/* ios/chrome/browser/main/browser_list.h */
class BrowserList : public KeyedService {
    /* ... */
    virtual std::set<Browser*> AllRegularBrowsers() const = 0;
    virtual std::set<Browser*> AllIncognitoBrowsers() const = 0;
    /* ... */
};

/* ios/chrome/browser/sessions/ios_chrome_tab_restore_service_client.mm */
sessions::LiveTabContext* FindLiveTabContextWithCondition(
    base::RepeatingCallback<bool(Browser*)> condition
) {
    /* ... */
    for (/* ... */) {
        for (Browser* browser : browsers->AllRegularBrowsers()) {
            if (condition.Run(browser)) {
                return LiveTabContextBrowserAgent::FromBrowser(browser);
            }
        }
    }
    /* ... */
}
```

Figure 6.3: Code from Chromium 95.0.4627.1 exhibiting user-visible behavior which depends on the heap layout. The iteration order of `AllRegularBrowsers` depends on specific values of the pointers stored in the `std::set` instance. The first value (in this order) which matches a given criterion is returned.

```
/* chrome/browser/ui/tabs/tab_strip_model.cc;
   in the implementation of method TabStripModel::CloseWebContentses
   base::flat_map is an array of pairs sorted by their first element */
base::flat_map<content::RenderProcessHost*, size_t> processes;
for (content::WebContents* contents : items) {
    /* ... */
    content::RenderProcessHost* process = /* ... */;
    ++processes[process];
}
for (const auto& pair : processes)
    pair.first->FastShutdownIfPossible(pair.second, false);
```

Figure 6.4: Another example from Chromium 95.0.4627.1 where the specific pointer values cause changes in observable behavior (in this case, the order in which renderer processes are shut down).

```
/* lib/Transforms/Vectorize/VPlanSLP.cpp */
std::pair<VPlanSlp::OpMode, VPValue *>
VPlanSlp::getBest(/*...*/,  SmallPtrSetImpl<VPValue *> &Candidates, /*...*/) {
    SmallVector<VPValue *, 4> BestCandidates;

    /* Iteration order depends on numeric values of the contained pointers. */
    for (auto *Candidate : Candidates) {
        /* ... */
        if (areConsecutiveOrMatch(LastI, CandidateI, IAI))
            BestCandidates.push_back(Candidate);
    }
    /* ... */
    VPValue *Best = nullptr;
    unsigned BestScore = 0;
    for (unsigned Depth = 1; Depth < LookaheadMaxDepth; Depth++) {
        for (auto *Candidate : BestCandidates) {
            unsigned Score = getLAScore(Last, Candidate, Depth, IAI);
            /* Among values with the same score, the one selected is the first
               in Candidates iteration order, which depends on numeric values
               of unrelated pointers. */
            if (Score > BestScore) {
                BestScore = Score;
                Best = Candidate;
            }
        }
    }
    /* ... */
    Candidates.erase(Best);
    return {Mode, Best};
}
```

Figure 6.5: Ambiguous pointer operations in LLVM 9.0.1, in the autovectorizer, with possible observable consequences in the generated code.

```
/* arch/riscv/include/asm/sections.h */
extern char _start[];
extern char __init_data_begin[],
            __init_data_end[];
static bool is_va_kernel_text(
  uintptr_t va
) {
  uintptr_t start = _start;
  uintptr_t end = __init_data_begin;
  return va >= start && va < end;
}
```

```
/* drivers/scsi/csiostor/csio_wr.c */
void *wr = (void *)((uintptr_t)
  q->vstart + (q->cidx * q->wr_sz)
);
/* ... */
while (csio_is_new_iqwr(q, ftr)) {
  CSIO_DB_ASSERT(
    ((uintptr_t)wr + q->wr_sz) <=
     (uintptr_t)q->vwrap
  );
  /* ... */
}
```

Figure 6.6: Possibly ambiguous pointer operations in Linux version 4.19.205. Note that in the cases shown, the code is not *wrong*, but it is impossible to tell without analyzing the possible values of the pointers involved.

At the point of writing this thesis, I am not aware of any tools which would perform a sound analysis of program behavior dependent on physical heap layout. For instance, the bounded model checker CBMC [CKL04; KT14] limits instantiation of pointers to values that the pointer might get in previous assignments. The symbolic executor KLEE [CDE08] or Symbiotic [CSV19] combines concretization with state forking, foregoing any relational reasoning. Other tools, for example UAutomizer [HHP13] detect pointer comparisons and reject the program. In our approach, we aim to minimize state-space branching while retaining precise relational information about pointers.

[CSV19]: Chalupa et al. (2019), "Joint forces for memory safety checking revisited"

One of the tools that do track address ordering is 2LS, a static analyzer based on k-induction and shape analysis. For each allocation, 2LS creates a unique abstract memory object along with additional constraints on pointer comparisons which then allow more precise modelling of memory reuse [Mal+18; SK16]. As our evaluation shows, however, the model still fails to capture a number of phenomena correctly.

[Mal+18]: Malík et al. (2018), "Template-Based Verification of Heap-Manipulating Programs"
[SK16]: Schrammel et al. (2016), "2LS for Program Analysis"

Static Analysis

Static analysis tools like Coverity [ALS06] or Astrée [Cou+05] can detect some ambiguous operations on pointers (predominantly only pointer subtraction [JB23]). However, they are not able to reason about the consequent nondeterministic program behavior. Therefore, they only announce potentially ambiguous operations. Similarly, modern compilers provide sanitizer options to detect ambiguous operations, e.g., GCC compiler [Sta03] provides an option `-fsanitize=pointer-subtract` to produce warnings on subtraction of unrelated pointers. However, programmers sometimes use ambiguous operations intentionally, e.g., in the before-mentioned implementation of data structures or low-level code of operating systems and allocators implementations. In these cases, sanity detection is insufficient and exhaustive exploration of all possible executions is required.

[ALS06]: Almossawi et al. (2006), *Analysis tool evaluation: Coverity prevent*

[Cou+05]: Cousot et al. (2005), "The ASTREÉ Analyzer"

[JB23]: Johns et al. (2023), *Do not subtract or compare two pointers – SEI CERT C Coding Standard – Confluence*

[Sta03]: Stallman (2003), "The GCC developer community"

6.4 Heap Semantics

In the following section, we will build upon the semantics introduced in Chapter 3. Our focus will be on dynamic memory modeling and state equivalence, particularly from the perspective of ambiguous memory layouts. This information will later be used to automatically detect and refine the necessary set of pointers (allocations) to be abstracted while ignoring pointers that do not affect the soundness of the analysis.

Recall concrete semantics of dynamic memory allocation operation:

$$\sigma \xrightarrow{\mathtt{r_p} \longleftarrow \textbf{malloc}\ \mathtt{s}} \{(\varepsilon_s, \varepsilon_p[\mathtt{r_p} \mapsto \langle q, 0\rangle], \mu[q \mapsto \mathsf{block}\,(s)]) \mid q \in \mathsf{Id}\}$$

Thus far, we have assumed a fixed choice of a memory location given by q. However, the arbitrariness in the choice of the location q means that there is not a single result state σ' – instead, many are possible. In fact, too many to enumerate, and for this reason, the semantics of `malloc` are often under-approximated: for example, by only considering a single choice for the value of q. Nonetheless, there are instances when programs observe their own memory layout, rendering the analysis unsound if it fails to consider all possible layouts (cf. Figure 6.7).

Our goal is to analyze the state space induced by ambiguous allocations faithfully while keeping the state space as small as possible. To achieve this, we need to establish how states differ based on allocations. Rather than enumerating all possible layouts, we prefer to consider only one. And only if the observes its layout (allocation is used in relational operations) do we want to observe all possible outcomes.

Let us revisit the semantic domains related to memory layout. Specifically, in the context of heap layout abstraction, we are interested in memory block identifiers (Id). These identifiers indicate the position of the block within the flat memory.

Definition 6.4.1 *We say that two states σ and σ' are **equivalent** if they are equal up to a permutation of memory identifiers from* Id*, and they give rise to identical memory graphs.*

That is because, in the program analysis, we are mainly interested in state data (registers and memory content). Therefore, we want to avoid inspecting the non-determinism of allocations in the state equivalence. In other words, we consider two states equal if the memory content is equal up to the isomorphism of allocated places in the memory and the content of register maps is pairwise equal.

Definition 6.4.2 *We say that an operation* op *is **ambiguous** in state σ if an equivalent state σ' exists such that $\sigma \xrightarrow{\mathsf{op}} \tau$ and $\sigma' \xrightarrow{\mathsf{op}} \tau'$ but τ and τ' are not equivalent.*

To put it differently, a statement is ambiguous if it leads to inequivalent program states when applied to states with the same memory graph but different memory identifiers. Examples of such programs were demonstrated in Figure 6.7.

```
void *x = malloc(1);
void *y = malloc(1);
if (std::less{}(x, y))
    error();
```

```
void *x = malloc(1);
void *y = malloc(1);
intptr_t xv = intptr_t(x);
intptr_t yv = intptr_t(y);
ptrdiff_t len = yv - xv;
if (len > 100)
    error();
```

```
void *x = malloc(1);
uintptr_t xv = uintptr_t(x);
free(x);
void *y = malloc(1);
uintptr_t yv = uintptr_t(y);
if (xv == yv)
    error();
```

Figure 6.7: Examples of C/C++ programs with behavior dependent on the physical layout of the heap. In these programs, the reachability of an error location is dependent on the ambiguous behavior of pointer operations. Hence, programs might randomly fail when the allocator returns addresses in a specific order that trigger the ambiguous condition (or reuses a released address in the third case). Similar pointer operations often appear in ordered collections, e.g., in pointer instances of the standard C++ containers `set` or `map`.

Corollary 6.4.1 *Let σ_1 and σ_2 be a pair of equivalent states, and* op *an unambiguous operation which yields σ_1' and σ_2' when starting in σ_1 and σ_2 respectively. Then σ_1' and σ_2' are equivalent.*

Recall pointer-to-scalar operations from Section 3.3. This category includes operations that exclusively accept pointer arguments and always produce a scalar output. It includes operations similar to scalar relational operations but with pointer operands, as well as the subtraction of two pointers. Since results are scalars, only the scalar environment ε_s is updated, and since both operands are pointers, only the pointer environment ε_p is consulted.

$$(\varepsilon_s, \varepsilon_p, \mu) \xrightarrow{\mathsf{r}_s \leftarrow \mathsf{p}_1 - \mathsf{p}_2} (\varepsilon_s[\mathsf{r}_s \mapsto [\![\mathsf{p}_1]\!]_\sigma - [\![\mathsf{p}_2]\!]_\sigma], \varepsilon_p, \mu)$$
$$(\varepsilon_s, \varepsilon_p, \mu) \xrightarrow{\mathsf{r}_s \leftarrow \mathsf{p}_1 \diamond \mathsf{p}_2} (\varepsilon_s[\mathsf{r}_s \mapsto [\![\mathsf{p}_1]\!]_\sigma \diamond [\![\mathsf{p}_2]\!]_\sigma], \varepsilon_p, \mu)$$

where $\diamond \in \{<, >, =, \neq\}$. This category of operations is the source (and the only source) of ambiguous behavior that we seek to capture. This is because only the results of these operations are reliant on the actual value of memory identifiers.

However, there are also cases when operations from this category are *not* ambiguous.

Theorem 6.4.2 *Comparison of any pointer with a null pointer is unambiguous. Subtraction of two null pointers is unambiguous.*

Proof. There are two possible cases of a comparison with a null pointer:

1. both pointers are null pointers: there is no ambiguity, because the numeric value of a null pointer is always zero,
2. one of the pointers is not null: again, there is no ambiguity, since the result does not depend on the numeric value of the non-null pointer (the only pointer with numeric value 0 is the null pointer). For instance, `p > nullptr` always evaluates to `true`. Analogously for all other relational operators.

□

Theorem 6.4.3 *Comparisons of pointers which point to the same memory object are unambiguous, as is the result of subtracting them.*

Proof. Each valid pointer p can be uniquely decomposed into a sum of two parts: the *base address* b of the object into or after which it points, and an *offset* $o \geq 0$. That is, $p = b + o$ (or equivalently, $o = p - b$). The standard rules of modular arithmetic apply (overflow can be neglected, since $o \geq 0$ implies that $p \geq b$). Crucially, only the base address is affected by ambiguity: this is the address at which the memory object was placed by the `malloc` operation. Clearly, different choices of the base address lead to equivalent states, though not identical.

However, if two pointers point to the same memory object, their base address must be the same (since it is a property of the memory object), hence they are of the form $p_1 = b + o_1$ and $p_2 = b + o_2$. Their difference, $p_1 - p_2 = b + o_1 - b - o_2 = o_1 - o_2$ clearly does not depend on the value of b.

The argument for comparisons is analogous: assuming $\diamond$ is any relational operator, $b + o_1 \diamond b + o_2$ is equivalent to $o_1 \diamond o_2$ and again, there is no dependence on the value of b. □

Theorem 6.4.4 *Equality comparison (the $=$ and $\neq$ operators) on pointers which point to different objects is unambiguous iff both pointers are within object bounds.*

Proof. The pointers decompose as $p_1 = b_1 + o_1$ and $p_2 = b_2 + o_2$, with $b_1 \neq b_2$. Let s_1 and s_2 be the sizes of the respective objects. There are two cases to consider:

1. if both pointers are within bounds, that is, $o_1 < s_1 \wedge o_2 < s_2$, there is no ambiguity, since the objects cannot overlap and the pointers always compare unequal,
2. without loss of generality, assume that $o_1 \geq s_1$ and further assume that $b_1 < b_2$ (this is always possible) – in this case, the comparison is equivalent to $o_1 = b_2 - b_1$, which is clearly an ambiguous statement under the assumptions.

□

Theorem 6.4.5 *All other pointer-to-scalar operations are ambiguous.*

Proof. In this case, $p_1 = b_1 + o_1$ and $p_2 = b_2 + o_2$, while $b_1 \neq b_2$. There are two sub-cases:

1. either $b_1 = 0$ or $b_2 = 0$ and the operation is subtraction, in which case the result is simply the numeric value of the non-null pointer, or its negative, both of which are arbitrary, or
2. both b_1 and b_2 are arbitrary and distinct and the operation is either ordering or subtraction (other cases being covered by preceding theorems); then clearly $b_1 - b_2$ is also arbitrary, and ordering reduces to $b_1 - b_2 > 0$.

□

6.5 Pointer Arithmetic Domain

The underlying concept behind the proposed abstract domain is to disassociate memory access operations from arithmetic operations on pointers. This decoupling is primarily motivated by the fact that it enables the model checker to continue utilizing whichever convenient under-approximation it prefers for representing pointers and memory content.

For this reason, the domain maintains two versions of the pointer, stored side by side as a product of:

1. the *dereference* part: the original concrete representation, which is used for memory access (loads, stores), and
2. the *numeric* part: an additional value which is used whenever scalars are derived from pointers (i.e., in relational operators and pointer subtraction).

The *dereference* and *numeric* components of the abstract pointer must remain synchronized, which is the responsibility of the operations specified by the domain. These operations fall into four categories:

1. creating and destroying pointers (memory allocation),
2. updating pointers (pointer arithmetic),
3. deriving scalars from pointers (subtraction of two pointers)
4. memory access (load and store).

Parametric Formulation

The pointer arithmetic domain $\widehat{\mathsf{Pa}}$, in its entirety, is a functor domain. Its operations ensure the synchronization of the two components of the pointer abstraction and determine which domain is utilized to execute memory access or relational reasoning. Nonetheless, these operations are agnostic of the specific domains in use. In fact, we can regard the pointer domain as a product domain of two value domains: dereference and numeric domain.

- $\widehat{\mathsf{D}}$, the *dereference domain* which will be used to represent the dereference part of the pointer (in our current implementation, this is simply the concrete domain $\mathcal{C}$)[2],
- $\widehat{\mathsf{N}}$, the *numeric domain* which will be used to represent the numeric part (this is taken to be $\mathcal{T}$, the term domain, in our implementation).

2: Nevertheless, it can be instantiated with an abstract aggregate domain that also abstracts the memory's contents.

An element of the functor domain $\widehat{\mathsf{Pa}}(\widehat{\mathsf{D}}, \widehat{\mathsf{N}})$ is therefore a tuple $\langle d, n \rangle$ where $d \in \widehat{\mathsf{D}}$ and $n \in \widehat{\mathsf{N}}$. There are no special requirements for the $\widehat{\mathsf{D}}$ domain, other than that it is a value domain with the requisite operations on pointers (memory allocation, addition of scalars to pointers, etc.). The domain $\widehat{\mathsf{N}}$ is expected to provide scalar operations (in the sense of Section 3.3 – particularly addition, subtraction and comparisons).

Constraining the Numeric Part

The goal of the $\widehat{\mathsf{Pa}}$ domain is to model pointer comparison as accurately as possible. While there is significant freedom in choosing numeric values of pointers, they are not completely arbitrary: the natural properties of pointers give rise to certain constraints. For instance, two pointers which point to distinct memory objects with overlapping lifetimes must not compare equal (more examples are given in Figure 6.8).

For this reason, we need to be able to constrain the numeric part of the result of `malloc` such that it does not collide with any part of an existing live object. This places an additional requirement on $\widehat{\mathsf{N}}$: it needs to provide an `assume` operation which constrains its values so that they satisfy a relational predicate. For example, given $x, y \in \widehat{\mathsf{N}}$, the statement `assume(x < y)` must be valid.

There are 3 possible outcomes for $\widehat{\mathsf{Pa}}(\widehat{\mathsf{D}}, \widehat{\mathsf{N}})$, depending on the behavior of `assume` provided by $\widehat{\mathsf{N}}$:

1. if `assume` may over-constrain values, $\widehat{\mathsf{Pa}}(\widehat{\mathsf{D}}, \widehat{\mathsf{N}})$ cannot be sound and it yields an under-approximation (real errors may be missed in analysis),
2. if `assume` may under-constrain values, $\widehat{\mathsf{Pa}}(\widehat{\mathsf{D}}, \widehat{\mathsf{N}})$ yields a sound over-approximation (spurious errors may appear during analysis),
3. if `assume` exhibits neither of the above behaviors, $\widehat{\mathsf{Pa}}(\widehat{\mathsf{D}}, \widehat{\mathsf{N}})$ is *exact* – it is (in theory) possible to find all errors, and all errors which are found are real (feasible). The term domain $\mathcal{T}$ falls into this category.

6.5.1 Operation Semantics

In this section, we revisit the operations defined in Section 3.3, giving them appropriate semantics within the $\widehat{\mathsf{Pa}}$ domain. To that end, we first need a definition of the abstract state $\hat{\sigma}$. The abstract semantic domains are affected as follows:

```
1  x <- malloc(4)
2  y <- malloc(4)
3  if (y == nullptr)
4      /* ... */
5  store 7 -> x
6  v <- load x of i32
7  if (x < y)
8      /* ... */
9  free(y)
10 z <- malloc(4)
11 len <- x - z
12 if (len > 10)
13     /* ... */
```

The first allocation in the program is unconstrained, `x` *can refer to any location without limitation.*

Subsequent allocations cannot reference the same exact location as prior allocations, and allocated entities cannot have any overlap. This means that the pointer `y` *cannot have an identical value to* `x`*, and the memory interval* `[y, y + 4)` *must not intersect with the interval* `[x, x + 4)`*.*

Comparing a pointer with `nullptr` *always yields a deterministic outcome, regardless of the pointer's value. As a result, such a comparison avoids any potential ambiguity in the result.*

For memory access, like `store` *or* `load`*, the numeric value of the pointer is unimportant. These instructions behave the same regardless of the numeric value of the pointer* `x`*.*

Relational comparison of two heap pointers is ambiguous since the result depends on their numeric values, and those may differ between runs.

After this point, the allocator may reuse the address of the freed object.

A newly allocated object `z` *must not overlap with live object* `x`*, but it may reuse the space previously occupied by* `y`*.*

Subtraction of pointers is also ambiguous since the result depends on the numeric values of the operands.

Branching on an ambiguous value results in nondeterministic control flow.

Figure 6.8: A program which illustrates constraints on numeric values of pointers. These situations depicted are modelled accurately by the $\widehat{\mathsf{Pa}}$ domain.

$$
\begin{aligned}
\hat{p} \in \widehat{\mathsf{Pa}} &\equiv \widehat{\mathsf{D}} \times \widehat{\mathsf{N}} && \text{(pointer arithmetic domain)} \\
\hat{\varepsilon}_s \in \widehat{\mathbb{Env}}_{\mathsf{s}} &\equiv \mathsf{Reg}_{\mathsf{s}} \to \mathcal{C}_{\mathsf{s}} \cup \widehat{\mathsf{N}} && \text{(scalar registers environment)} \\
\hat{\varepsilon}_p \in \widehat{\mathbb{Env}}_{\mathsf{p}} &\equiv \mathsf{Reg}_{\mathsf{p}} \to \mathcal{C}_{\mathsf{p}} \cup \widehat{\mathsf{Pa}} && \text{(pointer registers environment)} \\
\langle s, \hat{v} \rangle \in \widehat{\mathsf{Blk}} &\equiv \mathcal{C}_{\mathsf{s}} \times \hat{\mathbb{V}} && \text{(memory regions)} \\
\hat{v} \in \hat{\mathbb{V}} &\equiv \mathcal{C}_{\mathsf{s}} \to \mathcal{C} \cup \widehat{\mathsf{N}} \cup \widehat{\mathsf{Pa}} && \text{(memory region content)} \\
\hat{\mu} \in \widehat{\mathbb{Env}}_{\mu} &\equiv \mathsf{Id} \to \widehat{\mathsf{Blk}} && \text{(memory environment)} \\
\lambda \in \Lambda &\equiv \mathsf{Id} \times \widehat{\mathsf{N}}
\end{aligned}
$$

Figure 6.9: Abstract semantic domains for pointer arithmetic abstraction.

To summarize, the $\widehat{\mathsf{Pa}}$ domain extends the set of admissible pointer values. We allow storing pointer and numeric abstract values in respective registers or memory blocks. Unlike aggregate abstraction, we do not abstract memory offsets. Note that the dereference mapping needs to define a projection $id : \widehat{\mathsf{D}} \to \mathsf{Id}$ since memory is still represented concretely. In our case, $\widehat{\mathsf{D}}$ is equivalent to the set of concrete pointers $\mathcal{C}_{\mathsf{p}}$, meaning we can access memory as in a concrete execution. One could also define abstract memory as $\mathsf{Id} \cup \widehat{\mathsf{D}} \to \widehat{\mathsf{Blk}}$.

An abstract state is then $\hat{\sigma} = (\hat{\varepsilon}_s, \hat{\varepsilon}_p, \hat{\mu}, \lambda)$ where the new element of the state, $\lambda \subseteq \mathsf{Id} \times \widehat{\mathsf{N}}$, records the address range constraints assigned to live objects (or rather to the numeric parts of pointers to such objects). Essentially, elements of λ keep a range of possible addresses as an abstract value in a domain $\widehat{\mathsf{N}}$, for example, in an interval domain. Using elements

of λ, we will later generate constraints on newly allocated objects to not overlap in the abstracted memory layout but still represent all admissible layouts.

Memory allocation

The allocation of size s in the $\widehat{\mathsf{Pa}}$ domain wraps allocation from dereference domain $d \triangleq \mathbf{malloc}_{\widehat{\mathsf{D}}}\ s$. It is then the responsibility of $\widehat{\mathsf{Pa}}$ domain to generate constraints on the numeric part of the pointer $\widehat{\mathsf{N}}$. As we have discussed in the previous section, to obtain an *exact* domain, we must ensure that objects with overlapping lifetimes are given non-overlapping addresses. The numeric part of the pointer needs to reflect this. Memory allocation, therefore, takes the following form:

$$\hat{\sigma} \xrightarrow{\mathtt{r_p} \longleftarrow \mathbf{malloc}_{\widehat{\mathsf{Pa}}}\ \mathtt{s}} \{(\hat{\varepsilon}_s, \hat{\varepsilon}_p[\mathtt{r_p} \mapsto \langle d, n\rangle], \hat{\mu}, \lambda \cup \{(\mathit{id}(d), n)\})\}$$

where n is arbitrary except as constrained below, and $\hat{\varepsilon}_s$, $\hat{\varepsilon}_p$ and $\hat{\mu}$ are determined by a sequence of `assume` operations starting from $(\hat{\varepsilon}_s, \hat{\varepsilon}_p, \hat{\mu})$, one for each $(i_\lambda, n_\lambda) \in \lambda$, of the form:

$$\mathtt{assume}(n + \mathit{size}(d) < n_\lambda \vee n_\lambda + \mathit{size}(i_\lambda) \leq n)$$

where $\mathit{size}(i)$ refers to the size of the memory block identified by $i \in \mathsf{Id}$. Figure 6.10 depicts how the generated constraints capture all possible object orderings.

It should be noted that the responsibility of memory block creation does not lie with the $\widehat{\mathsf{Pa}}$ domain, as the allocation in $\widehat{\mathsf{D}}$ has already created a new memory block.

Memory deallocation, through the `free` operation, proceeds analogously (except there are no constraints involved):

$$(\hat{\varepsilon}_s, \hat{\varepsilon}_p, \hat{\mu}, \lambda) \xrightarrow{\mathbf{free}\ \mathsf{p}} \{(\hat{\varepsilon}'_s, \hat{\varepsilon}'_p, \hat{\mu}', \{(i_\lambda, n_\lambda) \mid (i_\lambda, n_\lambda) \in \lambda, i_\lambda \neq \mathit{id}(d)\})\}$$

where $[\![\mathsf{p}]\!]_{\hat{\sigma}} = \langle d, n\rangle$ and $\hat{\varepsilon}'_s, \hat{\varepsilon}'_p$ and $\hat{\mu}'$ are determined by `free` from the domain $\widehat{\mathsf{D}}$.

Pointer-to-Pointer Operations

Operations that modify the value of a pointer need to be performed on both parts of the $\widehat{\mathsf{Pa}}(\widehat{\mathsf{D}}, \widehat{\mathsf{N}})$ representation, in order to keep them synchronized. Clearly, the outcomes of both a `load` or a `store` (which consult only the dereference part of the pointer) and of any pointer-to-scalar operation (which consult only the numeric part) are affected by pointer updates. If $[\![\mathsf{p}]\!]_{\hat{\sigma}} = \langle d, n\rangle$ and $[\![\mathsf{s}]\!]_{\hat{\sigma}} = \mathit{off}$, the semantics are given by:

$$\hat{\sigma} \xrightarrow{\mathtt{r_p} \longleftarrow \mathtt{p + s}} \{(\hat{\varepsilon}_s, \hat{\varepsilon}_p[\mathtt{r_p} \mapsto \langle d + \mathit{off}, n + (\alpha_{\widehat{\mathsf{N}}} \circ \gamma_{\widehat{\mathsf{D}}})(\mathit{off})\rangle], \hat{\mu}, \lambda)\}$$

In the implementation, $\alpha_{\widehat{\mathsf{N}}} \circ \gamma_{\widehat{\mathsf{D}}}$ is expected to be implemented as a single, comparatively efficient operation (i.e., it would not enumerate all possible concrete values of $\hat{\varepsilon}_s(\mathsf{s})$).

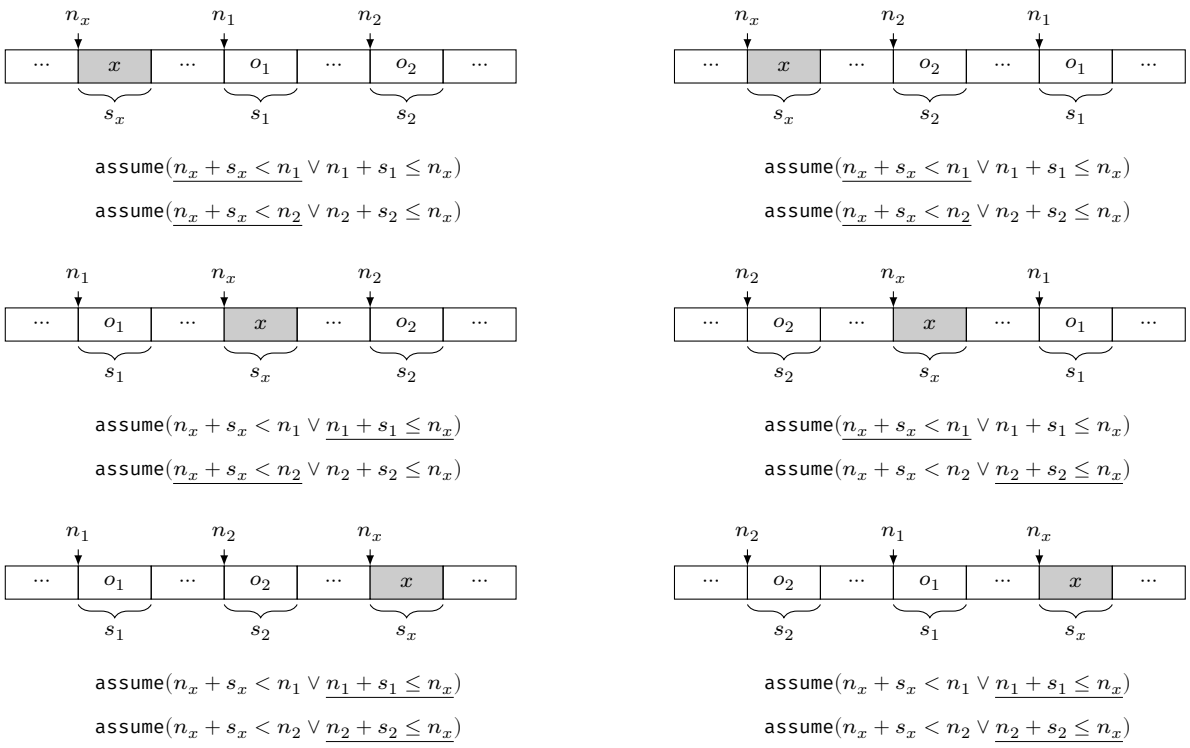


Figure 6.10: An example of constrained allocation of object x of size s_x in the situation when there are already two allocated objects o_1 and o_2, with respective sizes s_1 and s_2. Abstract pointers to these objects are $\hat{\varepsilon}_p(p_x) = \langle d_x, n_x \rangle$, $\hat{\varepsilon}_p(p_1) = \langle d_1, n_1 \rangle$ and $\hat{\varepsilon}_p(p_2) = \langle d_2, n_2 \rangle$, where d_i is a dereference part of a pointer, and n_i is an abstract description of memory location in domain $\widehat{\mathsf{N}}$. In the abstraction, we describe the relations between allocated objects via generated constraints (`assume`). For each new allocation, there is always one constraint per object with an overlapping lifetime. In the case of allocation of object x there are two constraints generated from live objects o_1 and o_2. These two constraints describe all six possible orderings of the three objects. The figure depicts these six cases – the parts of the constraints which describe the depicted case are underlined.

Pointer-to-scalar Operations

This category includes subtraction of two pointers and relational operations on pointers. The result is always a scalar, while all operands are always pointers. Like above, $\alpha_{\widehat{\mathsf{D}}} \circ \gamma_{\widehat{\mathsf{N}}}$ is expected to have an efficient implementation. First we give the general form of the operations (with $[\![\mathsf{p}_1]\!]_{\hat{\sigma}} = \langle d_1, n_1 \rangle$ and $[\![\mathsf{p}_2]\!]_{\hat{\sigma}} = \langle d_2, n_2 \rangle$):

$$\hat{\sigma} \xrightarrow{\mathsf{r_s} \longleftarrow \mathsf{p}_1 - \mathsf{p}_2} \{(\hat{\varepsilon}_s[\mathsf{r_s} \mapsto (\alpha_{\widehat{\mathsf{D}}} \circ \gamma_{\widehat{\mathsf{N}}})(n_1 - n_2)], \hat{\varepsilon}_p, \hat{\mu}, \lambda)\}$$
$$\hat{\sigma} \xrightarrow{\mathsf{r_s} \longleftarrow \mathsf{p}_1 \diamond \mathsf{p}_2} \{(\hat{\varepsilon}_s[\mathsf{r_s} \mapsto (\alpha_{\widehat{\mathsf{D}}} \circ \gamma_{\widehat{\mathsf{N}}})(n_1 \diamond n_2)], \hat{\varepsilon}_p, \hat{\mu}, \lambda)\}$$

Additional optimization can be achieved in this situation. If $\widehat{\mathsf{D}}$ confirms that both pointers refer to the same object, the operations can be carried out in $\widehat{\mathsf{D}}$ (which is a concrete domain in our case) as stated by Theorem 6.4.3:

$$\hat{\sigma} \xrightarrow{\mathsf{r_s} \longleftarrow \mathsf{p}_1 - \mathsf{p}_2} \{(\hat{\varepsilon}_s[\mathsf{r_s} \mapsto d_1 - d_2], \hat{\varepsilon}_p, \hat{\mu}, \lambda)\}$$
$$\hat{\sigma} \xrightarrow{\mathsf{r_s} \longleftarrow \mathsf{p}_1 \diamond \mathsf{p}_2} \{(\hat{\varepsilon}_s[\mathsf{r_s} \mapsto d_1 \diamond d_2], \hat{\varepsilon}_p, \hat{\mu}, \lambda)\}$$

Memory Access Operations

These are the `load` and `store` instructions, which are not affected by the ambiguity of the identifier. They are therefore entirely realized in the domain $\widehat{\mathsf{D}}$. Letting $[\![\mathtt{r_p}]\!]_{\hat{\sigma}} = \langle d, n\rangle \in \widehat{\mathsf{Pa}}$:

$$\hat{\sigma} \xrightarrow{\textbf{store}\ \mathtt{v} \longrightarrow \mathtt{r_p}} \{(\varepsilon_s, \varepsilon_p, \mu[id(d) \mapsto \mathsf{store}^{\widehat{\mathsf{D}}}_{\hat{\sigma}}([\![d]\!]_{\hat{\sigma}}, [\![\mathtt{v}]\!]_{\hat{\sigma}})], \lambda)\}$$

where $\mathsf{store}_{\hat{\sigma}}([\![d]\!]_{\hat{\sigma}}, [\![\mathtt{v}]\!]_{\hat{\sigma}})$ is store defined by $\widehat{\mathsf{D}}$, which is in the case of a concrete domain defined as in Chapter 3 or as an abstract store from aggregate abstraction defined in Chapter 5.

Likewise, we perform `load` only on dereference part of the domain. In this case $[\![\mathtt{r}_a]\!]_{\hat{\sigma}} = \langle d, n\rangle \in \widehat{\mathsf{Pa}}$:

$$\hat{\sigma} \xrightarrow{\mathtt{r_s} \longleftarrow \textbf{load}\ \mathtt{r}_a\ \textbf{of}\ \mathtt{ty}} \{(\hat{\varepsilon}_s[\mathtt{r_s} \mapsto \mathsf{load}^{\widehat{\mathsf{D}}}_{\hat{\sigma}}[\![d]\!]_{\hat{\sigma}}, \mathsf{sizeof}\,(\mathtt{ty}))], \varepsilon_p, \mu, \lambda)\}$$
$$\hat{\sigma} \xrightarrow{\mathtt{r_p} \longleftarrow \textbf{load}\ \mathtt{r}_a\ \textbf{of}\ \mathtt{ty}} \{(\hat{\varepsilon}_s, \hat{\varepsilon}_p[\mathtt{r_s} \mapsto \mathsf{load}^{\widehat{\mathsf{D}}}_{\hat{\sigma}}([\![d]\!]_{\hat{\sigma}}, \mathsf{sizeof}\,(\mathtt{ty}))], \mu, \lambda)\}$$

This produces an efficient abstraction for memory access, as we perform concrete memory `load` and `store`, which minimizes the impact of heap layout abstraction on the abstract program execution.

Recall the short examples from Figure 6.7, transcribed to our LLVM-like language in Figure 6.11. These programs abstracted to the $\widehat{\mathsf{Pa}}$ domain give rise to the depicted execution graphs. For each allocation, constraints are generated as described in $\widehat{\mathsf{Pa}}$ domain abstact semantics. Notice that pointer `y` is not constrained in the third case because `x` does not point to a live object – it is not present in λ at the moment of allocation. Furthermore, notice that in the case of a concrete execution, only one path is executed in each case, while the $\widehat{\mathsf{Pa}}$ domain allows us to explore all possible program paths.

6.5.2 Pointer Arithmetic for Symbolic Execution

In our evaluation, we instantiate the $\widehat{\mathsf{Pa}}$ domain as $\widehat{\mathsf{Pa}}(\mathcal{C}, \mathcal{T})$ (where $\mathcal{C}$ is the concrete and $\mathcal{T}$ is the term domain, as described in Section 4.3.6). Since the verifier uses an SMT solver for bit-vector logic to interpret the terms, the $\widehat{\mathsf{Pa}}(\mathcal{C}, \mathcal{T})$ domain is functionally equivalent to symbolic execution with bit-precise reasoning about pointer relations. Unlike most other symbolic approaches, however, ours does not require the SMT solver to support the theory of bit-vector arrays, since memory access is performed in the concrete domain.

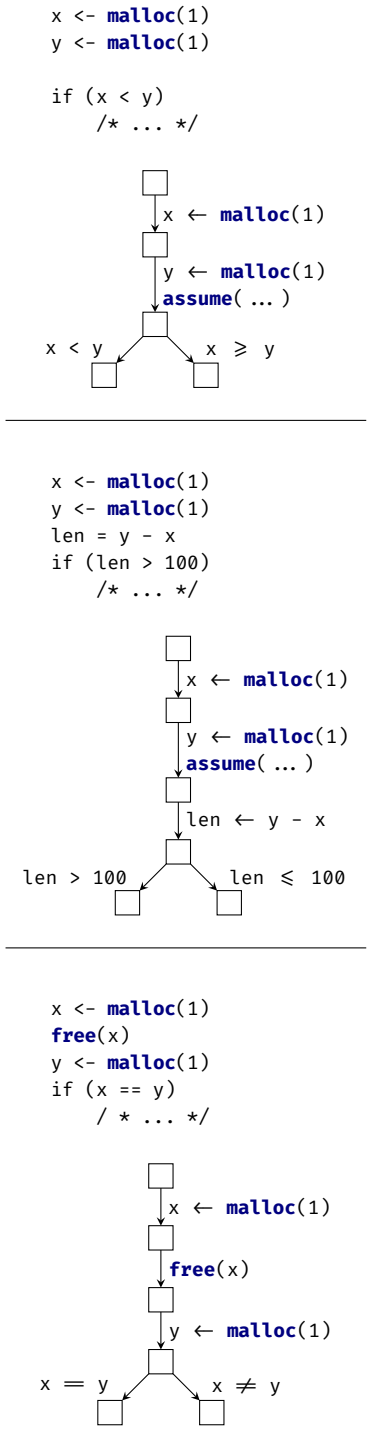


Figure 6.11: Examples of execution using $\widehat{\mathsf{Pa}}$ domain.

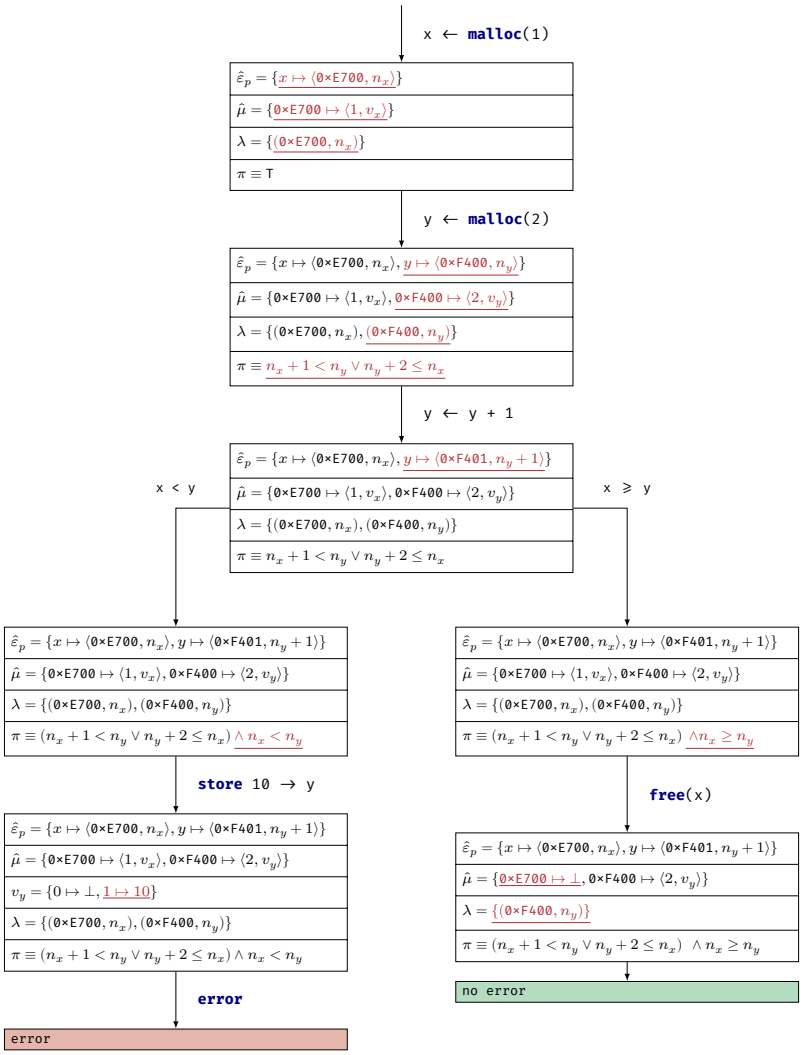


Figure 6.12: The execution graph illustrates the use of symbolic execution with the $\widehat{\text{Pa}}(\mathcal{C}, \mathcal{T})$ domain of the following program:

```
x ← malloc(1)
y ← malloc(2)
y ← y + 1
if x < y
    store 10 → y
    error
else
    free(x)
```

In Figure 6.12, we present an execution graph that demonstrates the use of symbolic execution with the $\widehat{\text{Pa}}(\mathcal{C}, \mathcal{T})$ domain. We indicate each change to the state using colored underlined notation. The path condition π of the program is determined by the control flow and `assume` statements in the term domain $\mathcal{T}$. Pointers are depicted as 16-bit numbers for ease of comprehension. It should be noted that both possible executions are examined, and both program allocations occur in the $\widehat{\text{Pa}}(\mathcal{C}, \mathcal{T})$ domain. The execution performs the following operations for each line:

1. `x ← malloc(1)`: the first allocation of size 1 returns a dereference object at address `0×E700` and term
2. `x ← malloc(2)`: the second allocation allocates 2 bytes of memory with dereference part at address `0×F400` and constraints the term representation n_y to not overlap with the live objects stored in λ. The constraint is added to path condition π.
3. `y ← y + 1`: the addition is performed on both dereference and term part of the abstract pointer $\langle \texttt{0×F400}, n_y \rangle$.
4. `x < y`: a nondeterministic choice is taken on the ambiguous comparison.

5. **store** `10` → `y`: a memory access operation uses a dereference part of the pointer – concrete address `0×F401` – to store the constant 10.
6. **free**(`x`): deallocation removes the object from λ and clears the memory map $\hat{\mu}$.

Given that $\widehat{\text{Pa}}$ is specified as a parametric domain, it is possible to obtain less precise but more cost-effective analysis from alternative instances. Some suitable candidates would be the *linear equality* domain [Kar76] that captures only information about affine relationships among program variables, or the *octagon* domain [Min06c] that reasons about invariants of the form $\pm x \pm y \leq c$, where x and y are program variables and c is a constant, or even a *predicate* domain, constructed on-demand using predicate abstraction [BBM97].

[Kar76]: Karr (1976), "Affine relationships among variables of a program"

[Min06c]: Miné (2006), "The octagon abstract domain"

[BBM97]: Bjørner et al. (1997), "Automatic generation of invariants and intermediate assertions"

As a future line of research, we propose to study the integration of the pointer arithmetic domain with other symbolic memory models, respectively aggregate domains, which support reasoning about unbounded memory. In a different direction, the domain can be extended to better handle pointer fragmentation.

Despite the benefits of the approach, it still requires abstracting all program allocations to ensure soundness. However, in many real-world scenarios, such extensive abstraction may not be necessary. Therefore, in Chapter 9 on syntactic abstraction, we will introduce a refinement loop that abstracts memory allocation on demand. For further information about the pointer arithmetic domain, we refer the reader to our evaluation in Appendix D, which demonstrates that state-of-the-art tools often underapproximate heap layout analysis. We also showcase the effectiveness of our approach.

Summary

This chapter introduces the concept of dynamic memory abstraction, which differs from scalar and aggregate abstraction in that it focuses on abstracting memory identifiers. We have explored various relevant approaches to this topic, notably the different symbolic and abstract memory models and their limitations. By combining dynamic memory abstraction with previously discussed scalar and aggregate abstractions, we can finally abstract all program state components.

Furthermore, we have discussed the effect of physical dynamic memory layout on the behavior of programs and identified *ambiguous pointer comparisons* as the underlying cause of unsound analyses. To overcome the challenges this causes in the program verification, we have proposed a new *pointer arithmetic* abstract domain. Compared to other techniques, the pointer arithmetic domain, together with symbolic representation, soundly keeps track of pointer relations during the course of analysis.

7 Metadomains & Refinement

To perform comprehensive and efficient analysis, different abstractions are necessary to be employed in the single system under test based on the specificities of the program data. Previous chapters have discussed different approaches to various abstraction possibilities, including: a) scalar abstraction, which focuses on abstracting integer and floating-point values using numeric abstract domains, b) aggregate abstraction, which abstracts specific memory blocks, and c) dynamic memory (pointer) abstraction, which reasons about multiple memory blocks and their relationships, were also discussed as memory-related abstraction categories. Each abstraction category requires a slightly different analysis approach. However, the analyses must be able to work with combinations of domains and facilitate transfers between them. For instance, when accessing an abstract array, the result might be an abstract scalar. In this chapter, we will delve into the interaction between these categories and discuss how various domains can interact in a program while being adapted to different analysis algorithms. It is worth noting that although this research has not yet been published, it forms an essential part of the overall approach and is embedded in the implementation of all the discussed abstractions, including M-String and pointer arithmetic domain.

We propose to resolve domain interaction using a so-called *metadomain*, which serves as a domain of domains and handles their interactions. Essentially, metadomain is a "sum" domain, similar to the product domain, but instead of representing a value in multiple domains simultaneously, it represents a value in a single domain chosen from a set of available domains and resolves their interactions.[1] The goal of this approach is to handle domain interactions in a unified manner, without exposing the conversion mechanism to all domains. Instead, the responsibility of managing domain interactions and conversions is delegated to the metadomain.

1: This distinction in domains is analogous to the one made in type theory between sum types (unions) and product types (tuples).

Treating values from different sources may be advantageous using different abstract domains. However, when these values interact during execution, the abstraction needs to resolve the computation in a single domain. Due to that, another direction worth exploring is how to convert uniformly between abstract domains of the same category.

In programs that utilize multiple domains, a challenge is applying them effectively, determining when to use specific domains, and refining them statically or dynamically during runtime. Common techniques involve employing counterexample-guided abstraction and refinement (CEGAR) to deal with these cases. In summary, our objective is to reduce the computational demands of abstraction while maintaining its precision. In cases where the abstraction is found to be insufficiently precise, we aim to refine it to achieve a higher level of accuracy. For example, in cases where we need to reason about relations, it may not be necessary to use a computationally demanding bit-precise term domain that involves theorem proving. Instead, we can employ more efficient domains, such as affine or octagon domains, and only resort to employing term domains when infeasible counterexamples are detected.

Additionally, refinement techniques from the SMT community, such as program variable bitwidth abstraction [JS18], theory refinement [Hyv+17], or lazy abstraction approach [Hen+02], can be adapted in conjunction with CEGAR further to enhance the effectiveness of domain abstraction in program analysis.

This chapter will focus on interdomain refinement techniques, in contrast to the intradomain refinement techniques discussed in Chapter 4, specifically backward constraint propagation. In the following sections, we provide a comprehensive overview of various refinement techniques that ultimately lead to meta-domain analysis and its refinement. Our objective is to address the following tasks:

RQ1 How to uniformly perform multi-domain analysis that encompasses scalar, aggregate, and pointer domains?

RQ2 How can domain interactions be effectively managed without the necessity of defining conversions between all domains?

RQ3 What is the relationship between common interdomain refinement techniques and their applicability to our program analysis?

7.1 Refinement Techniques

In verification, it is crucial to preserve the soundness of the analyses. To achieve this, the abstraction must maintain all possible paths leading to an error, even if it means introducing additional paths. We call such abstraction over-approximating, meaning that if an abstract model is correct, the concrete model also is. Nevertheless, the inclusion of new paths in the abstraction makes it possible that the error state reachable in the abstract model may not be reachable in the original concrete model, resulting in errors known as "spurious".

7.1.1 Counterexample-guided Abstraction Refinement

[Cla+00]: Clarke et al. (2000), "Counterexample-Guided Abstraction Refinement"

A common technique to handle spurious errors (counterexamples) is counterexample-guided abstraction refinement (CEGAR), first presented by Clarke et al. in [Cla+00]. The idea of the CEGAR approach is to examine a counterexample and if it is spurious, refine the abstraction in such a way that the spurious error would become unfeasible. Subsequently, the analysis is restarted with refined abstraction, which is more precise and avoids the spurious error (see Figure 7.1).

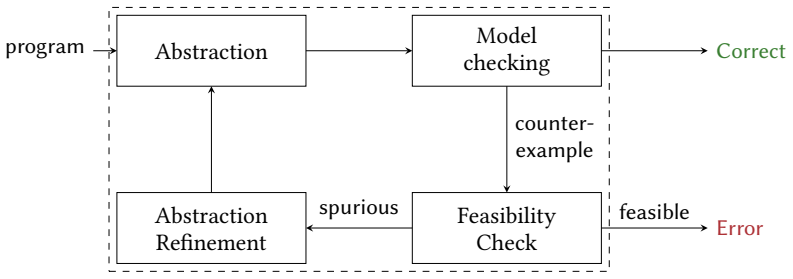


Figure 7.1: Counterexample guided abstraction refinement loop.

The key challenge in CEGAR is how to refine the abstraction effectively. In predicate-based abstraction techniques, refinement involves the introduction of new predicates. Furthermore, to obtain more precise information, refinement predicates are often generalized using interpolation techniques [JM06] (cf. Section 7.1.3). Predicate abstraction in combination with CEGAR has proven to be a highly successful abstraction technique for software model checking, as its symbolic state representation integrates well with strongest post-condition calculations, and the refinement process can be efficiently computed using SMT solvers [BL13].

[JM06]: Jhala et al. (2006), "A Practical and Complete Approach to Predicate Refinement"

[BL13]: Beyer et al. (2013), "Explicit-State Software Model Checking Based on CEGAR and Interpolation"

As listed in the Figure 1.1, many state-of-the-art tools apply CEGAR in their analysis. Other notable applications include the predicate-based model checker BLAST [Bey+05; Hen+03], which implements a lazy predicate abstraction (cf. Section 7.1.2) on control flow automaton of a program; SATABS that extends the previous approach with a SAT solver for bit-precise predicate abstraction [Cla+05]; and combination with other techniques like explicit-value analysis in CPAchecker [BF18; BL13].

[Bey+05]: Beyer et al. (2005), "Checking memory safety with Blast"
[Hen+03]: Henzinger et al. (2003), "Software Verification with BLAST"
[Cla+05]: Clarke et al. (2005), "SATABS: SAT-based predicate abstraction for ANSI-C"
[BF18]: Beyer et al. (2018), "In-Place vs. Copy-on-Write CEGAR Refinement for Block Summarization with Caching"
[BL13]: Beyer et al. (2013), "Explicit-State Software Model Checking Based on CEGAR and Interpolation"

A unified approach to the above-mentioned techniques was presented in the recent work of Beyer et al. [BGS18] as a combination of model checking and dataflow analysis. The unified framework enables an easy combination of abstract domains, no matter whether they were invented for dataflow analysis or for model checking. Alternatively, we can perform the refinement of abstraction on-the-fly [BHT08].

[BGS18]: Beyer et al. (2018), "Combining Model Checking and Data-Flow Analysis"
[BHT08]: Beyer et al. (2008), "Program analysis with dynamic precision adjustment"

7.1.2 Lazy abstraction

A bottleneck in CEGAR loop is the repeated analysis of the program. One technique that aims to mitigate this problem is lazy abstraction, which involves refining a single abstract model on-demand, allowing different parts of the model to exhibit varying levels of abstraction [Hen+02]. This is achieved by maintaining explicit control-flow graph information, which describes how the program locations are traversed, and symbolic dataflow information, which describes what holds at a program location [Alb+12b]. Lazy abstraction can be improved through interpolation [McM06] (cf. Section 7.1.3). In lazy interpolant-based model checking, the refinement is guided by refuting program paths instead of predicate refinement. This avoids the high cost of postcondition evaluation in the predicate approach.

[Hen+02]: Henzinger et al. (2002), "Lazy Abstraction"

[Alb+12b]: Alberti et al. (2012), "Lazy abstraction with interpolants for arrays"
[McM06]: McMillan (2006), "Lazy Abstraction with Interpolants"

Lazy abstraction is particularly useful in shape analysis, where it allows for local refinement of shapes only when necessary [BHT06; Hen+03], as well as in array abstractions [Alb+12b; Alb+12c].

[BHT06]: Beyer et al. (2006), "Lazy Shape Analysis"
[Hen+03]: Henzinger et al. (2003), "Software Verification with BLAST"
[Alb+12b]: Alberti et al. (2012), "Lazy abstraction with interpolants for arrays"
[Alb+12c]: Alberti et al. (2012), "SAFARI: SMT-Based Abstraction for Arrays with Interpolants"

7.1.3 Interpolation

Another technique commonly used to improve abstraction-based techniques is interpolation. The fundamental idea of interpolation is to take both feasible and infeasible abstract paths, where the latter subsumes the former, and refine the feasible path to generate more concrete but still feasible paths that are subsumed by the infeasible path [McM03]. Interpolation is often employed as a refinement procedure for spurious counterexamples within the CEGAR loop. It has a wide range of use cases, such as generating relevant predicates for predicate abstraction [Cim+16;

[McM03]: McMillan (2003), "Interpolation and SAT-based model checking"

Hen+04], range predicates in array abstraction [JM07], and as a refinement technique for symbolic [Ibi16] and concolic execution [JMN13].

Interpolation has also been applied in non-symbolic abstractions. For instance, in the tool DAGGER [Gul+08], interpolation is used to improve the precision of widening of octagon and polyhedra domains in order to avoid false alarms. Similarly, in VINTA [AGC12], interpolation is used to refine the interval domain.

7.1.4 Domain Refinement

There are several techniques that can be employed to address the imprecision in abstract domains without the need for repeated analysis. One approach is to enhance the capabilities of the domains in use through the use of functor domains.

In abstraction, one major source of imprecision is the approximation induced by abstract unions, which are used to merge multiple paths of the program. However, not all abstract domains are closed under the union operation, which can result in a loss of precision. The *disjunctive completion* method is a common solution to this problem [FR99]. This method introduces to the abstraction all the concrete disjunctions of elements from the domain that were initially missing, resulting in the most abstract domain that is exact on disjunctions of abstract properties in the original domain. Similarly, *complementation* is another technique used to supplement missing complements to the abstraction [Cor+95].

On the other hand, *fixpoint completion* is a program-dependent refinement technique that improves the abstract domain by introducing all the properties that would make the fixpoint iteration imprecise [GQ01; GRS00].

Relational abstract domains, however, often suffer from scalability issues. One general technique to ensure linear cost of abstract domains is to split variables into independent smaller sets, known as *variable packing* technique [Min06c]. Variable packing can be further refined by maintaining relations between sets of variables [Bou12].

State partitioning techniques, based on decision-tree data structures such as binary decision diagrams, work similarly to variable packing. They split program states into smaller parts and enable relational reasoning about value properties only within those parts [MJ81]. State partitioning is commonly adopted in state-of-the-art tools [Ber+10; Ber+15]. Some abstract domains also incorporate decision trees to handle disjunctions in programs or prove conditional termination [CCM10; UM14].

In the pursuit of increased precision, some techniques deviate from the lattice-based approach and develop weaker formalisms for abstract domains with the quasi-join operation [Gan+13].

Thus far, we have discussed abstract domains with varying degrees of expressiveness, cost, and precision. However, in many cases, it is desirable to commence with the most straightforward domains and dynamically refine them when encountering spurious counterexamples. One approach to refinement is based on the observation that abstract domains themselves form a lattice, as depicted in Figure 7.2. Within this lattice, two

[Cim+16]: Cimatti et al. (2016), "Infinite-state invariant checking with IC3 and predicate abstraction"
[Hen+04]: Henzinger et al. (2004), "Abstractions from proofs"
[JM07]: Jhala et al. (2007), "Array Abstractions from Proofs"
[Ibi16]: Ibing (2016), "Dynamic symbolic execution with interpolation based path merging"
[JMN13]: Jaffar et al. (2013), "Boosting concolic testing via interpolation."
[Gul+08]: Gulavani et al. (2008), "Automatically Refining Abstract Interpretations"
[AGC12]: Albarghouthi et al. (2012), "Craig interpretation"
[FR99]: Filé et al. (1999), "The powerset operator on abstract interpretations"
[Cor+95]: Cortesi et al. (1995), "Complementation in abstract interpretation"
[GQ01]: Giacobazzi et al. (2001), "Incompleteness, Counterexamples, and Refinements in Abstract Model-Checking"
[GRS00]: Giacobazzi et al. (2000), "Making Abstract Interpretations Complete"
[Min06c]: Miné (2006), "The octagon abstract domain"
[Bou12]: Bouaziz (2012), "TreeKs: A Functor to Make Numerical Abstract Domains Scalable"
[MJ81]: Muchnick et al. (1981), *Program Flow Analysis: Theory and Application*
[Ber+10]: Bertrane et al. (2010), "Static Analysis and Verification of Aerospace Software by Abstract Interpretation"
[Ber+15]: Bertrane et al. (2015), "Static Analysis and Verification of Aerospace Software by Abstract Interpretation"
[CCM10]: Cousot et al. (2010), "A Scalable Segmented Decision Tree Abstract Domain"
[UM14]: Urban et al. (2014), "A Decision Tree Abstract Domain for Proving Conditional Termination"
[Gan+13]: Gange et al. (2013), "Abstract Interpretation over Non-lattice Abstract Domains"

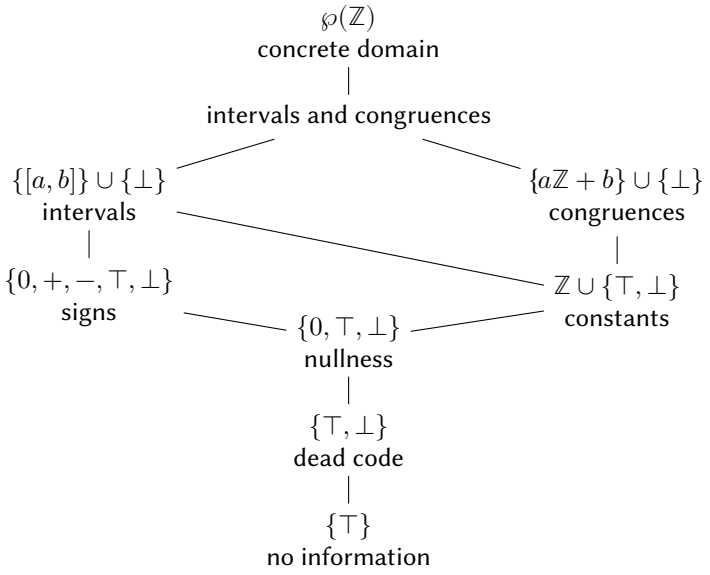


Figure 7.2: Lattice of non-relational abstract domains.

domains can be refined through a meet operation, also referred to as reduced product construction, as expounded upon in the works of Toubhans et al. [TCR13] and Cousot [CCM11]. A more comprehensive examination of large-scale reduced-products can be found in the study conducted by Cousot [Cou+07].

[TCR13]: Toubhans et al. (2013), "Reduced Product Combination of Abstract Domains for Shapes"

[CCM11]: Cousot et al. (2011), "The Reduced Product of Abstract Domains and the Combination of Decision Procedures"

[Cou+07]: Cousot et al. (2007), "Combination of Abstractions in the ASTRÉE Static Analyzer"

7.2 Metadomain

Program semantics can be thought of as a collection of concrete domains, each corresponding to a specific data type and its associated operations in the programming language. However, careful consideration must be given to interactions between different concrete domains to ensure proper abstraction. For example, boolean and scalar domains may interact in relational operations, or array dimensions may be specified using scalars from a specific concrete domain.

Furthermore, it is often desirable to support the use of multiple abstract domains simultaneously in a program. While the concrete domain is always available, verifiers may employ various abstract domains to reason about program inputs from the environment. The challenge lies in integrating these diverse domains, which may belong to different categories, and resolving any interactions that may arise during program verification. As mentioned earlier, abstract operations expect all operands to belong to the same domain.

Dealing with these interactions independently for each domain is impractical and undesirable. Instead, it is desirable to implement domains as a homomorphism from the algebra of the concrete domain to the algebra of the abstract domain, with minimal effort to handle interdomain interactions.

For instance, boolean interaction can be addressed by fixing its domain to a tristate domain, and all relational interactions must go through this domain. On the other hand, interactions between other domains are more complex. To address this, we create a metadomain that is

responsible for interactions between abstract domains, and we implement the entire abstraction as a homomorphism from the concrete domain to the metadomain.

The metadomain follows a similar approach as the lattice of domains. Whenever two domains interact, we find the least upper bound domain in the lattice, where a conversion between the respective domains is defined, and perform the operation in that domain. The order of the lattice is determined by the existence of a conversion between the domains, denoted as domain $\mathcal{A}_1 \preceq \mathcal{A}_2$ if conversion $\chi_{\mathcal{A}_1 \to \mathcal{A}_2}$ is feasible. It is necessary for the lattice to be complete, ensuring that there is always a domain in which the operation can be performed.

Definition 7.2.1 *We define a **conversion relation** R_χ as follows: domains $\mathcal{A}_1$ and $\mathcal{A}_2$ are in R_χ if there exists a conversion $\chi_{\mathcal{A}_1 \to \mathcal{A}_2}$.*

*We say that conversion is **feasible** $\chi_{\mathcal{A}_1 \to \mathcal{A}_2}$ if $\mathcal{A}_1$ is in relation with $\mathcal{A}_2$ in the transitive closure of R_χ.*

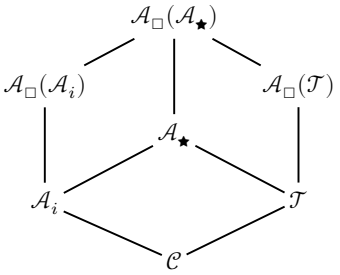


Figure 7.3: Example of metadomain lattice consisting of the unit domain $\mathcal{A}_\bigstar$, term domain $\mathcal{T}$, smashed array domains with parametric content domains $\mathcal{A}_\square$, interval domain $\mathcal{A}_i$ and concrete domain $\mathcal{C}$.

This approach reduces the number of needed conversions, as in many cases, we do not want to define conversions between all possible domains, despite the fact that their interactions might occur in the program.

Example 7.2.1 Consider the domain lattice shown in Figure 7.3, which determines the domain in which the following operations should be performed:

- Addition of $[0, 1]$ and 4 results in the conversion of concrete 4 into the interval domain.
- Addition of term ★ and 4 results in the conversion of 4 through intermediate domain (either interval or term domain).
- Addition of term offset to smashed arrays of interval produces a smashed array of ★ values.

Note that this hierarchy implies that aggregate pointers must be able to be constructed from abstract scalar values. This can often be achieved by using the abstract scalar as an offset part and setting the identifier part of the pointer to $\bot$ or 0.

We propose that an additional linear order be established for domains to ensure the selection of a unique conversion path in cases where multiple paths may be available. However, further investigation is required to determine the feasibility of this solution and its potential implications. Alternatively, one can explore the possibility of designing abstractions that avoid such interactions altogether. In fact, in most of our applications, such cases do not arise.

Interesting interactions are those between scalars and aggregates, as well as pointers. Recall admissible pointer statements (cf. Definition 3.2.1), where only the addition of pointers and scalars is allowed, implementing pointer arithmetic and resulting in an abstract pointer. For this to be valid, we require that all scalar domains in the metadomain are ordered below aggregate and pointer domains in the lattice. This is because if a

pointer domain $\mathcal{A}_{\mathsf{p}}$ was ordered below a scalar domain $\mathcal{A}_{\mathsf{s}}$, i.e., $\mathcal{A}_{\mathsf{p}} \preceq \mathcal{A}_{\mathsf{s}}$, the pointer arithmetic would result in a value $\mathcal{A}_{\mathsf{s}}$ which does not define memory manipulating operations and can not be used as a pointer value later in the program. It is important to note that some domains may cover multiple categories. For example, $\mathcal{A}_{\bigstar}$ may define both scalar and aggregate operations.

The compilation-based abstraction, presented in this thesis, leverages the concept of representing common analysis techniques as domains. This holds true for domain interactions as well. We can view the lattice of domains $\mathcal{L}$ as a functor domain that is parametrized by its constituent domains. Consequently, computations are carried out exclusively within the lattice domain and the concrete domain $\mathcal{C} \times \mathcal{L}$. When encountering mixed computations involving both concrete and abstract values, we employ a "lifting" mechanism to map concrete values to specific lattice domain elements. This enables us to lift scalars, aggregates, and pointers to different domains based on their origin. However, it is worth noting that the concrete domain can be treated as the bottom of the lattice, as all other domains can be constructed from concrete values using their abstraction function α. Though, it is necessary to differentiate between categories of concrete values in the lattice, as they may need to be lifted to different abstract domains.

Each metadomain needs to specify what is an "entry" domain into which we abstract ambiguous values. We define entry domains for each value category, which effectively dictates in which domain to perform abstract `amb` operation and allocation for abstract aggregates. Furthermore, since the concrete domain is not kept in the lattice of the metadomain, we define the lowest domain for each value category, into which conversion from concrete values is performed. This corresponds to the definition of abstraction operator α within the metadomain.

Nevertheless, there are instances wherein it is advantageous to establish entry domains that are tailored to a specific abstraction. For instance, defining specialized entry domains for abstract strings or loop indices can yield more precise abstraction of specific scalars and aggregates. These entries can be inferred from value types, or alternatively, static analysis can be utilized to annotate values, such as loop indices, that may not be readily apparent from the value construction.

Formally, a metadomain $\mathcal{L}$ is an n-ary functor domain that takes a set of domains $\mathcal{D}_{\mathcal{L}} = \mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_n$ and a conversion relation R_{χ} as input and constructs a new domain that enables computation in the union of these domains. The metadomain defines a partial order $\preceq$ on the constituent domains, given by their conversion relation R_{χ}. Additionally, the metadomain defines a function ι that maps program values to a specific constituent domain $\mathcal{D} \in \mathcal{D}_{\mathcal{L}}$.

Definition 7.2.2 *Let $\mathcal{L}$ be a metadomain with a join operator $\sqcup$ on its domain lattice. For set of values $\{v_1, v_2, \dots, v_n\}$ we say that their **superdomain**, denoted as ς, in the metadomain $\mathcal{L}$ is:*

$$\varsigma(\{v_1, v_2, \dots, v_n\}) \triangleq \bigsqcup_{i=1}^{n} \iota(v_i)$$

Definition 7.2.3 *A metadomain also defines requisite components to be a value domain:*

- *a set of computer-representable abstract values* $\mathcal{L} = \bigcup_{i=1}^{n} \mathcal{D}_i$,
- *an effective partial order of values* $a \sqsubseteq b$ *is computed in their superdomain* $\mathcal{S} = \varsigma(\{a, b\})$*:*

$$\chi_{\iota(a)\to\mathcal{S}}(a) \sqsubseteq_{\mathcal{S}} \chi_{\iota(b)\to\mathcal{S}}(b) \implies a \sqsubseteq_{\mathcal{L}} b$$

- *a monotonic concretization function* $\gamma_{\mathcal{L}}(v) \triangleq \gamma_{\iota(v)}(v)$,
- *a set of monotonic abstraction functions*

$$\alpha_{\mathcal{L}_{\mathsf{s}}} : \mathcal{C}_{\mathsf{s}} \to \mathcal{D}_{\mathsf{s}}$$
$$\alpha_{\mathcal{L}_{\mathsf{p}}} : \mathcal{C}_{\mathsf{p}} \to \mathcal{D}_{\mathsf{p}}$$

  where $\mathcal{D}_{\mathsf{s}}, \mathcal{D}_{\mathsf{p}} \in \mathcal{D}_{\mathcal{L}}$ *are abstraction domains for scalars, pointers respectively,*

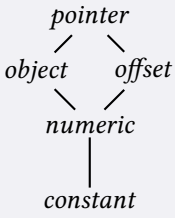
- *a smallest* $\bot_{\mathcal{L}} \in \mathcal{L}$ *and a largest element* $\top_{\mathcal{L}} \in \mathcal{L}$,
- *a sound and effective abstraction of program operations performed in superdomain of their operands,*

Note that one needs to be careful with the design of metadomain conversions. When interacting between scalar and memory domains, it is possible that an aggregate may not be capable of storing a given abstract value. For example, consider a store operation of an interval value to a smashed array of terms (cf. Figure 7.3). Ideally, we do not want to convert the interval to a smashed array domain but rather to its content domain (term). To handle aggregates with scalar content domains, we let aggregates define their content domain and lift stored values to the superdomain of the content domain and the domain of the value to be stored. However, this may result in a superdomain that is not representable in the content of the aggregate, and we may need to convert the entire abstract aggregate, which requires the presence of this domain in the lattice. This is also the case in Figure 7.3, where the superdomain of a content term domain and to be stored interval value is a unit domain. As a result, we would need to convert the smashed array to an alternative with unit content.

Another issue is that in a general setting, we need to be able to store aggregates of aggregates and their mixed variants of domains. In this thesis, we mainly employed metadomains to resolve scalar interactions, but as one can see, a metadomain for all domain categories still raises many questions and needs to be fully formalized. This opens up interesting avenues for potential research in the future.

Example 7.2.2 Recall pointer arithmetic abstraction from the previous chapter. We consider programs with three domains: the concrete domain $\mathcal{C}$, a numeric domain $\widehat{\mathsf{D}}$ and the proposed $\widehat{\mathsf{Pa}}(\widehat{\mathsf{D}})$ domain (cf. the metadomain lattice on the left). In the simplest case, $\widehat{\mathsf{Pa}}$ is instantiated with the same domain $\widehat{\mathsf{D}}$ that is normally used to represent scalars

in the program. In this case, lifting a value x from $\widehat{\mathsf{D}}$ into $\widehat{\mathsf{Pa}}(\widehat{\mathsf{D}})$ is simple and does not cause any loss of precision: the abstract object identifier is simply taken to be the value x, while the concrete part of the pointer is assigned the value $\bot_{\mathcal{C}}$: we do not know what concrete memory location to assign to an arbitrary value. The invalid concrete part also prevents the program from accessing memory using such a pointer (yielding an error instead).



In general, the $\widehat{\mathsf{Pa}}$ domain is complementary to domains that abstract pointer offsets – aggregate domains (e.g. those which allow symbolic indices). The product domain constructed from $\widehat{\mathsf{Pa}}$ and a domain abstracting pointer offsets would allow performing complete abstraction of memory – a possible hierarchy is depicted on the right.

While all unambiguous allocations are kept in the concrete domain $\mathcal{C}$, each ambiguous allocation can be tracked at a different level of granularity, using a suitable instance of $\widehat{\mathsf{Pa}}$.

For instance, pointers that only occur in equality comparisons can be abstracted with a weaker instantiation, e.g., $\widehat{\mathsf{Pa}}(EQ)$, where EQ is an *equality domain* that only keeps track of equivalence relations between values and hence leverages simpler SMT theory of equalities [KS16].

In contrast, pointers that occur in arithmetic operations need more expressive domain, e.g., the $\widehat{\mathsf{Pa}}$ domain[2] that utilizes the theory of bit-vectors. These instantiations then form a chain in the hierarchy, from the weakest to the most expressive, in a manner similar to theory refinement [Hyv+17]. Whenever different pointer representations meet in the computation, we can lift the weaker domain to the stronger one.

[KS16]: Kroening et al. (2016), "Equality Logic and Uninterpreted Functions"

2: In this case, values from $\widehat{\mathsf{Pa}}(EQ)$ and values from $\widehat{\mathsf{Pa}}$ use a different theory in their respective SMT queries.

[Hyv+17]: Hyvarinen et al. (2017), "Theory Refinement for Program Verification"

One of the key benefits of the metadomain approach is its ability to enable domain composability. The elementary metadomain consists of a single abstract domain, allowing for easy extension by adding new domains by providing conversions, which define their placement in the domain lattice, and potentially new abstraction entry domains.

We can extend existing refinement techniques to incorporate metadomains. For instance, backward constraint propagation can be applied without adjustments, as we treat backward operations in the same manner as forward operations, with the only difference being the potential need to lift domains of values during backward refinement. Another potential application of metadomain analysis is in counterexample-guided abstraction, where entry domains can be refined during the refinement loop to achieve more precise abstraction. However, further investigation is needed to fully explore the potential of these applications.

Our implementation of abstraction is entirely based on metadomain analysis. Nevertheless, we acknowledge that there are limitations and loose ends that require formalization. For example, the interaction between incompatible aggregate domains and the problem of multiple possible conversion paths remain open problems for this technique.

Summary

In this chapter, we presented general refinement techniques, with a focus on interdomain refinement. We defined common counterexample-guided abstraction refinement and its enhancements using interpolation and lazy abstraction. However, our main objective was to introduce an analysis approach that spans multiple domains, including scalars, aggregates, and pointers. To achieve this, we proposed a metadomain abstraction that implements program semantics of the union of multiple abstract domains. The primary advantage of this approach is that it provides a uniform and extensible mechanism for domain interaction, eliminating the need to consider mixed operations in other domains. The core idea is to maintain a lattice of domains that defines how to convert between domains. This technique allows us to view abstraction as a single homomorphism from the concrete domain to a single metadomain, simplifying the overall abstraction design and enabling integration with other single-domain techniques.

8 Abstract Analysis

With the abstraction framework we developed in the preceding chapters, we can now explore how it can be integrated into common program analyses. This chapter discusses the various design options that exist for compilation-based abstraction and compares them to more conventional methods. We also redefine common techniques like symbolic execution and explicit state model checking to take advantage of compilation-based abstraction in the latter part of the chapter.

The content of this chapter is based on the following publication:

- Henrich Lauko. *Abstractions via Program Transformations*. Brno, 2020. (PhD Thesis Proposal) [Lau20]
- Henrich Lauko, Petr Ročkai, and Jiří Barnat. "Symbolic Computation via Program Transformation." In: *Theoretical Aspects of Computing – ICTAC 2018. 15th International Colloquium, Stellenbosch, South Africa, October 16-19, 2018, Proceedings*. Ed. by Bernd Fischer and Tarmo Uustalu. Cham: Springer, 2018. ISBN: 978-3-030-02507-6 [LRB18]
- Henrich Lauko, Petr Ročkai, Vladimír Štill, and Jiří Barnat. "DIVINE – Model Checker for C++." In: *Automatic Software Verification*. Ed. by Dirk Beyer. (forthcoming) [Lau+ng]

This chapter will consider various aspects of program analysis design and their effects on the resulting analysis. For instance, one important aspect to consider is the level of precision. It ranges from imprecise dataflow analysis to highly precise but computationally expensive explicit state model checking. To improve precision even further, we can choose from various approaches to abstraction, such as eager and lazy abstraction. In the case of dynamic analysis, we can choose different abstraction levels, whether we refine the abstraction at each location, per path, or incrementally refine the state representation instead of restarting the analysis.

Another aspect to consider is the size of program primitives that are abstracted at once. At the minimum level of abstraction are individual values and statements, but it is also possible to perform abstraction of branch-free, loop-free blocks, or even entire functions [BKW10]. It is important to note that all of these choices are interconnected, as the choice of abstraction can impact the efficiency of other aspects of the analysis, such as refinement and, subsequently, the exploration of the program state space.

[BKW10]: Beyer et al. (2010), "Predicate Abstraction with Adjustable-Block Encoding"

We can also categorize different approaches based on the intended semantics of the analysis, which is what the analysis aims to achieve. In program verification, we aim to exhaustively cover the program's state space, which we refer to as program-proving tools. Such tools typically overapproximate the program's state space. As a result, if the abstraction is error-free, we can infer that the original program is also error-free. On the other hand, we have underapproximative tools, such as bug hunters, fuzzers, or test generators. Some of these tools provide guarantees for

specific program properties, such as the absence of a bug up to a certain depth in bounded model checking. Whereas other heuristic-based approaches are more challenging to reason about.

In recent years, the program analysis community has been shifting toward designing modular approaches that others can build upon. Using the same backend, representation, or even analysis algorithms makes it easier to compare research findings, such as the efficiency of different program domains, summarization techniques, or refinement approaches. This allows for more effective comparative research.

A new research question arises in this context: How can we integrate existing verification systems in order to benefit from their respective strengths maximally? To enable cooperation between verification tools, we need standardized interfaces that allow passing artifacts with valuable information from one tool to another. Besides the programs and their specifications, such verification artifacts include transformed or reduced programs, error paths, invariants, witnesses, and partial verification results in general [Bey22a; BP22; BW20].

For instance, multiple production-quality SMT solvers are readily available and even provide a standard interface [BFT16]. While a certain degree of integration is required to achieve optimal performance, solvers have attained nearly commodity status.

On the other hand, program analysis tools are falling behind, but a few exceptions exist. The symbolic execution engine KLEE [CDE08] for LLVM bitcode [LA04] is used as a backend by other analysis tools [Cha+21; CKC12]. Undoubtedly, the fact that it is based on the (ubiquitous) LLVM intermediate language has helped it foster wider adoption. Similarly, other tools create families of tools with shared backends. For instance, CPAchecker [BK11] was designed with modularity in mind to allow configuration of an abstract domain in use, summarization techniques, and memory representation, and also permits the change of the entire model checking algorithm. This led to multiple tools based on the framework: CPA-BAM [And+17], CPAlien [MV14], CPALockator [AMK21], and CPARec [Che+15]. Likewise, we can mention tools like CBMC [CKL04] or UAutomizer [HHP13] that also led to the development of kindred tools.

In abstract interpretation, the trend is to create a library toolkit for domain creation [Gur+15; JM09; SPV15], but using the domains in between tools is relatively rare since the design usually restricts the domain to a particular representation or algorithm used in the tool.

Since symbolic verification is tightly coupled with SMT solving, it is desirable to make the representation in use easy to translate and moreover easy to reason about. Therefore, in software model checking, numerous tools build upon constrained horn clauses representation CHC that extends standard SMT representation [BMR12]. In fact this technique corresponds to property-directed reachability [HB12]. Representation of a verification problem in standard CHC allows to utilize traditional verification algorithms, like bounded model checking [Bie+03], k-induction [SSS00], or interpolation & lazy abstraction [McM06], on a simple representation, and share it across the tools [Gur+15; KGC14].

As one can observe, a common language for program representation is at the core of modular techniques. A good choice of program repre-

[Bey22a]: Beyer (2022), "Cooperative verification: Towards reliable safety-critical systems (invited talk)"
[BP22]: Beyer et al. (2022), "Software Model Checking: 20 Years and Beyond"
[BW20]: Beyer et al. (2020), "Verification artifacts in cooperative verification: Survey and unifying component framework"
[BFT16]: Barrett et al. (2016), *The Satisfiability Modulo Theories Library (SMT-LIB)*
[CDE08]: Cadar et al. (2008), "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs"
[BK11]: Beyer et al. (2011), "CPAchecker: A Tool for Configurable Software Verification"
[And+17]: Andrianov et al. (2017), "CPA-BAM-BnB: Block-abstraction memoization and region-based memory models for predicate abstractions"
[MV14]: Müller et al. (2014), "CPAlien: Shape Analyzer for CPAChecker"
[AMK21]: Andrianov et al. (2021), "CPALockator: Thread-Modular Analysis with Projections"
[Che+15]: Chen et al. (2015), "CPArec: Verifying Recursive Programs via Source-to-Source Program Transformation"
[Gur+15]: Gurfinkel et al. (2015), "The SeaHorn Verification Framework"
[JM09]: Jeannet et al. (2009), "Apron: A Library of Numerical Abstract Domains for Static Analysis"
[SPV15]: Singh et al. (2015), "Making numerical program analysis fast"
[BMR12]: Bjørner et al. (2012), "Program Verification as Satisfiability Modulo Theories."
[HB12]: Hoder et al. (2012), "Generalized property directed reachability"
[Bie+03]: Biere et al. (2003), "Bounded model checking."
[SSS00]: Sheeran et al. (2000), "Checking safety properties using induction and a SAT-solver"
[McM06]: McMillan (2006), "Lazy Abstraction with Interpolants"
[Gur+15]: Gurfinkel et al. (2015), "The SeaHorn Verification Framework"
[KGC14]: Komuravelli et al. (2014), "SMT-Based Model Checking for Recursive Programs"

sentation and framework extension points are probable reasons for the success of tool families like KLEE [CDE08], CPAchecker [BK11], UAutomizer [HHP13], or CBMC [CKL04]. Their representation is easy to interpret but expressive enough to capture interesting program behaviors. Using a common language allows combining static and dynamic analysis, share analysis results, and provide easy-to-learn semantics if it is a well-known representation.

Program Representation

Regarding program representation used for analysis, we recognize four main approaches:

1. **IR-based analysis:** In this approach, we transform the target program into an intermediate representation (IR), providing a high-level view of the program's behavior. Abstraction is performed on the IR level, where it is also interpreted. This approach is simpler to implement because IRs are typically easier to work with than machine code, but it is slower as the IR needs to be interpreted rather than executed directly. Due to simplicity, many state-of-the-art tools use this approach.
2. **IR-less (native) analysis:** This approach involves executing the unmodified machine code of the target program and instrumenting it at runtime to perform abstract (symbolic) execution. This approach is more efficient because it avoids the overhead of translating the program to an IR and interpreting it. However, it requires using a framework like Intel Pin to control the execution of the target program at runtime.
3. **IR-to-native (mixed) analysis:** This approach considers IR as the best representation for abstraction, but after the static abstraction, it compiles the abstracted program to a native binary and performs IR-less analysis. It combines the advantages of both IR-based and IR-less analysis, as it can provide the benefits of a simpler abstraction on the IR level and the efficiency of native execution.
4. **Source-level analysis:** In this approach, the program is analyzed in its original source code form without any intermediate representation or compilation. While this approach can offer the most precise understanding of the program's intended semantics, it is usually analyzed only statically due to its complexity, making it difficult to be interpreted accurately.

Given our objective to perform abstraction on the LLVM IR level, we will focus on the first and third approaches that operate on an intermediate level or translate the intermediate representation to native representation.

8.1 Program Abstraction

Abstraction in program verification is indispensable for the successful verification of complex systems. It simplifies the reasoning about interactions with the program environment and computation with nondeterministic data in general. In order to simplify abstract computation even further, many techniques employ further refinements to abstraction, e.g., counterexample-guided abstraction refinement [Cla+00] or lazy abstraction techniques [Hen+02]. Most of these techniques, however, are tightly embedded in the core of verification tools. Even though the abstraction techniques share a common foundation, tools usually reimplement the standard analyses and abstraction domains. Moreover, the integration of abstraction into a robust verification engine might introduce undesirable complexity and can be a cause of faulty analysis.

In this thesis, we study the design of self-contained abstraction techniques independent of an execution engine. This work aims to provide an abstraction engine and program runtime that can be used with LLVM IR or binary-based tools, regardless of their program abstraction capabilities.

The main concept behind this approach is to perform IR-to-native or IR-to-IR abstraction and then use respective tools to interpret or execute the abstracted program. The goal is to represent the abstraction in semantics that the analysis tool can already understand – concrete semantics. That is to describe abstract semantics in terms of concrete computation. This approach can generally be viewed as re-casting common techniques as an abstract domain, which we will describe in concrete semantics. As a matter of fact, we have seen examples of this approach in previous chapters where symbolic execution was implemented as a term domain or abstraction refinement as a backward propagation domain. By abstracting the program itself, we can enable the interpreter or native execution to prioritize the efficiency of the concrete semantics without the need to incorporate complex abstract analysis techniques.

[Hsi+97]: Hsieh et al. (1997), "Compilers for improved Java performance"
[Wie83]: Wiedmann (1983), "A Performance Comparison between an APL Interpreter and Compiler"

A similar shift of responsibilities is a well-known technique in computer science [Hsi+97; Wie83]. For example, it can be observed in compiled and interpreted languages that move various tasks between compilation time and execution time. This balances the tool's flexibility, dynamic extensibility, and performance.

We will now examine how much it is possible to shift abstraction from the interpreter to the compilation (static) part of the program analysis. The goal is to make the analyses more performant without losing the flexibility of interpretation-based analysis. Naturally, there exist many possible boundaries between abstraction and the execution engine — each defined by how much of the abstract interpretation is performed by the program or the executor (cf. Figure 8.1).

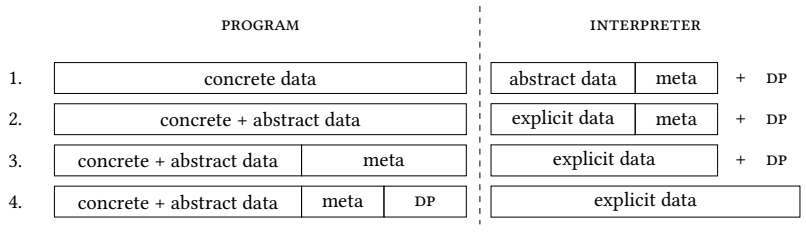


Figure 8.1: Possible division of abstraction responsibilities between program and interpreter.

1. The first boundary is a common approach to abstract interpretation where the interpreter takes care of the entire abstraction (maintaining abstract data, additional metadata, and optionally using decision procedure DP).

2. In the second approach, the program maintains abstract data. However, the interpreter still keeps metadata that keeps track for each value, whether it is abstract or concrete. This information is used when the interpreter needs to decide how to interpret program memory, e.g., whether to perform an explicit or an abstract comparison of states.

3. The third approach involves transferring the responsibility for metadata to the program itself, which can keep the metadata in the shadow memory. This method is similar to that used by program sanitizers [SS15]. Using this method, the interpreter must retrieve the metadata from the program.

4. The last approach gives full responsibility for abstraction to the program, and the interpreter executes the program explicitly. The decision procedure needs to take care of nondeterministic execution and determine what path to take when a nondeterministic branch occurs, or the responsibility of the nondeterminism can be left to the interpreter.

A common approach used by abstract interpreters is to be responsible for both the maintanence of abstract data and the execution of operations (abstract semantics). As an alternative, we suggest delegating the responsibility for abstract data to the program and incorporating abstract semantics directly into the program representation (realize abstract operations in a language the interpreter understands — as concrete code).

Irrespective of the type of abstraction used, it is desirable to minimize its effect on program execution and reduce the imprecision of the analysis. To achieve this, only ambiguous program data and corresponding operations need to be abstracted. Hence, the analysis must identify the operations to be abstracted. Another approach could be to determine whether to perform an abstract or concrete operation dynamically, and this decision can be made either by the program under analysis or the execution engine. In general, we will maintain metadata associated with the values to determine whether they are concrete or abstract, and this metadata will assist in making the decision.

Another aspect of abstract execution is the ability to decide the feasibility of a program path. While it may be possible to refactor this functionality out of the execution engine, it is impractical to compile entire SMT solver into the program and reinterpret it by further analyses.

Overall, we will recognize two main approaches based on the component responsible for program abstraction:

- An *interpretation-based approach* in which the interpreter is entirely responsible for abstracted data and execution of abstract semantics.

▶ A *compilation-based approach* in which the abstract semantics is "compiled" into a program under analysis, which then concretely executes the abstraction.

8.1.1 Interpretation-based Abstraction

The most broadly adopted interpretation-based approach is unsurprisingly abstract interpretation (cf. Section 3.6.1) that finds utilization in static analysis or compilation process. It falls into the first category described in Figure 8.1, i.e., the interpreter is responsible for everything abstraction-related. In general an abstract interpreter computes an inductive invariant — assigns invariants to program locations, and performs assertion checks. Interpretation-based tools often follow a similar architecture, which allows for configuring the interpretation algorithms and the domain choice. The interpretation strategy determines in what order program statements are evaluated, while the domain provides the value representation and semantics to the operations (transfer functions). One example of an interpretation-based static analysis tool is Clam [GN21] described in Figure 8.2. It operates on Crab IR translated from LLVM IR.

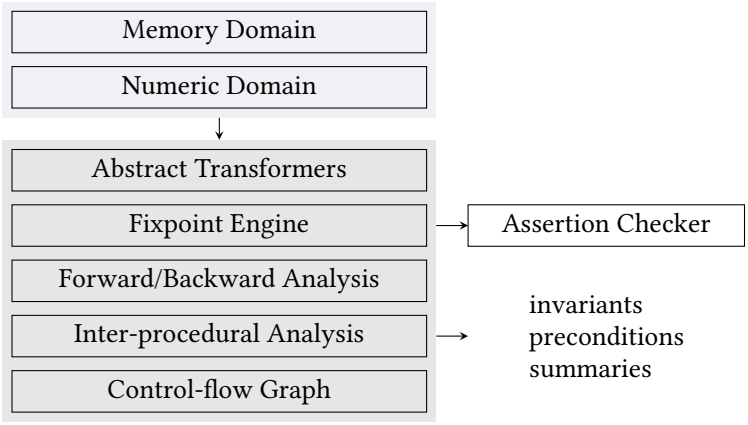


Figure 8.2: Clam has been designed in a modular fashion so that new components (abstract domains, fixpoint algorithms, and inter-procedural or backward analyses) can be easily plugged in. The only fixed component is its intermediate representation, called Crab IR [GN21].

The advantage of this approach is that we do not have to execute the program. Instead, we can use a fixpoint iteration algorithm to infer program invariants. Notably, abstract interpreters analyze the control flow graph and preserve the entire abstraction throughout the process internally. The interpreter also holds definitions of abstract transformers and applies them to internal program abstraction. We do not necessarily follow the order of execution but only update the values at locations that have changed. There are many types of iteration strategies with different accelerations and convergence of fixpoint iteration. For instance, it is common to iterate in the weak topological order of program control-flow graph to deal with loops more efficiently [Bra+14].

[Bra+14]: Brat et al. (2014), "IKOS: A Framework for Static Analysis Based on Abstract Interpretation"

In practice, static analysis based on abstract interpretation is the most scalable approach to program analysis. However, we pay for scalability by imprecision. Moreover, in comparison to model checking, abstract interpretation, in general, does not provide a counterexample and refinement possibilities since there is no path to be examined.

In contrast, abstract model checking (discussed earlier in Section 3.6.3) focuses on analyzing the dynamic behavior of a program and properties

of its runs. Effectively, it generates the transition relation from the initial state and merges equivalent states when reached. This still falls into the category of interpretation-based approaches since the state space generation interprets the effect of transition action on a particular state similarly to the transfer function application in the abstract interpretation. Moreover, the value abstraction in the model checking is analogous to the one used in the abstract interpretation.

Finally, we recognize abstract (symbolic) executors (cf. Section 3.6.2) that also employ an interpretation-based strategy since they usually operate on an intermediate representation and keep the symbolic state solely in the interpreter. They are comparable to concrete execution, with the distinction that they compute with a symbolic representation of data and execute abstract semantics that lead to branching executions. For instance, KLEE [CDE08] symbolically interprets LLVM IR.

Symbolic executors, in contrast to model checking, only solve the reachability problem. They generally do not take any specification, just the input program, and they iteratively explore all possible execution paths.

Naturally, the categorization is not strict, and various tools utilize techniques from multiple categories. For instance, UFO [Alb+12a] is a tool that improves the accuracy of abstract interpretation by incorporating abstraction refinement techniques from model checking. It iteratively employs Craig interpolants to refine the abstract interpretation and prevent it from producing false alarms. The algorithm continues its refinement process until it finds a safe inductive invariant, a counterexample, or runs out of resources. This allows UFO to regain precision often lost in multiple steps of abstract interpretation.

[Alb+12a]: Albarghouthi et al. (2012), "Ufo: A Framework for Abstraction- and Interpolation-Based Software Verification"

Advantages of interpretation-based strategy:

- **+** Resource efficiency & scalability in the case of static analysis.
- **+** A full control over the underlying program state representation.
- **+** A simple design of configurable algorithms and domains.
- **+** Strong theoretical and practical foundations.

Disadvantages of interpretation-based strategy:

- **–** Less efficient than execution.
- **–** Requires the interpretation of concrete semantics.
- **–** Usually does not employ optimization of abstracted code.
- **–** Difficult to compose or reuse abstraction with other tools.

State-of-the-art Approaches

In the context of static numerical analysis, the traditional (interpretation-based) approach is represented by tools like Astrée [Cou+05]. Astrée is a static analyzer designed for real-time embedded software analysis, but is limited to analyzing C programs without dynamic memory allocation or recursion. It primarily targets embedded synchronous systems [Ber+15;

[Cou+05]: Cousot et al. (2005), "The ASTREÉ Analyzer"

[Ber+15]: Bertrane et al. (2015), "Static Analysis and Verification of Aerospace Software by Abstract Interpretation"
[Bou+09]: Bouissou et al. (2009), "Space Software Validation using Abstract Interpretation"
[DS07]: Delmas et al. (2007), "Astrée: From Research to Industry"
[SD07]: Souyris et al. (2007), "Experimental Assessment of Astrée on Safety-Critical Avionics Software"
[Bla+03]: Blanchet et al. (2003), "A Static Analyzer for Large Safety-critical Software"
[MR05]: Mauborgne et al. (2005), "Trace Partitioning in Abstract Interpretation Based Static Analyzers"
[Fer04]: Feret (2004), "Static Analysis of Digital Filters"
[Fer05]: Feret (2005), "The Arithmetic-Geometric Progression Abstract Domain"
[Gur+15]: Gurfinkel et al. (2015), "The SeaHorn Verification Framework"
[Ger+19]: Gershuni et al. (2019), "Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions"
[Bra+14]: Brat et al. (2014), "IKOS: A Framework for Static Analysis Based on Abstract Interpretation"
[JM09]: Jeannet et al. (2009), "Apron: A Library of Numerical Abstract Domains for Static Analysis"
[SPV15]: Singh et al. (2015), "Making numerical program analysis fast"
[SPV17a]: Singh et al. (2017), "A practical construction for decomposing numerical abstract domains"
[SPV17b]: Singh et al. (2017), "Fast polyhedra abstract domain"
[BZ20]: Becchi et al. (2020), "PPLite: Zero-overhead encoding of NNC polyhedra"
[CDE08]: Cadar et al. (2008), "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs"
[Ahr+16]: Ahrendt et al. (2016), *Deductive Software Verification - The KeY Book*
[Sho+16]: Shoshitaishvili et al. (2016), "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis"
[CKC12]: Chipounov et al. (2012), "The S2E Platform: Design, Implementation, and Applications"
[Yun+18]: Yun et al. (2018), "QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing"
[BBH18]: Barsotti et al. (2018), "PEF: Python Error Finder"
[LRB18]: Lauko et al. (2018), "Symbolic Computation via Program Transformation"
[SMA05]: Sen et al. (2005), "CUTE: A Concolic Unit Testing Engine for C"

Bou+09; DS07; SD07]. Astrée is based on fixpoint iteration and combines various trace semantics such as non-relational and weakly relational abstract domains, along with reduced product refinement and widening/-narrowing techniques. It achieves high precision through smart handling of disjunctions [Bla+03; MR05] and the use of domain-specific abstractions [Fer04; Fer05].

Another tool, SeaHorn [Gur+15], is a software verification framework based on the interpretation that utilizes LLVM IR. It shares similarities with our approach to LLVM IR abstraction, as it focuses on the design of easily extendable abstraction for program verification. The management of abstraction in SeaHorn is provided by an abstract interpretation module CRAB (language-agnostic library for static analysis) which is designed for the generation of invariants into LLVM IR [Ger+19]. The underlying abstract interpreter for CRAB library is IKOS [Bra+14]. Likewise, numerical abstract domain library Apron [JM09], ETH library for numerical analysis Elina [SPV15; SPV17a; SPV17b], or convex polyhedra library for abstract interpretation PPLite [BZ20] provide a set of numerical abstract domains ready for use in other tools. They implement composable abstractions similar to our proposed abstractions from previous chapters.

In contrast to abstract interpretation, symbolic interpreters are distributed across a wider spectrum of approaches, encompassing fully interpretation-based, mixed, to fully compilation-based methods. We mainly distinguish two kinds of approaches, symbolic *interpretation* (IR-based) and approaches that dynamically execute the program (IR-less). The former approach is represented by tools like KLEE [CDE08], KeY [Ahr+16], angr [Sho+16] or S^2E [CKC12] that usually perform symbolic interpretation on LLVM IR. Symbolic execution of native programs has various flavors, e.g., QSYM [Yun+18] performs its analysis using runtime instrumentation, PEF [BBH18] extracts symbolic constraints using proxy objects, while SymCC [LRB18] compiles symbolic path constraints directly into the analyzed program.

This division is an oversimplification, many of these tools fall in between these two categories, for instance KLEE interprets LLVM bitcode but also utilizes dynamic native execution when interacting with external libraries and Linux kernel.

8.1.2 Compilation-based Abstraction

The goal of compilation-based abstraction is to address the drawbacks of the interpretation-based method while still retaining some of its benefits. If we consider the division of responsibilities from Figure 8.1, all except the first approach describe a compilation-based abstraction. The only distinction is how much abstract semantics is represented in the terms of concrete semantics.

The idea of performing the program analysis using program transformation is not novel. Actually, many of the first symbolic executors, like CUTE [SMA05], DART [GKS05], or EXE [Cad+08], implemented the technique by instrumenting the program under test at the source level.

In comparison to the compilation-based approach these suffered from two essential problems [PF20]:

[GKS05]: Godefroid et al. (2005), "DART: Directed Automated Random Testing"

[Cad+08]: Cadar et al. (2008), "EXE: Automatically Generating Inputs of Death"

- Source-level instrumentation ties the tool to a single programming language. Compilation-based approach, in contrast, works on the compiler's intermediate representation and is, therefore, theoretically independent of the source language.
- High-level programming languages supply rich syntax convenient for programming, but it is often inconvenient for analysis purposes. Source-level instrumentation might be achievable for simple languages or only a subset of operations, but implementing comprehensive instrumentation for languages such as C++ is challenging, and perhaps even impossible, to accomplish. Whereas performing the program instrumentation on the intermediate level simplifies the problem noticeably. The instrumentation can also leverage the compiler's tooling that performs similar analysis during optimization, like alias analysis or constant propagation.

Similarly, the model checker Spin employed a compilation-like technique already in 1997 [Hol97]. It generates an executable on-the-fly verifier from specification and model. Effectively it embeds the model checker directly into the executable, so it can leverage compiler optimization targeting the specific model under analysis.

[Hol97]: Holzmann (1997), "The model checker SPIN"

Nowadays, the prevalence of highly optimized and modular compilers has sparked a broader interest in adopting interpretation-based techniques as compilation-based methods. Since our work on symbolic computation [LRB18], multiple tools adopted a similar compilation-based strategy to symbolic execution. SymCC [PF20] compiles manipulation with symbolic path constraints directly into the analyzed program. On average, SymCC was able to run 12 times faster than KLEE and 10 times faster than QSYM. Furthermore, SymQEMU [PF21] adapted the technique to binary analysis. The most recent result is that of SymSan [Che+22], an efficient concolic execution engine based on the Clang Data-Flow Sanitizer (DFSan) framework. It outperforms SymCC and almost reaches the speed of native concrete execution. In the abstract interpretation, we are only aware of recent work on Python abstraction [CE20], which instrumented abstraction in terms of Python IR.

[LRB18]: Lauko et al. (2018), "Symbolic Computation via Program Transformation"

[PF20]: Poeplau et al. (2020), "Symbolic execution with SymCC: Don't interpret, compile!"

[PF21]: Poeplau et al. (2021), "SymQEMU: Compilation-based symbolic execution for binaries"

[Che+22]: Chen et al. (2022), "SYMSAN: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis"

[CE20]: Camacho et al. (2020), "Static Analysis of Python Programs using Abstract Interpretation: An Application to Tensor Shape Analysis"

We strive to provide a more all-encompassing solution than compilation-based symbolic executors. The goal of a generic compilation-based abstraction is to enable the usage of an arbitrary abstract domain within the program. Moreover, we aim to enable multi-domain programs. Fortunately, we have successfully devised a solution for this in the previous chapters, which are abstract metadomains (cf. Chapter 7). Using a metadomain in compilation-based abstraction, we can abstract scalars, memory objects, data structures, and pointers without extra effort. The metadomain also resolves domain interactions automatically, paving the way for further analysis extensions such as backward constraint propagation. This makes it possible to analyze programs that are typically challenging for symbolic execution, as we can apply memory abstractions to possibly infinitely large arrays or recursive data structures.

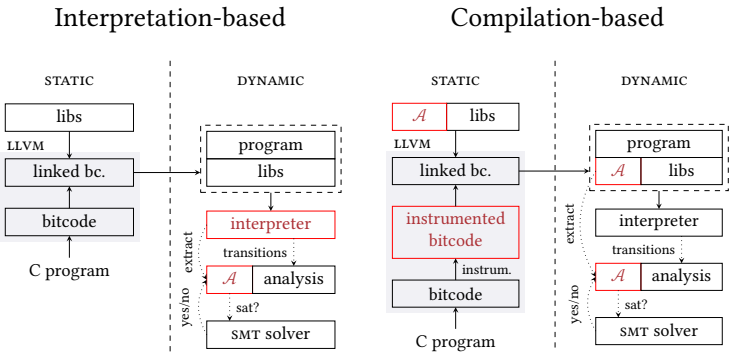


Figure 8.3: Comparison of abstract interpretation approach and compilation-based approach. All manipulations of abstract values are denoted by red color. In both cases, the interpreter generates transitions in the state space and passes them to the analysis component, which performs safety analysis. In the compilation-based approach, abstract operations are instrumented into the program, while in the interpretation-based one, they are the responsibility of the interpreter. If a domain requires a dedicated domain solver to determine the feasibility or equality of states, there is usually a dedicated component to deal with these kinds of queries.

Figure 8.3 illustrates the essential differences between interpretation-based and compilation-based abstraction. In interpretation-based abstraction, the entire abstraction process is the responsibility of the dynamic interpreter, whereas, in compilation-based abstraction, the essential component is static instrumentation, which injects abstract semantics into the program. It is worth noting that abstract data definitions and concrete implementation of abstract operations are linked to a program under analysis in compilation-based abstraction. This, in contrast to the interpretation-based approach, enables the optimization of abstract operations, which was also not possible in a purely concrete environment because the compiler could not infer properties about ambiguous program inputs. However, once we introduce the abstraction into the program through instrumentation, the program inputs are expressed as abstract constants, enabling the compiler to conduct constant folding on the inserted abstractions. It can also inline multiple abstract operations and infer better-combined abstract transformers. We will demonstrate this in the subsequent Chapter 9 devoted to syntactic abstraction. In the remainder of this chapter, we will explore how to adapt common techniques to compilation-based abstraction.

8.2 Abstract Execution

To further explore the features of compilation-based abstract execution, we need to understand that executing compiled abstraction is essential to achieve its goals.[1] This can be done through either reinterpretation or native execution. The simplest method of abstract execution involves performing generalized symbolic execution in any domain, where the program is being interpreted step by step and execution is branched based on ambiguous decisions.

1: It is challenging to conceive of any practicality of an instrumented program for static analysis since it is more effective to directly interpret the concrete program rather than a more intricate instrumented version.

In the abstract execution, we explore the implicit control flow graph given by the program. In contrast to concrete execution, branching can lead to multiple potential paths. In the context of compilation-based abstraction, multiple program paths exist if we branch on an uncertain value (i.e., a *maybe* value `M`).

As previously established in scalar abstraction, domains specify their conversion to the tristate domain. This conversion is useful for a uniform handling of potentially ambiguous path branching. Whenever the program includes a branch on an abstract value, we convert the value through

tristate conversion into a boolean. If the choice was ambiguous (`M`), we need to account for both paths. To do so, the running interpreter must split the execution paths, typically done through nondeterministic choice in verification tools. Afterward, we restrict the abstraction according to the selected choice. In the case of native execution, we can fork the execution or select a single path and re-execute the program later with a different choice.

In particular, when executing a term domain, we let the program to run computational statements to construct data definitions. Upon converting to tristate, we always return `M` and let the program explore both paths. Subsequently, the assume operation augments the path condition and calls the solver for satisfiability, i.e., the feasibility of the current location. If it is not feasible, we terminate the run.

By adopting this approach, we can centralize all of our SMT querying logic at a single point. Furthermore, if we use an external interpreter, we can generate a specific notification requesting feasibility validation and forward the path formula to the interpreter. This is a procedure that we employ in the DIVINE model checker, where we yield the path condition and allow the interpreter to determine whether to proceed. However, it is the domain's responsibility to initiate this request, so the interpreter need not verify feasibility after every step.

Although the approach is generally straightforward, there are various ways to optimize and implement abstract execution. Steinhöfel categorizes symbolic execution engines in [Ste22], which can be expanded to describe abstraction engines too (cf. Table 8.1), and serves as a valuable reference for comparing different implementations and their adaptation to a compilation-based approach.

[Ste22]: Steinhöfel (2022), "Symbolic Execution: Foundations, Techniques, Applications, and Future Perspectives"

The compilation-based abstraction falls under the category of compilation-based implementations. Due to the nature of native execution and our application to model checking, we consider only the abstraction of transition semantics. As discussed in the previous chapter, this approach does not align with traditional fixpoint-computed collecting semantics. The remaining characteristics will vary depending on the chosen domain and the overall analysis composition.

Most abstractions fall into the category of **internal theories**, where the tool is responsible for constraint solving and value representation. However, abstractions use solvers for symbolic and relational domains to assess the feasibility or equality of states or generate counter-example inputs. When using an off-the-shelf solver, the available theories and customization limit its applicability. Approaches also differ in the way how they embed an external solver in the tool. In general, we recognize two forms of embedding, namely shallow and deep embedding [GW14; SA13].

[GW14]: Gibbons et al. (2014), "Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl)"
[SA13]: Svenningsson et al. (2013), "Combining Deep and Shallow Embedding for EDSL"

Shallow embedding directly encodes symbolic values and constraints using pre-existing solver data structures. This approach is simpler, as it eliminates the necessity of creating new theories. However, shallow embedding may restrict the abstract execution engine's expressiveness since the current data structures may not accurately reflect the intended analysis use case.

Table 8.1: Characteristics of abstract execution engines [Ste22].

IMPLEMENTATION TYPE	ABSTRACTED SEMANTICS
1. Interpretation-based	1. Trace/collecting semantics
2. Execution-based	2. Denotational semantics
3. **Compilation-based**	3. Weakest precondition semantics
4. Runtime Instr.-based	4. Hoare logics
5. Using proxy objects	5. **Transition semantics**
CONSTRAINT & VALUE REPR.	**CONSTRAINT SOLVING**
1. External domains/theories	1. Off-the-shelf solver
1.1. Shallow embedding	2. With reduction / Reuse
1.2. Deep embedding	3. Non-exhaustive techniques
2. Internal domains/theories	4. Special solver
CALL TREATMENT	**LOOP / RECURSION TREATMENT**
1. Inlining	1. Widening & Narrowing
2. Summaries / Contracts	2. Bounded unrolling
3. On-demand concretization	3. Invariants
4. Compositional analysis	4. Concolic
PATH EXPLOSION COUNTERMEASURES	**PRECISION LOSS COUNTERMEASURES**
1. Summaries / Contracts	1. Relational analysis
2. Subsumption	2. CEGAR
3. State merging	3. Domain refinement

On the bright side, direct encoding allows the program to perform computations directly in the solver's language, which can lead to enhanced performance. This is especially important in the compilation-based approach, as we strive to minimize the amount of instrumented computation.

Deep embedding, on the other hand, involves the creation of a specialized abstract syntax tree for abstract values and constraints, and the formalization of operations on these values in terms of the underlying solver theories. This approach provides a high degree of customization and control over the abstract domain, but also requires a significant investment in the definition of new theories, which can lead to inconsistencies. One potential drawback of this approach is that encoding the theory computations into the program may make the abstract program overly complicated and inefficiently interpretable.

[Ahr+16]: Ahrendt et al. (2016), *Deductive Software Verification - The KeY Book*

An alternative approach is to develop a custom solver with specialized theories, as seen in the KeY [Ahr+16] tool. This approach offers the highest level of customization and control but requires a significant investment in terms of time and resources for its implementation. Moreover, this approach disconnects the tool from the advancements of general-purpose solvers.

Additionally, in the compilation-based approach, it is possible to internalize the solver not into the tool but into the program. However, this option is only feasible for simple decision procedures. Including more complex solvers such as Z3 [DB08] in the program would not be reasonable, and it is more suitable to either externalize the solver as a shared library or allow the analysis tool (interpreter) to invoke the solver instead of interpreting it.

We experimented with both shallow and deep embedding representation for our term domain. The shallow embedding was most suitable for native abstract execution, as it enabled the program to compute with the solver's term representation directly and link the solver as an external library. In comparison, we employed deep embedding for IR-based analysis of the model checker DIVINE, in which the program is interpreted on an LLVM IR level. In this approach, the program was used to construct an abstract syntax tree of terms, while the model checker managed the decision procedure. This provided the advantage of being able to select different solvers without the need to modify the domain or alter the decision procedure, for example, by using incremental solving or formula caching directly in the program.

Abstract execution faces similar challenges as symbolic execution, including the path explosion problem, complexity of memory analysis, and increasing complexity in constraint solving as the execution path grows. Furthermore, with longer execution paths, abstraction tends to lose its precision. To overcome these challenges, various strategies are employed by tools.

Summaries. Efficient countermeasures for path explosion problem are function or loop summaries. These allow us to compute an effect of loops statically before analysis or incrementally infer program invariants, function preconditions, or postconditions [BHW09; Ern+07; FMV14; FM10]. Thereafter, instead of interpreting the summarized function, we can assert on precondition and add its postcondition to program constraints. Formally, a summary of code block B forms a Hoare triple $\{P\}B\{Q\}$, where P is a precondition and Q is a postcondition. The summarization technique can be easily adapted to compilation-based abstraction. If a tool or manually written annotations provide a Hoare triple for code block B, we replace it during compilation with **`assert`** P and **`assume`** Q, requiring the realizability of conditions in the program syntax, for example, as described in [HP16]. This can be performed before syntactic abstraction, allowing us to intrinsically abstract precondition and postcondition as the rest of the program.

[BHW09]: Bubel et al. (2009), "Abstract Interpretation of Symbolic Execution with Explicit State Updates"
[Ern+07]: Ernst et al. (2007), "The Daikon system for dynamic detection of likely invariants"
[FMV14]: Furia et al. (2014), "Loop Invariants: Analysis, Classification, and Examples"
[FM10]: Furia et al. (2010), "Inferring Loop Invariants Using Postconditions"
[HP16]: Horváth et al. (2016), "Source Language Representation of Function Summaries in Static Analysis"

One can also implement dynamic program summaries, as presented in [AGT08]. In such a case, we need to instrument the program to cache function effects during the execution and query the cache on function calls.

[AGT08]: Anand et al. (2008), "Demand-Driven Compositional Symbolic Execution"

Call treatment. A whole program analysis entails a thorough exploration of all reachable function calls, which can result in significant growth of the state space. Additionally, if the program interacts with external libraries for which the implementation is unavailable, custom handling of these calls is necessary to ensure the analysis remains sound. There

are several ways to address the treatment of function calls. One approach is to utilize the previously discussed summaries that capture the effects of functions on the program's behavior. Another solution is to use a non-exhaustive technique, such as concretization, and call external functions with concrete inputs. Moreover, one can employ custom abstraction of specific system calls or functions from standard libraries as we did in M-String abstraction (cf. Section 5.3).

The technique of *concretization* is usually implemented as a concolic execution, where the program computes with symbolic representation while preserving a concrete representative for the current model [Cad+08; GKS05]. This representative can then be used as input for calls to external functions. In a compilation-based abstraction, this approach can be easily adapted by computing in a product domain $\mathcal{A} \times \mathcal{C}$, which holds both abstract values and concrete representatives.

[CKC12]: Chipounov et al. (2012), "The S2E Platform: Design, Implementation, and Applications"

One can also implement soft concretization used in S^2E [CKC12] in the compilation-based approach, where if the concretized value is used in the conditional of a branch, we backtrack to concretization and try to pick different values to explore as many paths as possible. We can implement concretization as a generator that awaits notification to create a new value. Moreover, the concretized value will be in the soft concrete domain which notifies the tool on a branch.

Another approach to handle function calls in interprocedural analysis is function summarization, which is paired with compositional analysis [God07; RHS95]. In program verification, these summaries are referred to as contracts, inspired by the *design-by-contract* concept [Mey92]. Not only do they improve the scalability of the analysis, but they also define the expected behavior of functions. In compositional analysis, functions are annotated by their contracts and then analyzed independently [Bal+18].

[God07]: Godefroid (2007), "Compositional dynamic test generation"
[RHS95]: Reps et al. (1995), "Precise interprocedural dataflow analysis via graph reachability"
[Mey92]: Meyer (1992), "Applying'design by contract'"
[Bal+18]: Baldoni et al. (2018), "A survey of symbolic execution techniques"

To compute function contracts, we can adopt a lazy approach and compute them as needed. One way to do this is to explore function dependencies and perform compositional analysis in a dependency-ordered manner. We explore this approach when paired with slicing domain ★ in Section 10.1.

Path explosion countermeasures. Besides reducing the number of possible paths to explore using summarization, the path explosion problem in program analysis can be mitigated through the use of techniques that merge multiple executions, such as subsumption and state merging. These techniques effectively reduce the number of possible paths to explore.

The literature proposes several merging techniques, including subsumption and state merging, which generally employ a disjunction of merged paths [BHW09; HSS09; Kuz+12; SHB16; Sen+15; Ste20]. Although this type of merging can be challenging to achieve in the native abstract execution, as branching creates multiple forks. It may be possible to instrument merge points where the program waits for other forks to synchronize, share data, and continue along a single path. Further investigation is needed to determine the feasibility and soundness of this approach.

[BHW09]: Bubel et al. (2009), "Abstract Interpretation of Symbolic Execution with Explicit State Updates"
[HSS09]: Hansen et al. (2009), "State Joining and Splitting for the Symbolic Execution of Binaries"
[Kuz+12]: Kuznetsov et al. (2012), "Efficient State Merging in Symbolic Execution"
[SHB16]: Scheurer et al. (2016), "A General Lattice Model for Merging Symbolic Execution Branches"
[Sen+15]: Sen et al. (2015), "MultiSE: Multi-Path Symbolic Execution Using Value Summaries"
[Ste20]: Steinhöfel (2020), "Abstract Execution: Automatically Proving Infinitely Many Programs"

Alternatively, at merge points, we can check if another execution has already covered the current path. If so, we can eliminate the current path. This can be done by keeping a shared cache of states at potential merge points. The challenge is figuring out how to compare the concrete part

of the program state in a native execution, as it is impossible to store the entire system's state. However, in IR-based analysis, it is possible because the interpreter has complete control over the program state.

Loops & Recursion treatment. In the context of abstraction, it is common to employ widening and narrowing techniques to deal with loops and recursion. These techniques are used to overapproximate the solution in cases where summarization techniques may not be applicable due to their complexity.

The standard approach involves widening to initially overapproximate the solution, followed by narrowing to refine the approximation and regain precision [CZ11; CC92; FG10; GH06]. Within compilation-based abstraction, this can be achieved by extending the abstract domain to include both a widening and narrowing operator and instrumenting their use at relevant points within the program. However, one must be careful not to discard possible executions, resulting in unsound analysis.

[Bie+03]: Biere et al. (2003), "Bounded model checking."
[KT14]: Kroening et al. (2014), "CBMC–C bounded model checker"

Bug-finding methods usually employ an underapproximation of the problem in the form of bounded analysis [Bie+03; KT14]. We can re-cast this approach into a domain to fit compilation-based abstraction. The bounded-analysis domain, instead of performing computation, would count the number of steps and terminate the analysis of the current path once a specific number of steps is reached. Such a domain can be used in a product with an analysis-focused domain responsible for actual computation and path guidance.

It is worth noting that the definition of a step may vary, and careful consideration should be given to what constitutes a step in the context of the bounded domain. For instance, the step count may be incremented only upon function calls, or, in the context of loop analysis, the counter may only be incremented upon the occurrence of branching on an abstract value, i.e., in case we perform conversion to tristate.

The benefit of using compilation-based widening and bounded analysis is that the compiler can determine the loop bound, allowing for loop unrolling and possibly even automatic summarization.

[Min04]: Miné (2004), "Weakly relational numerical abstract domains"

[Cla+00]: Clarke et al. (2000), "Counterexample-Guided Abstraction Refinement"

Precision loss countermeasures. Abstract computation, due to its approximative nature, often leads to loss of information, resulting in potential false alarms. To address this, abstraction tools use various refinement techniques. One approach is to refine the domain, for example, by adding relational information [Min04] or switching to a better-suited domain. During execution, we may mitigate the over-approximation by narrowing as described before. Another option is to use counterexample-guided abstraction refinement (CEGAR) [Cla+00], which responds to false alarms by repeating the refinement process. In compilation-based abstraction, we can examine two possible CEGAR loops, one on the level of actual program execution, where we can refine the linked domain library, for instance, by providing new predicates to predicate abstraction, or we can perform refinement on IR-level, where we instrument additional information directly into the program. These can be also arbitrarily interleaved.

8.2.1 Symbolic Execution

Symbolic execution, within the context of abstraction, is simply the execution with a term domain. It has seen broader adoption as compilation-based program analysis in recent years [Chi+09; CKC12; PF20; Sho+16; Yun+18]. Notably in binary analysis, since it is natural to use compilation-based techniques to get close to binary representation. Symbolic executors in this domain mainly focus on fast and flexible analysis on the binary level. These tools leverage the fact they can analyze large codebases without the need to understand their build process because they analyze the final binary.

Interpretation-based approaches to symbolic execution frequently present a trade-off between flexibility and performance. In contrast, compilation-based methods offer an optimal balance between performance and flexibility, as well as ease of extensibility. This combination of characteristics is particularly desirable in the context of security analysis, where auditors frequently require the ability to steer the tool toward specific objectives.

We shall now examine the process of analysis performed by compilation-based symbolic executors. Afterward, we will highlight the unique features of our compilation-based abstraction and identify the similarities between various approaches.

[Yun+18]: Yun et al. (2018), "QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing"

QSYM [Yun+18] employs a rather straightforward approach of dynamic binary instrumentation (see Figure 8.4). During program execution, QSYM utilizes Intel Pin to insert symbolic computation at the x86 machine-code level. On one side, this approach is performant but lacks flexibility due to its tight to a single architecture. Moreover, it requires implementing symbolic semantics for the whole instruction set of x86 architecture.

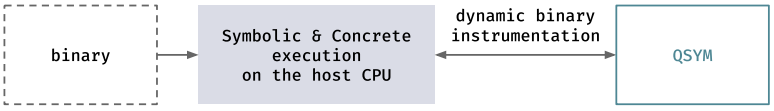


Figure 8.4: QSYM architecture.

[Sho+16]: Shoshitaishvili et al. (2016), "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis"

[NS07]: Nethercote et al. (2007), "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation"

ANGR [Sho+16] synchronously performs concrete and symbolic analysis (see Figure 8.5). It dynamically translates binary to VEX IR, the IR of the Valgrind framework [NS07], to interpret it symbolically. On the other hand, it utilizes the *Unicorn* CPU emulator to execute a fast concrete path. ANGR synchronizes both execution paths during the analysis. Due to its implementation in python and interpreted symbolic execution the analysis is not very performant, but it allows to easily extend analysis through publically exposed customization points of the interpreter.

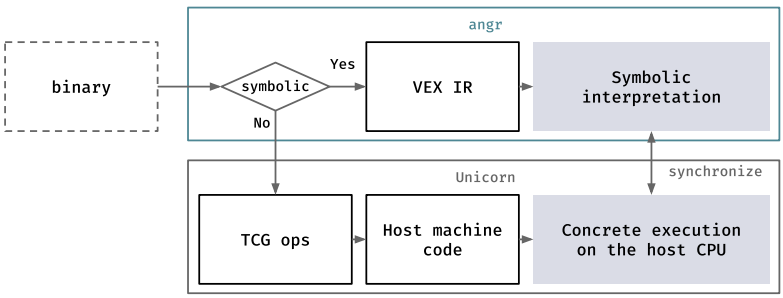


Figure 8.5: ANGR architecture

S^{2}E [CKC12] is a platform for writing tools that analyze the properties and behavior of software systems (see Figure 8.6). It comes as a modular library that gives virtual machines symbolic execution and program analysis capabilities.

[CKC12]: Chipounov et al. (2012), "The S2E Platform: Design, Implementation, and Applications"

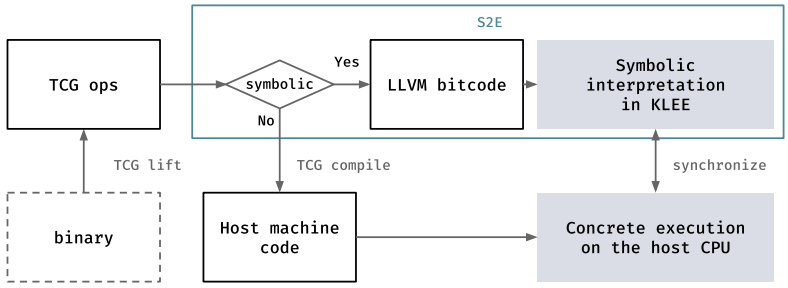


Figure 8.6: S^{2}E architecture.

S^{2}E runs unmodified x86, x86-64, or ARM software stacks, including programs, libraries, kernel, and drivers. Similarly to ANGR, S^{2}E combines concrete and symbolic execution. It utilizes QEMU [Bel05] to emulate the concrete execution of a binary and KLEE [CDE08] to interpret the program symbolically.

[Bel05]: Bellard (2005), "QEMU, a Fast and Portable Dynamic Translator"

[CDE08]: Cadar et al. (2008), "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs"

SymCC [PF20] is a compiler wrapper which embeds symbolic execution into the program during compilation, and an associated run-time support library. In essence, the compiler inserts code that computes symbolic expressions for each value in the program. The actual computation happens through calls to the support library at run time (see Figure 8.7).

[PF20]: Poeplau et al. (2020), "Symbolic execution with SymCC: Don't interpret, compile!"

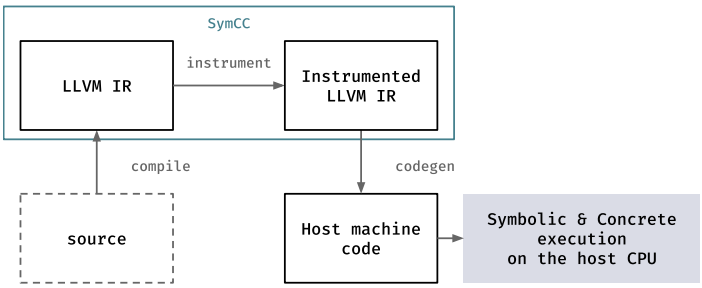


Figure 8.7: SymCC architecture.

This approach is most similar to the one presented in this thesis. The distinguishing feature of our compilation-based abstraction to SymCC approach is a broad spectrum of domains and related refinement procedures. This allows the utilization of abstraction on a wider variety of problems.

SymQEMU [PF21] combines the previous approaches to achieve both flexibility and performance (see Figure 8.8). Its approach is inspired by SymCC compilation-based symbolic execution but adapted to binary analysis. SymQEMU first translates the binary to a single intermediate representation independent of the target architecture. This gives a flexible representation that is easy to extend, analyze and maintain. But in contrast to S^{2}E and ANGR, it does not interpret the intermediate representation. Instead, SymQEMU instruments the symbolic computation into binary and executes it directly. Therefore, it also achieves performance advantage in comparison to interpretation-based approaches.

[PF21]: Poeplau et al. (2021), "SymQEMU: Compilation-based symbolic execution for binaries"

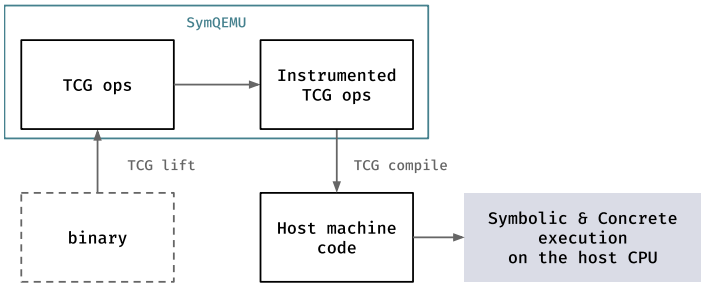


Figure 8.8: SymQEMU architecture.

LART. While our approach is primarily geared towards source code analysis, it is also possible to elevate binaries to LLVM and apply our tool to abstract them. In fact, this technique was utilized to conduct symbolic execution of binary programs [Kor+20]. The key feature that distinguishes our approach from prior methods is its modularity – we are not restricted to a specific domain or backend analysis tool. We can conduct native analysis, like the previous executors, but the components can also be used off-the-shelf with other interpretation tools. This is due to the fact we decouple abstract semantics, runtime library, and instrumentations into standalone components.

[Kor+20]: Korenčik et al. (2020), "On Symbolic Execution of Decompiled Programs"

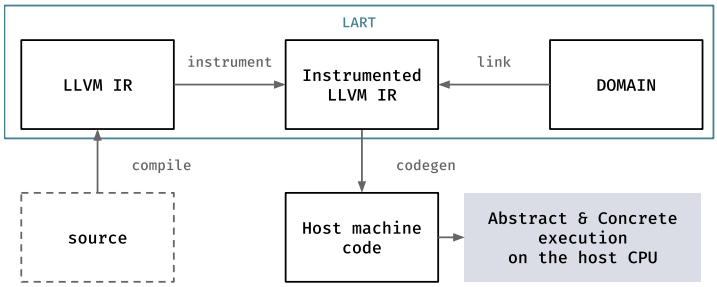


Figure 8.9: LART architecture for native execution.

We have shown that compilation-based abstraction execution can be adapted to many standard techniques. These techniques include bounded analysis instrumentation, widening, and path exploration guidance, which have the potential to improve performance, reduce dependence on interpreter design, and optimize abstraction using compilation. Furthermore, our approach is not limited to abstract execution, as we will illustrate in the next section where we can seamlessly apply the technique to abstract model checking.

8.3 Abstract Model Checking

The abstract model-checking presented in this section was inspired by the *control-explicit data-symbolic* approach of tool SymDIVINE [Mrá+16]. Compared to the exhaustive enumeration of states in explicit-state model checker DIVINE, the *control-explicit data-symbolic* approach tries to group states into sets when they differ only in data values but not in the control location. In this way, SymDIVINE is able to represent inputs as sets of

[Mrá+16]: Mrázek et al. (2016), "SymDIVINE: Tool for Control-Explicit Data-Symbolic State Space Exploration"

possible values. These sets, also called multi states, are described by an explicit control location and symbolic representation of data.

SymDIVINE's exploration algorithm operates directly on multi-states. To explore the state space, SymDIVINE must determine whether two multi-states represent the same set of explicit states, which can be time-consuming. However, the symbolic approach can result in exponential memory savings and prevent state space explosion caused by inputs.

We expand the approach to the arbitrary domain and augment the explicit-state model checker DIVINE by abstraction capabilities using the compiled abstraction (for details about DIVINE refer to Section 10.1). The fundamental concept of compilation-based abstraction for model checking remains consistent with the interpretation or native execution of abstracted programs presented before. We instrument a desired abstraction into the program, making it effectively operates within an abstract state-space. And the model checker proceeds to explore this abstract state-space of the program. Nevertheless, we need to modify the approach a bit.

The key difference between explicit-state model checking and program execution is the necessity of model checkers to bookkeep their states and determine their equality. One way to do this is by directly comparing in-memory states. However, this approach can lead to false negatives as it considers semantically irrelevant details, such as numerical addresses returned from allocations. In such case, the model checker determines the inequality of states, even though they are equal from the semantic point of view. A better approach is to represent memory as a graph, which allows for analysis independent of memory layout [Roč+18]. Alternative methods suggest overapproximation techniques like memory-hamming distance or hashing [Gut12; GQC15]. When incorporating compilation-based abstraction in explicit-state model checking, it is important to consider the implications on state equality checking, and to design representations and techniques that allow for efficient comparison of abstract states.

[Roč+18]: Ročkai et al. (2018), "DiVM: Model checking with LLVM and graph memory"

[Gut12]: Guthmuller (2012), "State equality detection for implementation-level model-checking of distributed applications"

[GQC15]: Guthmuller et al. (2015), "System-Level State Equality Detection for the Formal Dynamic Verification of Legacy Distributed Applications"

The approach we are discussing is commonly referred to as a set-based model checking, as we are effectively computing with a set of states represented abstractly. To simplify the graph-based memory-layout view, we will observe a program state only as an n-tuple of values. This can naturally be extended to the graph-based approach used in DIVINE [Roč+18].

8.3.1 Equality of Abstract States

In explicit state model checking, the equality check is performed on a per-value basis. This means that two states, $\sigma = \langle v_1, \dots, v_n \rangle$ and $\sigma' = \langle v'_1, \dots, v'_n \rangle$, are equal if and only if all of their values are pairwise equal:

$$\forall_{i=1}^{n} v_i = v'_i.$$

A direct equality check can be used to compare concrete states, but this approach is not applicable to abstract states. For example, a symbolic state with a data definition that reverses operands, such as $x+1$ and $1+x$, will have different in-memory representations, but they represent the same set of concrete states. Additionally, abstract states can subsume each

other. For instance, a state with the path condition $x > 5$ is subsumed by a state with the path condition $x > 0$.

To ensure a sound analysis of temporal properties, it is necessary to detect loops in the program state space. Thus, we need to be able to identify already explored states and avoid re-exploring them. Moreover, in this way, we minimize the number of explored paths.

Definition 8.3.1 *An abstract state $\hat{\sigma}$ subsumes state $\hat{\sigma}'$ iff $\gamma(\hat{\sigma}) \subseteq \gamma(\hat{\sigma}')$.*

As previously mentioned , states of abstracted programs are essentially the Cartesian product of concrete and abstract values. Therefore, we can check for subsumption at the level of individual elementary values. Specifically, $\hat{\sigma}$ subsumes $\hat{\sigma}'$ iff $\gamma(\hat{v}_i) \subseteq \gamma(\hat{v}'_i)$ holds for all elementary values $\hat{v}_i$ and $\hat{v}'_i$ of compared states.

However, in practice, we do not want to perform concretization to decide whether $\gamma(\hat{v}_i) \subseteq \gamma(\hat{v}'_i)$. We can employ domain-specific ordering instead, which specifies the subsumption order between two values within the domain. For example, in the interval domain, we can solve subsumption by comparing the bounds:

$$[\underline{a}, \overline{a}] \subseteq [\underline{b}, \overline{b}] \iff \underline{b} \leq \underline{a} \wedge \overline{a} \leq \overline{b}.$$

Definition 8.3.2 (Abstract Subsumption) *Consider $\sigma = \langle v_1, \ldots, v_n \rangle$ and $\sigma' = \langle v'_1, \ldots, v'_n \rangle$ and abstract domain $\mathcal{A}$:*

$$\sigma \subseteq \sigma' \iff \forall_{i=1}^{n} \gamma_{\mathcal{C}}(v_i) \subseteq \gamma_{\mathcal{C}}(v'_i) \iff \forall_{i=1}^{n} \alpha_{\mathcal{C}}(v_i) \sqsubseteq \alpha_{\mathcal{C}}(v'_i)$$

where we resolve mixed comparisons as

$$\gamma_{\mathcal{C}}(v) = \begin{cases} \{v\} & \text{if } v \text{ is concrete} \\ \gamma_{\mathcal{A}}(v) & \text{otherwise} \end{cases} \quad \Bigg| \quad \alpha_{\mathcal{C}}(v) = \begin{cases} \alpha_{\mathcal{A}}(v) & \text{if } v \text{ is concrete} \\ v & \text{otherwise} \end{cases}$$

8.3.2 Symbolic Model Checking

One of the main goals of this thesis was to conduct control-explicit data-symbolic model checking using compilation-based methods. This means that control flow is resolved explicitly, while data are represented symbolically in the term domain [Mrá+16]. This approach is different from traditional symbolic model checking, where the entire program is encoded as a symbolic formula.

[Mrá+16]: Mrázek et al. (2016), "Sym-DIVINE: Tool for Control-Explicit Data-Symbolic State Space Exploration"

In the abstraction, there are two options for performing symbolic computations. One is to directly compute in bit-vector theory, which results in a **tight coupling** between the term representation and the decision procedure. For example, one can directly operate with the term representation used in the Z3 SMT solver. This approach benefits from optimized communication between the program and decision procedure, as there is no

need for translation between program terms and solver terms. However, it limits the abstraction to a particular theory and the solver's capabilities. Furthermore, the actual term representation in memory may not be suitable for observing the memory used by the program under test, and the term representation may need to be optimized for the purposes of analysis. A compact and lightweight representation is crucial for verification, as model checkers need to retain millions of states.

Alternatively, with a **loose coupling** approach, there will be two separate representations of terms. Like the subterm domain, the program will build ground terms using a set of defined function symbols. The analysis tool will be responsible for extracting the terms from the program and converting them into the desired theory and representation for the solver. This approach offers more flexibility and allows for optimizing the term representation specifically for the needs of the compilation-based abstraction. However, this comes at the cost of having to perform term translation. We explore both approaches in our implementation.

The loose coupling opens another question: What is the best term representation for compiled programs? One option is to represent terms as trees within the program. This approach is efficient as the operation simply creates a new root (application of function symbol) and adds the arguments as its children. However, it also presents challenges, as values may suddenly share data definitions across multiple variable representations that span across function calls, memory, and even thread executions. Therefore, one needs to be careful about how to make atomic changes and collect unnecessary values efficiently. Another option is to give term values a value semantic, in which case each operation will need to make a copy of its arguments and recreate a whole new term for the return value. This approach may lead to larger chunks of memory being copied as terms become bigger, but a higher level of abstraction on top of terms can minimize their size while still maintaining their soundness [Yao+21].

[Yao+21]: Yao et al. (2021), "Program Analysis via Efficient Symbolic Abstraction"

Regardless of the representation, the term domain presented in Section 4.3 makes it easy to extract the terms from program state and convert them to a form appropriate for further processing by the analysis tool. Recall that one of the motivating applications of the proposed approach was symbolic execution and model checking. In this case, the state space is explored by an explicit-state interpreter and the extracted terms are converted into SMT queries. To this end, the interpreter must be slightly extended and coupled to an SMT solver, since:

1. transitions of the program must be checked for *feasibility*,
2. the state *equality check* must compare terms semantically, not syntactically.

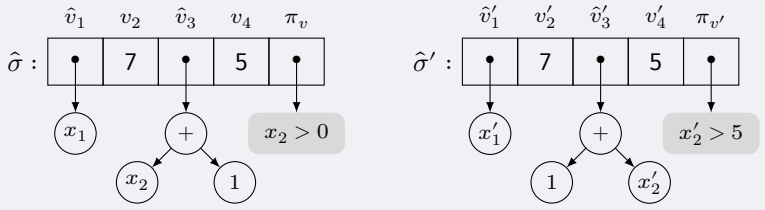
Of course, the hitherto extracted terms must be left out of byte-wise comparison that is performed on the remaining (concrete) parts of program states.

Definition 8.3.3 *We say two semi-symbolic states* $\sigma = \langle v_1, \dots, v_n, \pi \rangle$ *and* $\sigma' = \langle v'_1, \dots, v'_n, \pi' \rangle$*, where* π *and* π' *are equal if:*

$$\sigma = \sigma' \iff \pi \wedge \pi' \wedge \forall_{i=1}^{n} v_i = v'_i$$

In our implementation, we prove the disequality instead of equality by querying the formula negation, as the solver finds existential proof more easily. Additionally, we optimize the query to avoid proving path condition satisfaction since we already know that all visited states are feasible. We illustrate term equality check in Example 8.3.1.

Example 8.3.1 Consider two semi-symbolic states $\hat{\sigma}$ and $\hat{\sigma}'$:



States consist of four variables and a global path condition π. We split the subsumption check into concrete and abstract part:

- **Concrete equality** is straightforward, as we perform the explicit pairwise in-memory comparison: $v_2 = v'_2 \wedge v_4 = v'_4$.
- **Abstract equality** recreates formulae for path condition and data definitions:

$$\begin{aligned} &\hat{v}_1 = x_1 \wedge \hat{v}_3 = x_2 + 1 \wedge \\ &\hat{v}'_1 = x'_1 \wedge \hat{v}_3 = 1 + x'_2 \wedge \\ &\hat{v}_1 = \hat{v}'_1 \wedge \hat{v}_3 = \hat{v}'_3 \wedge \\ &x_2 > 0 \wedge x'_2 > 5 \end{aligned}$$

Furthermore, we optimize the equality check by minimizing the number of SMT queries. We first perform a cheap in-memory comparison of the concrete parts of the two states. If the concrete memory is not identical, we can immediately conclude that the states are not equal, and we avoid performing an expensive SMT query. Only if the concrete memory is identical, we proceed to perform an SMT query to check the equality of the abstract parts of the states.

As a matter of fact, in compiled abstraction, we do not let the program bookkeep all states, but we let the interpreter orchestrate the whole state space exploration and bookkeeping. Therefore, we need to inform the interpreter about which parts of the program state are concrete and which are abstract since the program appears entirely concrete from the outside.

Other domains are typically less computationally demanding than the term domain, which requires an entire SMT solver to decide feasibility or subsumption. Feasibility is often not a concern in these other domains, as they naturally constrain computed values using instrumented assumptions. However, one can use a domain solver to increase the precision of specific domains. Subsumption, on the other hand, may require a similar mechanism as in the term domain to extract marked values and provide a dedicated subsumption procedure, as abstract values are rarely structurally comparable.

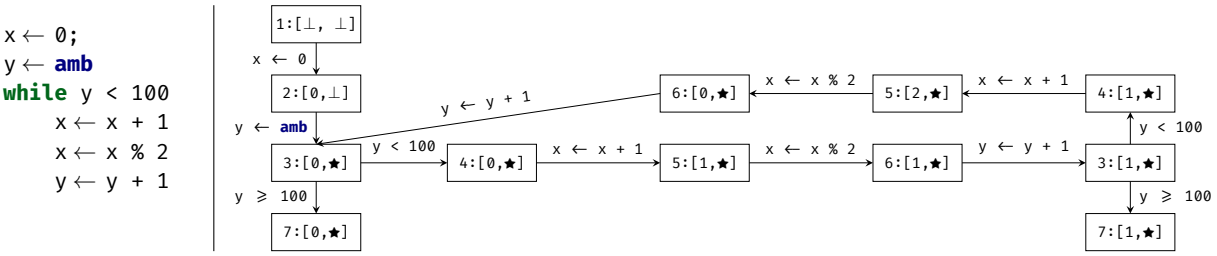


Figure 8.10: Example of unit abstraction state space for program on the left. Each state is represented as `l:[x, y]`, where `l` is a control location and `x` and `y` are values of variables. Notice that the unit abstraction reduces state space size, as after two iterations, we observe the already visited state. In comparison, a primitive abstract execution would branch infinitely.

The unit domain is one of the few domains that can be structurally compared, as it contains only a single value that always subsumes itself. The only situation it needs to resolve is when one state contains a concrete value on a location, whereas the other has a unit value. In such a case, we need to claim the former state to be subsumed by the latter. Otherwise, we can conduct a purely explicit-state model checking that employs just in-memory comparison. We illustrate a unit state space in Figure 8.10.

Other domains that have a disjunctive description of abstract values can also use almost purely structural subsumption. For instance, the sign domain describes values as disjunctive sets such as `<0`, `0`, and `>0`. The only exception is for $\top$ values, which subsume other abstract values and require special treatment.

Summary

The present chapter discussed possible approaches to program abstraction, with a primary focus on the compilation-based technique, which forms the foundation of the analyses presented in this thesis. This technique involves encoding abstraction into the program itself, enabling the execution of the program in the provided abstract semantics. We have shown that it is possible to translate abstract semantics and various refinement techniques into as an abstract domain and effectively apply them in the context of compilation-based abstraction.

In particular, we discussed how symbolic execution and model checking can be adapted into the compilation-based abstraction framework, as well as other approaches such as summarization and bounded analysis. We compared these techniques to their more common interpretation-based definitions and highlighted the advantages and disadvantages of each.

9 Syntactic Abstraction

This chapter provides a detailed technical overview of syntactic abstraction for abstract execution. The approach referred to as syntactic abstraction involves modifying the program being analyzed entirely at the syntactic level during program compilation. The abstraction is accomplished by inserting abstract operations into the program without their accompanying semantics, resulting in a semi-abstract program. The semantics of these operations are provided later in the process through the domain linked to the analyzed program.

In the process of syntactic abstraction we need to consider various aspects of the interaction between abstraction and the concrete environment and program execution. Specifically, the syntactic abstraction must address the interaction with the control flow of the program and the program's memory. We will provide intuition into syntactic abstraction in this section, which will be elaborated on in greater detail later in this chapter on a per-component basis.

As mentioned in the previous chapter, it is advantageous to divide the transformation process into two stages using intermediate abstract instructions. These instructions work with abstract values and produce abstract values as output. It is crucial that each abstract instruction can be translated into a sequence of concrete instructions. This is what makes it possible to obtain an abstract program without any abstract instructions and run it using standard concrete methods. One can think of concrete realization as semantic implementation in the programming language using the toolset of the concrete domain. In the following, we expect that the abstract semantic of domains is realizable in the concrete domain. Nevertheless, the syntactic abstraction is completely realized on the former semantic-less representation, and the semantics are later provided through a specific metadomain (cf. Chapter 7).

In order to maintain optimal performance, we aim to selectively abstract only certain program values. In fact, we strive to minimize the amount of abstract instrumentation performed. It is safe to keep operations in the concrete domain (unaffected by instrumentation) as long as they do not interact with the environment or participate in ambiguous computations.

It is impractical and potentially uncomputable to detect operations that involve abstract values precisely. Thus, it is necessary to overapproximate the set of such operations. To achieve this, we statically identify a set of potentially abstract operations and only instrument these. By performing a dataflow analysis (cf. Section 3.7) starting from input values (uniformly represented as `amb` operation), we can identify the set of operations, which may be involved in the abstract computation. Consequently, we can omit the abstraction of possibly complex concrete computations.

Now, given the set of possibly abstract operations, we perform a syntactic abstraction — instrumentation that replaces the concrete operations with abstract alternatives. However, these operations do not compute in any particular domain. They are technically realized as function calls to a

❶ Concrete program:

```
x ← amb
y ← 100
if x < 10
    y ← call f(y)
    x ← x + y
else
    y ← call f(x)
    x ← x + y
```

❷ Operations abstraction:

```
x ← amb
y ← 100
if x ⌃< 10
    y ← call f(y)
    x ← x ⌃+ y
else
    y ← call f(x)
    x ← x ⌃+ y
```

❸ Constraints instrumentation:

```
x ← amb
y ← 100
if x ⌃< 10
    x ← assume x < 10
    y ← call f(y)
    x ← x ⌃+ y
else
    x ← assume x >= 10
    y ← call f(x)
    x ← x ⌃+ y
```

Figure 9.1: Example of syntactic abstraction.

1: Even if the metadomain consists of a single abstract domain.

library that implements the domain. To give these operations meaning, we will link a desired abstract domain in the end, just before the program analysis. This allows us to perform instrumentation only once but execute the analysis with various domains. We depict steps of syntactic abstraction of program ❶ in Figure 9.1. Abstract operations are marked with ˆ symbol in program ❷.

Another characteristic of abstract programs is their control flow. In concrete computation, a particular input always determines a single program path. However, in abstract execution, this is not the case. When branching on abstract values, all possible paths must be taken in order for the analysis to be sound. For unconstrained input x in ❸, this means that both conditional paths are viable. Further, it is necessary to constrain the program's variables that are dependent on the condition, in order for them to acknowledge that a particular path has been taken in the program.

Symbolic execution utilizes a path condition to maintain constraints, which can also be adopted in abstract execution. However, this would require each domain to incorporate its own solver to establish the feasibility of constraints. Instead, we have opted for a strategy in which each domain implements its own solution for constraining values. This is accomplished by instrumenting assume operations following each abstract branch and allowing the domain to provide its own mechanism for constraining values. For instance, backward constraint propagation (cf. Section 4.4.2) is achieved through such a custom assume implementation.

It is important to note that the syntactically abstracted operations do not operate in a single abstract domain, but rather in a union of the concrete and abstract metadomain ($\mathcal{C} \cup \mathcal{A}$).[1] This is because there may still be instances where an operation encounters a combination of arguments, i.e., both concrete arguments when reached by one path and abstract arguments in another case.

In LLVM IR, it is not straightforward to perform the mixed-computation over concrete and abstract values, because it does not provide any mechanism to work with union types. Moreover, abstract values do not typically fit in the memory reserved for concrete values. To tackle this problem, we instrument a dual abstract computation side-by-side with concrete computation — see Figure 9.2. To distinguish between concrete and abstract values, we utilize a third value (flag) that indicates whether to perform abstract or concrete computation. This flag is akin to the metadata that interpreters typically retain internally, and we refer to it as *taint*. A value marked as *tainted* is treated as abstract. By employing this method, we can maintain the original program structure, which simplifies subsequent analysis and instrumentation.

In order to further enhance efficiency and perform abstract operations only when necessary, we utilize dynamic dispatch which decides, based on the taints, whether to perform an abstract operation. If values are not tainted, we can revert to the original concrete computation. This strategy is particularly useful in cases where it is uncertain whether an operation will be abstract or concrete, such as when a function is invoked with both abstract and concrete parameters or when operands are retrieved from memory. Syntactic abstraction involves synthesis of operations that manage these interactions and calling abstraction functions if only a

subset of operands is tainted (cf. synthesized addition in Figure 9.2). This is necessary since we require abstract operations to be contained within a single (meta)domain (for additional information about metadomains, please refer to Chapter 7). In syntactic abstraction, we only resolve interaction between the concrete domain and abstract metadomain. The interaction between multiple abstract domains is resolved at the metadomain level. We will describe the precise dispatch mechanism in the next section.

Finally, we must address the interaction between abstraction and the program's dynamic memory and stack. The challenge at hand is that abstract representations frequently do not align with the concrete memory layout. Since program computation often depends on specific value offsets of in-memory objects, inserting abstract values into concrete memory objects is not straightforward and will change the program memory layout. To address this issue, we employ shadow memory, which maintains abstract memory objects assigned to specific concrete locations. As a result, we also need to syntactically abstract all operations that store or load abstract values to/from memory to access our specific shadow address space. Similarly, when we invoke functions with abstract parameters, we do not want to modify the function interface. To address this issue, we use a shadow stack, which is used to pass abstract parameters and their taints to and from the function.

It should be noted that, as discussed in the previous chapter, taints and shadow memory are a form of metadata for the computation, and if the interpreter provides a more efficient representation, this responsibility can be shifted from the program to the interpreter. In fact, we explored both possibilities. For instance, in the case of explicit state model checking, we let the interpreter automatically propagate taints and maintain shadow memory, whereas, in native abstract execution, we let the program maintain all its metadata. The former approach is advantageous for IR-based approaches since the interpreter does not need to re-interpret semantically irrelevant metadata-related operations and can omit shadow memory from the program state representation.

To summarize, the static part of the compilation-based approach involves the following steps:

1. Conduct dataflow (DFA) and alias analysis (AA) to identify the set of abstract operations.
2. Perform syntactic abstraction on these operations.
3. Instrument constraint propagation.
4. Instrument shadow memory and stack operations.
5. Synthesize domain interactions.
6. Apply generic compiler optimization.

The steps outlined above are not fixed and can be extended for specific analysis needs. For example, in refinement, we can instrument additional constraints based on previous analysis executions to further refine our analysis and improve its accuracy. This allows us to adapt our approach

```
x ← amb
y ← x + 1
```

The abstract operation takes a triplet of values, representing concrete, abstract, and taint value for each original argument. The abstract addition takes care of abstracting concrete (untainted) values.

```
x ← amb
x̂ ← ⊤ /* in 𝒜 */
tx ← T /* taint */

y ← x + 1
ŷ ← (a, â, ta) +̂ (1, ⊥, F)
ty ← tx or F
```

Synthesized implementation of addition:

```
fn +̂(x, x̂, tx, y, ŷ, ty)
   if !tx and !ty
       ret ⊥
   if !tx
       x̂ ← α(x)
   if !ty
       ŷ ← α(y)
   ret x̂ +̂_𝒜 ŷ
```

Figure 9.2: Example of value triplets. Note that we first perform lifting to abstract domain $\mathcal{A}$ and later call addition from the domain.

to the specific requirements of the program and to achieve more precise and reliable results.

Depending on the analysis requirements, certain instrumentation can be omitted and instead rely on the runtime to provide the necessary implementation. One example of this is the omission of shadow memory instrumentation ❹ in IR-based analysis, if the interpreter is able to handle its representation.

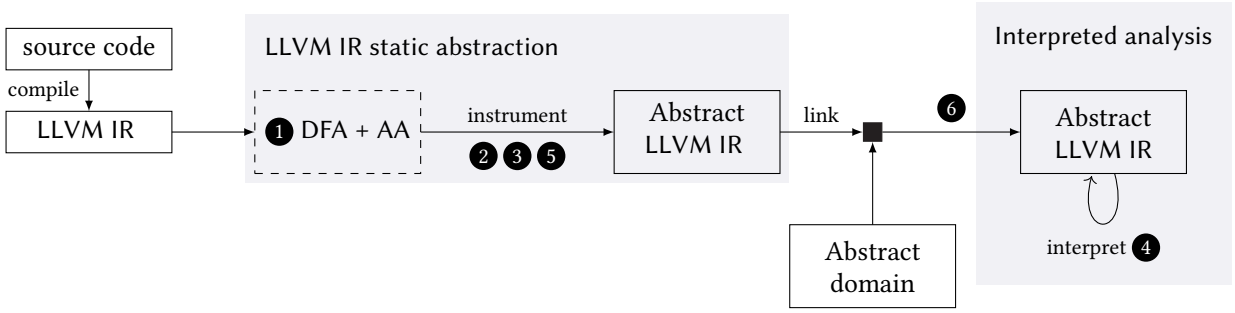


Figure 9.3: IR-based abstraction workflow with denoted steps of syntactic abstraction.

A completely tool-independent abstraction can be implemented as IR-to-native abstraction, where we instrument all abstract computation and memory handling into the program and let it perform analysis natively, as shown in Figure 9.4.

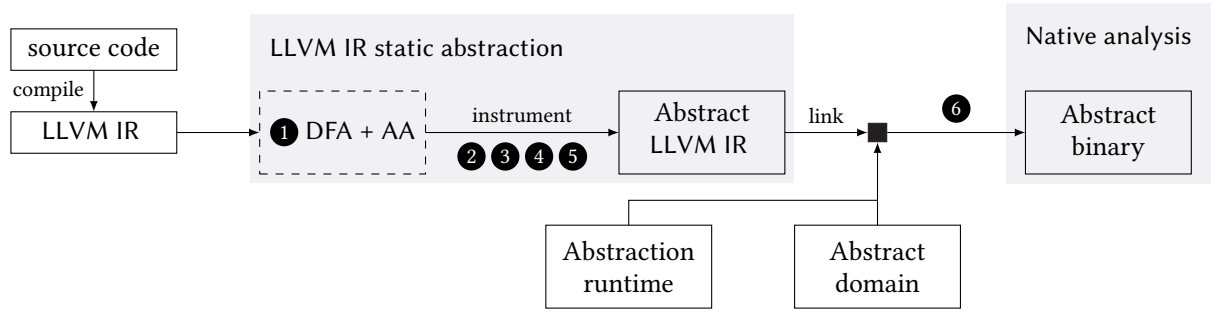


Figure 9.4: IR-to-native abstraction workflow with denoted steps of syntactic abstraction.

9.1 Data Flow

The applicability of compilation-based abstraction relies heavily on data-flow analysis to reduce the amount of runtime abstraction required. The primary objective of dataflow analysis is to identify which operations should be syntactically abstracted. *Reaching abstraction analysis*, as described in Section 3.7, can be used for this purpose. The result of reaching abstraction analysis indicates whether a value has an internal origin (it is produced purely by constant values in all program paths), an external origin (it originates from `amb` in all program paths), or a mixed origin (it can have both internal and external origins). These three categories of values are denoted as v for internal origin, $\hat{v}$ for external origin, and $\overline{v}$ for mixed origin. In this case, we consider all external origins abstract and internal ones concrete.

We leverage these origin classifications to distinguish which operations to abstract syntactically. If the operands of an operation have external

or mixed origins, it indicates that there might exist a path in which the operand is abstract. In such cases, we let the syntactic abstraction instrument an abstract operation in its place.

Furthermore, the reaching abstraction analysis also computes information about the abstractness of memory content. We define $\mathsf{P}(v)$ as a concrete pointer to must-be-concrete content, $\hat{\mathsf{P}}(v)$ as a must-be-abstract pointer to concrete content, and $\overline{\mathsf{P}}(v)$ as a pointer with a mixed origin. Additionally, we retain information about the origin of the content that the pointer values point to. For instance, $\mathsf{P}(\hat{v})$ is a pointer with a concrete origin that points to an abstract value.

This information is used to syntactically abstract operations related to shadow (abstract) memory. We consider two cases: 1) if the source of a load operation has abstract content, but the pointer is concrete, we instrument an operation to retrieve the value from shadow memory, 2) however, if the source value is abstract, we create an abstract access operation and let the domain return the content value. For example, the content of M-String values is not stored in shadow memory but directly in an abstract aggregate value.

The difference between abstract and shadow memory is that shadow memory stores abstract values at concrete addresses and can be accessed in a "usual" way, and it only serves the purpose of not altering the real memory layout. In contrast, abstract aggregates may define their own schemas for memory accesses. For instance, M-String allows accessing values at abstract offsets. In cases where the pointer has mixed origin, we must consider all three cases of memory access: concrete, shadow, and abstract.

Regarding writing to memory, we apply a similar distinction as in the case of reading. Specifically, we consider whether to perform a write into concrete memory when both operands are concrete and contain concrete content. We write to shadow memory if the value is abstract and the pointer is concrete. Lastly, we write to an abstract memory (aggregate) if the pointer is abstract.

It is worth noting that in the case of writing a concrete value to a location with potentially abstract content, we still need to access shadow memory to destroy any potentially stored abstract value.

Finally, we must consider function call abstraction. Whenever the parameters of a function call might be abstract, we synthesize a push to the abstract stack and retrieve them at the function entry point. Similarly, we must treat return values from functions.

In cases where an abstract domain provides a custom implementation of entire functions, such as the case with the M-String domain and `libc` string functions, it is important to differentiate whether to call the abstract operation at a given location when the operands satisfy the required conditions. Each domain specifies the constraints that these functions need to satisfy, such as whether they require abstract pointers, scalars, etc. For example, in the M-String abstraction, we want to avoid calling abstract libc functions for $\mathsf{P}(\hat{v})$ values, even if the content is abstract. Instead, we only want to apply the abstract function to abstract aggregates, which are denoted as $\hat{\mathsf{P}}(x)$ or $\overline{\mathsf{P}}(x)$.

When instrumenting a **`store`** v → p operation, we classify it according to the abstraction category of the pointer value of its destination. Observe that we do not have to take into account the category of the value being stored (v), as it is already included in the pointer category through the reaching abstraction analysis that propagated the category to the destination pointer.

pointer p	Store Category
$\mathsf{P}(x)$	into concrete memory
$\mathsf{P}(\widehat{x})$	into shadow memory
$\mathsf{P}(\overline{x})$	into shadow memory
$\hat{\mathsf{P}}(a)$	into abstract aggregate
$\overline{\mathsf{P}}(a)$	into abstract aggregate

In the table, x refers to arbitrary concrete content, $\widehat{x}$ denotes arbitrary abstract content, and $\overline{x}$ denotes arbitrary mixed content. Finally, a represents entirely arbitrary content, concrete, abstract, or mixed. In the case of abstract pointers, we carry out a store operation on the abstract aggregate regardless of the content category.

Figure 9.5: Store classification.

Based on dataflow analysis, three categories of interactions with program execution can be distinguished, which are addressed in syntactic abstraction:

1. *computation interactions*: this category includes computational statements and interactions with the concrete domain, where concrete values need to be lifted to the abstract domain, and interactions between abstract domains need to be resolved.
2. *control flow interactions* : this category includes conversion to tristate values (cf. Section 4.2.2) and concretization to conditional booleans for program branches, followed by instrumentation of constraint application on abstract values.
3. *memory interactions*: this category involves the manipulation of shadow memory and abstract aggregates, as well as the manipulation of the abstract stack in the case of function calls.

In the following sections, we will discuss various aspects of these interactions.

9.2 Domain Interaction

In abstract programs, we want to allow simultaneous execution of abstract and concrete semantics. We achieve that by computing in the union domain $\mathcal{C} \cup \mathcal{A}$ of concrete and abstract values. However, we require that operations are performed in a single domain, either concrete or abstract. Similarly, we want to be able to have values in multiple abstract domains at the same time. From the point of semantics, it is a computation in a union of multiple abstract domains $\bigcup_{i=1}^{n} \mathcal{A}_i$. However, concerning syntactic abstraction, we can consider that the union is a single metadomain $\mathcal{A} = \bigcup_{i=1}^{n} \mathcal{A}_i$ that internally resolves interactions between domains. Hence, during the instrumentation process, we only need to differentiate between a single concrete domain and a single abstract metadomain.

It is important to note that using the runtime abstract computation can introduce overhead as it involves almost always more complex computation, potentially even an invocation of an SMT solver. Just the call to the library of domain implementation represents overhead compared to the execution of single instruction. There are two phases in a compilation-based approach where it is possible to determine if data is concrete: (1) during compilation and (2) during execution. If data is identified as concrete at either of these stages, it is not necessary to involve abstract computation.

At *compile time* to distinguish between concrete and abstract execution we employ dataflow analysis described in the previous section. However, there are situations where the compiler cannot determine at compile time whether data will be concrete or abstract at runtime (marked as mixed in reaching abstraction analysis). For example, data that is read from memory may be either abstract or concrete, and its concreteness may change during execution. In these cases, it is only possible to check dynamically at runtime whether all operands to an operation are concrete and prevent the invocation of the abstract library if necessary.

Since low-level language like LLVM does not provide direct support for union types, we had to develop a mechanism to distinguish between concrete and abstract domains at runtime with the smallest overhead possible. Let us consider only register values for now.

We require three pieces of information for each potentially abstract value: ❶ a taint flag to denote whether the "union" contains a concrete or abstract value, ❷ the concrete value, and ❸ the abstract representation. Therefore, for each register `r` marked as possibly abstract by the dataflow analysis, we introduce two new registers `tr` to store taint and $\hat{r}$ to store abstract value. To propagate flags, we will use classic taint propagation techniques during runtime. When a value is considered concrete, its assigned taint is `F`, and the abstract value is represented as $\bot$. Likewise, all abstract values have taint set to true (`T`), and initially unconstrained abstract values are represented as top $\top_{\mathcal{A}}$. To clarify this, examine Figure 9.6.

In the IR-to-native analysis, we instrument everything into the IR and let the program manage the taint propagation. However, in the IR-based analysis, every interpreted instruction counts. Moreover, branching is usually expensive, so we want to avoid branching on each instruction.

In DIVINE, we utilized the fact it can assign metadata to values in the interpreter. We stored a single-bit taint internally for each value and let it perform taint propagation automatically. Moreover, we introduced a special kind of instruction that only got executed when at least one of the parameters was tainted. This effectively means that the interpreter executes only cheap bit manipulation with taints. The computationally expensive process of the so-called *lifter* ❹ is only executed if any of its operands are tainted (abstract). In this case, we can also omit the first conditional branch in the lifter, as the interpreter guarantees it to be satisfied, because DIVINE executes specially marked operations only in case of tainted operands. In the syntactic abstraction, we synthesize lifters for all operations automatically. They always take care of lifting parameters to the best-fit domain, using meta-domain abstraction, and call the specific operation from the domain.

The drawback of this method is that we continue to carry out redundant checks. As illustrated by the chain of abstract operations in Figure 9.7, at each operation, we assess whether the operation is abstract and determine if any of its operands need to be lifted. However, the fact that the first operation is abstract implies that all subsequent operations are also abstract, and there is no requirement to lift their operands. This problem can be addressed by using a compiler that infers transitive properties and optimizes the abstracted code, effectively eliminating unnecessary checks. The only issue that may arise is when taints originate from memory, as the compiler cannot infer their source. In Section 9.6 at the end of this chapter, we will delve further into this approach and analyze the impact of compiler optimizations on syntactic abstraction.

It is important to note that the distinction between concrete and abstract values in programs is only theoretical, as all values are, in fact, realized concretely, e.g., intervals as two integer values, and operations on them modify both interval bounds (concrete values) in a single abstract transformation step – see Figure 9.8.

Consider a program:

```
a ← amb
b ← 10
c ← a + b
```

There are multiple possibilities for how to instrument the abstraction into the program. In theory, we want the program to have semantics similar to the following:

```
a ← amb
b ← 10
ia ← is_abstract(a)
ib ← is_abstract(b)
if ia or ib
    c ← α(a) +̂ α(b)
else
    c ← a + b
```

Due to the fact the abstract values may not fit into concrete registers, we create dual values and taints to distinguish which of the values is alive at a particular moment:

```
a ← amb ❷
â ← ⊤_A ❸
ta ← T ❶

b ← 10

c ← a + b
ĉ ← (a, â, ta) +̂ (b, ⊥, F)
tc ← ta or tb
```

It is now the responsibility of abstract addition to resolve the mixed computation. And finally, the operation calls addition in a single domain $\mathcal{A}$:

```
fn +̂(x, x̂, tx, y, ŷ, ty) ❹
    if !tx and !ty
        ret ⊥
    if !tx
        x̂ ← α(x)
    if !ty
        ŷ ← α(y)
    ret x̂ +̂_A ŷ
```

Figure 9.6: Domain interaction instrumentation example.

```
a ← amb
b ← amb
c ← a + b
d ← a + c
e ← c + d
```

Figure 9.7: Purely abstract computation.

A concrete program state σ with two variables `a`, `b` is captured by Cartesian product $\sigma \in \mathcal{C} \times \mathcal{C}$. Whereas an abstract state $\widehat{\sigma}$ contains elements from the union of concrete and abstract domain $\mathcal{D} = (\mathcal{C} \cup \mathcal{A})$, where $\widehat{\sigma} \in \mathcal{D} \times \mathcal{D}$.
We will visualize program states in the Cartesian form. For example a state where `a = 1` and `b = [2, 10]` is abstracted in the interval domain:

a	b
1	[2, 10]

The actual state in compilation-based abstraction includes both concrete and abstract values, with the latter being realized in a concrete domain. Moreover, it also encompasses value taints:

a	ta	â		b	tb	b̂	
1	F	⊥	⊥	⊥	T	2	10

Figure 9.8: Program state representations.

9.3 Abstract Control Flow

In practice, we want to be conservative with over-approximation as much as possible not to yield false positive results. For that, we want to instrument value constraints into the program during program transformation. In syntactic abstraction, we instrument constraints to all possibly abstract branch destinations (that is, if branch condition was marked by dataflow analysis as possibly abstract). Since LLVM IR models all conditional control flow by branch instructions, we do not need to distinguish between loops and if statements. We cover all conditional control flow using simple branch instrumentation.

In LLVM IR, we can not branch on a non-boolean value. Therefore, it is not possible to condition the branch using an abstract value. In practice, we need to convert the abstract condition to a concrete domain, but this can be suboptimal because it may involve a complex enumeration process, e.g., condition on unbounded interval $[-\infty, \infty]$. To address this issue, we introduce an intermediate step in which we translate the condition to a tristate domain that implements three-value logic, which allows for more straightforward concretization—in the worst case, concretizing a *maybe* value (`M`) results only in two concrete values. For instance $\chi_{\mathcal{A}_i \to \mathtt{Tr}}([-\infty, \infty]) = \mathtt{M}$, which is finite to concretize.

In the syntactic abstraction, we automatically perform instrumentation of all abstract conditions as depicted in Figure 9.9. For each such condition, we instrument conversion to `Tr` domain (realized as call to domain-specific `to_tristate` function), and subsequently, we concretize it ❶ to perform the conditional branch. In contrast to other lifters, the tristate lifter ❷ merges the concrete and abstract dataflow. We lift the value to tristate in both cases. The concrete conversion to tristate ❸ is straightforward since concrete values easily map `Tr` domain

$$\chi_{\mathcal{C} \to \mathtt{Tr}}(v) \triangleq \begin{cases} \mathtt{F}, & \text{if } v = 0 \\ \mathtt{T}, & \text{otherwise} \end{cases}$$

Moreover, each domain implements its optimal translation to the `Tr` domain ❹.

Subsequently, in the abstract control flow, we instrument constraints as assumptions ❺ so that each domain can then implement its own constraint mechanism.

Several different approaches can be used to constrain abstract values to improve an analysis's precision. In this thesis, we considered two main approaches: path condition-based constraining, in which assumptions accumulate constraints in the program path condition and are resolved using an external solver, like in the symbolic execution, and the refinement of non-relational domains, such as intervals using backward constraint propagation.

The mechanism that implements the execution branching is hidden in concretization of tristate values. Whenever we encounter a *maybe* value, we lower it as concrete `amb(0, 1)`, splitting the execution into two. This effectively exposes a minimalistic interface for the execution runtime,

which only needs to implement concrete a `amb` operation. In the implementation, we explored two possibilities, a model-checker-provided state-space branching operation and process forking in the native execution.

The translation to tristate and subsequent concretization represents an interesting extension point for guided analyses. For instance, by assigning the probabilities to particular branches, one can implement guided path exploration [Rua+21] or heuristic-based exploration [CDE08]. In tristate concretization, in the case of an ambiguous branch, we can prioritize a path with a higher weight. This extension point allows us to implement many successful approaches like subpath-guided search [Li+13], KLEE's coverage optimization [CDE08], or AEG's loop exhaustion [Avg+14].

9.4 Shadow Memory

The last ingredient of syntactic abstraction is shadow memory. It has two primary responsibilities: 1) to keep track of abstract content in the concrete memory, and 2) to maintain call frames. This involves both passing abstract parameters when making function calls and preserving abstract return values. Moreover, the abstraction needs to deallocate dead abstract values on exit from scopes.

To perform multi-function analysis, we need to be able to pass abstract arguments to functions and similarly return abstract values from functions. Since we cannot pass values to and from functions as unions, we need to extend the function call mechanism similarly to the value computation. That is, each argument will be accompanied by a dual abstract value and a taint value. However, we want to keep the function prototype the same since it might be called from a non-abstract context, and such a change would break its compatibility.

9.4.1 Abstract Stack

To instrument an abstract call site, we create an alternative abstract *stack* to pass abstract values and taints to and from functions. The stack provides two functions: 1) `stash` to store value and its taint onto the stack, and 2) `unstash` to obtain the pair from the stack. Since we cannot determine beforehand how many parameters will be abstract, we always instrument stash and unstash for all possibly-abstract parameters. In the case of a concrete value, the stack will contain a $\bot$ value and a false taint representing the absence of the abstract parameter (see Figure 9.10).

In the case of indirect calls or virtual functions, the approach leverages the fact that the call site and function entry/exit are independent. So we can instrument abstract stack manipulation around the call site without knowing the precise function called. Due to dataflow analysis, we know which function entries to instrument so that stash and unstash match up during runtime.

In this way, we can also abstract multiple compilation units independently if we share dataflow analysis results or abstract all boundaries between

```
c ← x ◇ y
if c
  /* ... */
else
  /* ... */
```

```
c ← x ◇ y
ĉ ← (x, x̂, tx) ◇̂ (y, ŷ, ty)
tc ← T

t ← to_tristate(c, ĉ, tc)
if γ_Tr(t) ❶
  /* ... */
else
  /* ... */
```

```
fn to_tristate(a, â, ta) ❷
  if !ta
    ret to_tristate_C(a) ❸
  ret to_tristate_A(â) ❹
```

```
c ← x ◇ y
ĉ ← (x, x̂, tx) ◇̂ (y, ŷ, ty)
tc ← T

t ← to_tristate(c, ĉ, tc)
if γ_Tr(t)
  assume x̂ ◇̂ ŷ ❺
  /* ... */
else
  assume !(x̂ ◇̂ ŷ) ❺
  /* ... */
```

Figure 9.9: Control flow abstraction.

[Rua+21]: Ruaro et al. (2021), "SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning"

[Li+13]: Li et al. (2013), "Steering symbolic execution to less traveled paths"

[Avg+14]: Avgerinos et al. (2014), "Automatic exploit generation"

two units. Because, again, stash and unstash operations will resolve the passing of arguments between the boundaries.

Consider the following program that calls the function `first`, which always returns the first parameter.

```
fn first(x, y, z)
    ret x

x ← amb
z ← amb
a ← call first(x, 2, z)
b ← call first(1, 2, z)
```

For each possibly abstract argument, we synthesize `stash` before the call site and `unstash` at the function's entry. We unstash values in a reversed order to stashed values because we store all values in an abstract stack. Note that for parameter `y`, we do not emit an abstract stack place, because it is marked as concrete by dataflow analysis.

```
fn first(x, y, z)
    ẑ, tz ← unstash
    x̂, tx ← unstash

    stash x̂, tx
    ret x

// x triplet:
x ← amb
x̂ ← ⊤
tx ← T
// z triplet:
z ← amb
ẑ ← ⊤
tz ← T
// the first call:
stash x̂, tx
stash ẑ, tz
a ← call first(x, 2, z)
â, ta ← unstash
// the second call:
stash ⊥, F
stash ẑ, tz
b ← call first(1, 2, z)
b̂, tb ← unstash
```

Stashing seamlessly also cooperate with mixed computation. As in the second call, we forward the untainted bottom value for the first concrete argument.

Figure 9.10: An abstract stack.

> An alternative approach would be to create a copy of function for each possible combination of its concrete and abstract arguments. Imagine a function $\mathtt{f} : \overline{v} \times \hat{v} \times v \longrightarrow \overline{v}$, where types denote annotation from the dataflow analysis, i.e., the first argument might be abstract, the second is abstract, and the third is concrete. It is ambiguous what kind of value the function returns. To minimize that amount of abstract computation, we would need to cover two possible scenarios, one when the first argument is concrete: $\mathtt{f} : v \times \hat{v} \times v \longrightarrow \overline{v}$ and the second when it is abstract $\mathtt{f} : \hat{v} \times \hat{v} \times v \longrightarrow \overline{v}$. However, we might not be able to determine the precise kind of return value since it can be dependent on the internal function logic — we may need to recompute reaching abstraction analysis with specialized parameters. Finally, the call operation would need to dispatch to precise specialization based on the runtime taint values. However, compared to the previous approach, we can omit some stashing and abstraction of some computations in the first case. Due to the fact that it requires more complex instrumentation and analysis in the case of indirect calls and complex interprocedural dependencies, we opted for a simpler, more overapproximative approach that does not create function copies.

We haven't touched on the implementation of abstract values yet. For each abstract value, instrumentation allocates type-erased storage. The storage maintains the domain data, an identifier of the domain, and a reference counter. We keep reference count to detect when to deallocate abstract values quickly. We need to perform reference counting since, in the case of relational analysis, it can be referenced by other values after the death of its concrete twin, so we cannot easily determine when to deallocate the value. From the point of instrumentation, abstract values are just pointers. Therefore, we can efficiently pass them along shadow stack and memory.

To update the reference count, we instrument each entry and exit of all scopes that contain abstract computation. For each scope entry, we create a new abstract scope storage. Whenever an abstract value is created, it is registered in the current scope storage. Scope storage also form a stack: on each scope exit, we pop the last scope and decrease the reference count for each value created in the scope. So if a value is not stored in the memory, i.e., referenced from outside the scope, it can be deallocated (see Figure 9.11).

9.4.2 Abstract Dynamic Memory

Regarding memory, we have two substantial tasks to solve: how to deal with abstract values in concrete memory and how to interact with abstract memory. This includes allocations of abstract size, pointers to abstract locations, or abstract offsets to arrays.

Similarly to register abstraction, we want to maintain for each object in memory a triplet, the concrete value, the abstract value, and a flag whether the abstract value is present. Since we can access the memory

at various offsets or overwrite only its parts, we need to represent the abstraction information at a finer granularity. The natural way is to split the memory into bytes and represent the abstraction of each byte separately. In this way, one can, for example, write 8 bytes into memory and read from the same place only 4 bytes (corresponds to storing a `long` integer and loading `int` from the same place). Moreover, we can write a concrete byte in the middle of an array of abstract bytes without breaking the abstraction of the rest of the array.

However, we cannot easily squeeze the taint and abstract value between bytes without breaking the memory layout significantly. In fact, we want to keep the concrete memory layout the same since, usually, program execution depends on the precise location of values, their padding, or relative distance. And trying to preserve these behaviors in the instrumentation might be too complex or even impossible.

For this reason, we employ *shadow memory* to keep abstract values. That is a copy of concrete address space, where each location corresponds to the respective abstract value. Theoretically, each concrete addressable byte has a corresponding place in the shadow memory. However, we can keep the shadow memory sparse for space efficiency, holding only actually abstracted bytes.

To manipulate abstract memory, we design two interface layers. First, a high-level memory interface takes care of storing and loading entire objects from memory using the second low-level interface. The low-level interface deals with shadow memory management at a byte level. It is then the responsibility of the high-level interface to reconstruct objects from a possible mix of concrete and abstract bytes. Recall the memory operations — the `store` and `load` operations correspond to the object-level (or type-level) interface since we specify the type to be loaded, or stored to the memory.

Shadow memory is essentially a map $\texttt{shadow} : \texttt{addr} \longrightarrow \texttt{meta}$ from concrete addresses to metadata (abstract value) addresses. We implement two operations on the byte-level interface: `poke` to assign metadata to the shadow memory and `peek` to retrieve the stored metadata. The `poke` operation takes a pointer to an abstract value and assigns it to the shadow of memory range `rng`[2]:

$$\texttt{poke} : \texttt{meta} \times \texttt{rng} \times \texttt{shadow} \longrightarrow \texttt{shadow}$$

The `peek` operation works in a similar way by taking a range of addresses and returning a sequence of all the metadata that is stored in its corresponding shadow memory. The metadata entries that are returned also contain information about the subrange of memory they correspond to.

$$\texttt{peek} : \texttt{rng} \times \texttt{shadow} \longrightarrow [\texttt{rng} \times \texttt{meta}]$$

This design allows for the possible optimization of interval-based shadow memory. Moreover, we can store larger abstract objects without the need to split them into bytes and postpone the bytewise reasoning for cases where we actually access smaller pieces of memory. We optimize for a more common pattern in programs to read the exact object that was stored.

Imagine a symbolic computation with terms:

```
fn add()
    x ← amb
    y ← amb
    z ← x + y
    ret z
```

If we compute with terms, abstract values for `x` and `y` need to outlive the scope of the function `add` because they are referenced from the term `z`. The term `z` keeps its reference count above zero because it is stashed before the exit from the function.

Figure 9.11: How to outlive own scope?

2: The address range `rng` is given by an address and size.

At the object level, we need to resolve cases when we read from the middle of an abstract object, only a part of the object, or when the read overlaps multiple objects. Again we need to provide two operations. We call them `freeze` and `melt`. They correspond to abstract versions of store and load. That is, `freeze` writes an abstract value $\mathcal{A}$ and its boolean taint $\mathcal{B}$ to the memory range:

$$\texttt{freeze} : (\mathcal{A} \times \mathcal{B}) \times \texttt{rng} \times \texttt{shadow} \longrightarrow \texttt{shadow}$$

And, the `melt` operation obtains an abstract value and its taint from the shadow memory range. In contrast to bytewise `peek`, the `melt` operation returns a single value. For that, it needs to internally reconstruct a single value utilizing abstract operations like `trunc` and `concat`:

$$\texttt{melt} : \texttt{rng} \times \texttt{shadow} \longrightarrow (\mathcal{A} \times \mathcal{B})$$

Freeze to the shadow memory $\mathcal{M}$ either pokes an abstract value to the given range r or erases the content of the memory because the value is concrete (not tainted).

The melt operation (algorithm 3), on the other hand is more complex, it needs to recreate a single abstract value and its taint representing the given region. For that, we need to peek at all chunks in the shadow memory range and concatenate them in the abstract domain. If a chunk contains a concrete value, we load it from the concrete memory and concatenate its abstraction to the result. Over the process, we accumulate whether some chunk was tainted.

Consider the following program:

```
p ← malloc sizeof(i32)
x ← amb
store x → p
y ← load p of i32
```

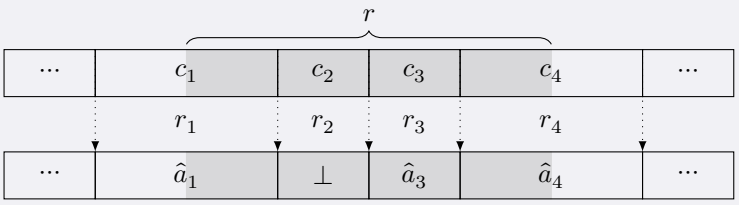
Similarly to the abstract stack, shadow memory is a global entity, so we do not need to pass it as an operation argument:

```
p ← malloc sizeof(i32)
x ← amb
x̂ ← ⊤
tx ← T

store x → p
freeze (x̂, tx) as i32 → p

y ← load p of i32
ŷ, ty ← melt i32 from p
```

Note that we infer the range to `freeze` or `melt` from the provided type.

Figure 9.12: Abstract memory operations.

Example 9.4.1 Consider a load of 4 bytes (an integer) from the memory that spans over four objects – the suffix byte of c_1, whole single-byte objects c_2 and c_3, and lastly, loading the first byte of c_4. The melt needs to reconstruct single 4-byte abstract value:



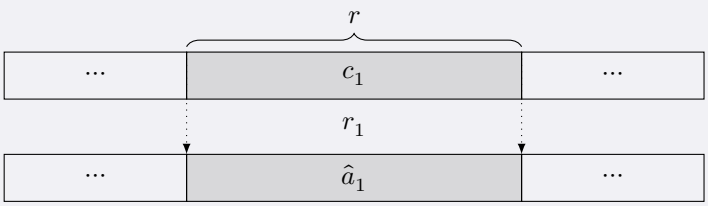
In melt, we gather values from shadow memory $\hat{a}_i$ if present, or abstract concrete bytes:

$$\begin{aligned} v_1 &= \texttt{extract}\ r \cap r_1\ \texttt{from}\ \hat{a}_1 \\ v_2 &= \alpha(c_2) \\ v_3 &= \hat{a}_3 \\ v_4 &= \texttt{trunc}\ \hat{a}_4\ \texttt{to}\ r \cap r_4 \end{aligned}$$

The resulting melted value v is reconstructed as the concatenation of 4 abstracted bytes $v = \texttt{concat}(v_1, v_2, v_3, v_4)$.

If the memory were not fragmented, and all 4 bytes were abstracted together as a single value $\hat{a}_1$, we would not need to reconstruct the

value. Instead, we could simply return the abstraction $\hat{a}_1$. This is the most common case, so we want to be able to perform it with the least effort.



Algorithm 3: `melt` memory operation

Input : memory range r, shadow memory $\mathcal{M}$
Output: abstract pair $(\hat{a}, t_a)$

1 $res_v \leftarrow \bot$ // value to return
2 $res_t \leftarrow$ F // taint to return
/* iterate over chunks in the shadow memory range */
3 **foreach** $r_a, \hat{a}, t_a \leftarrow$ peek$(r, \mathcal{M})$ **do**
4 $res_t \leftarrow res_t \vee t_a$
5 **if** t_a **then**
6 **if** $r_a = r$ **then**
7 $res_v \leftarrow \hat{a}$ // precise load
8 **else if** $r_a < r$ **then**
9 $v \leftarrow$ extract r from $\hat{a}$
10 $res_v \leftarrow$ concat(res_v, v)
11 **else if** $r_a > r$ **then**
12 $v \leftarrow$ trunc $\hat{a}$ to r
13 $res_v \leftarrow$ concat(res_v, v)
14 **else**
15 $res_v \leftarrow$ concat$(res_v, \hat{a})$
16 **end**
17 **else**
/* load a value from the memory range */
18 $v \leftarrow$ load $r_a \cap r$
19 $res_v \leftarrow$ concat$(res_v, \alpha(v))$
20 **end**
21 **end**
22 **return** (res_v, res_t)

We optimize taint propagation by representing a missing taint as a *null* pointer abstract value. So we pass only abstract values to operations. Moreover, in case the value is concrete, we don't need to store $\bot$ abstract value in the shadow memory — we simply do not store anything. Likewise, we return $\bot$ if shadows memory does not have any value assigned to a particular address.

Table 9.1: Shadow memory runtime interface. Abstract value from domain $\mathcal{A}$ is always paired with taint in the boolean domain $\mathcal{B}$. In the low-level interface, all shadow information is kept in the `meta` shadow metadata structure. We identify two main abstract memory structures: a `shadow` which is generic heap-allocated shadow memory, and an abstract `stack`. When referring to shadow memory, we always refer to a defined range (`rng`) of addresses.

`stash` : $(\mathcal{A} \times \mathcal{B}) \times$ `stack` `unstash` : `stack` $\longrightarrow (\mathcal{A} \times \mathcal{B})$	pushes a value and its taint to the abstract stack pops from the abstract stack
`poke` : `meta` $\times$ `rng` $\times$ `shadow` $\longrightarrow$ `shadow` `peek` : `rng` $\times$ `shadow` $\longrightarrow$ [`rng` $\times$ `meta`]	assigns metadata to the shadow memory range returns metadata for the give memory range
`test_taint` : `rng` $\times$ `shadow` $\longrightarrow \mathcal{B}$	returns whether the memory range is tainted
`freeze` : $(\mathcal{A} \times \mathcal{B}) \times$ `rng` $\times$ `shadow` $\longrightarrow$ `shadow` `melt` : `rng` $\times$ `shadow` $\longrightarrow (\mathcal{A} \times \mathcal{B})$	stores an abstract value to memory range obtains an abstract value from the memory range
`enter_scope` `exit_scope`	creates a new scope removes the last scope and collects old values

9.4.3 Aggregate Abstraction

In comparison to scalar abstraction, the syntactic abstraction of aggregates does not operate directly with aggregate types. In LLVM IR, aggregate values are usually represented by a pointer to the underlying aggregate type. Therefore, all the accesses and updates are made through pointers to aggregates. For instance, strings are represented as a pointer to a character array. We need to take this fact into account when we perform syntactic abstraction. During the analysis, we treat pointers to aggregates as the base types for abstraction. This means that when we are conducting string abstraction, a value of type `char*` represents the value that we are abstracting, along with all the memory content that it points to.

Aggregate domains pose unique challenges as they impact dataflow and interfere with concrete domains differently from scalar domains. The first distinction is that we abstract a different set of operations that include primitive memory operations such as `load` and `store`, as well as memory allocation/deallocation and domain-specific operations. Additionally, while scalar abstraction involves the use of shadow memory to store scalar values, we must now ensure that abstract aggregate storage is compatible with this approach.

If a memory block is abstracted, all accesses to it must be made through domain operations to maintain the domain's invariants. Therefore, we do not rely on shadow memory for abstract aggregates.

In contrast to scalar operations, where all operands are required to belong to the same domain, we need to differentiate between domains for offsets and memory content in aggregate operations. To achieve this, lifters will raise values to domains indicated by domain operations specification. For that we require that each aggregate domain defines its offset (index) domain $\widehat{\mathsf{Off}}$ and its value (content) domain $\widehat{\mathsf{Val}}$.

The sources of abstract aggregates for dataflow analysis are the allocation operations or domain-specific constructors. When the content domain is abstract, the allocation operation generates an abstract pointer to abstract content: $\hat{\mathsf{P}}(\hat{v}) \in \mathsf{RA}$ value in the reaching abstraction analysis. Note that the content domain can also be a concrete domain, in which case, an abstract pointer to concrete content is created, i.e., $\hat{\mathsf{P}}(v) \in \mathsf{RA}$. These values are then propagated through the dataflow algorithm outlined in Section 3.7. Ultimately, we classify operations by the dataflow annotations.

When we encounter a store operation in which either an abstract value or an abstract aggregate is being stored, we must instrument an abstract operation into the program. If the operation involves storing an abstract value, we create a freeze into shadow memory, as described earlier. On the other hand, if the operation involves storing into an abstract aggregate, we need to instrument an abstract store given by aggregate domain into the program. Note that both the pointer and value might be maybe-abstract, and it might not be feasible to distinguish this statically. In such cases, we must dynamically dispatch the correct operation during runtime, for instance as in case depicted in Figure 9.13.

In the case of potentially abstract stores, we synthesize the following store lifter. Similar to scalar abstraction, we propagate taints with pointer values. Specifically, when we allocate a pointer to an abstract aggregate, we propagate a taint to indicate that the pointer represents an abstract aggregate. We use taints to dispatch to the correct abstract operation:

```
fn store_lifter(v, v̂, tv, p, p̂, tp):
  if tp:
    ⟨â, ô⟩ ← p̂
    if not tv:
      v̂ ← α_Val̂(v)
    call update_𝒜 v̂, â, ô // aggregate store
  else if tv:
    freeze v̂, tv → p     // store to shadow memory
  else:
    store v → p          // concrete store
```

where arguments consist of two triplets – concrete, abstract, and taint values – for both the stored value and the pointer to the destination. We recognize three domains: the scalar value domain $\hat{v} \in \widehat{\text{Val}}$; the scalar offset domain $\hat{o} \in \widehat{\text{Off}}$; and the abstract aggregate domain $\hat{a} \in \mathcal{A}$. Note that abstract pointers are formed by two abstract values. We can store this information as a pointer to a pair of these two values or as metadata in the interpreter. In the example, we unpack the pair $\langle \hat{a}, \hat{o} \rangle \leftarrow \hat{p}$ into two constituent values.

We also instrument memory accesses in a similar manner. If the pointer is abstract, we perform the aggregate access. Otherwise, we melt the abstract value from shadow memory or perform only the concrete load operation.

The domain-specific operations are similar in nature. We identify a set of functions that the domain implements, such as standard library string functions.[3] However, we often cannot determine whether the call is purely abstract or concrete. Therefore, we create a lifter function that decides what operation to execute and lifts parameters to the desired domain during runtime. We can specify whether the parameters are purely concrete, abstract, or abstract offsets or respective pointers to abstract aggregates. Using these annotations, we synthesize the lifter method that checks if some of the desired abstract parameters are tainted. If they are not, we call the fast concrete path. Otherwise, we lift not-yet abstract values to the desired abstract domains and perform the domain-specific abstract operation (c.f. Example 9.4.2).

```
fn value()
  x ← amb
  if x
    ret x
  ret 0

fn mem()
  s ← amb
  if s
    v ← malloc s
  else
    v ← malloc sizeof(i32)
  ret v

fn main()
  v ← call value()
  m ← call mem()
  store v → m
```

Take the program depicted. It is not feasible to determine statically whether the store operation is concrete or abstract.

Figure 9.13: Example of statically ambiguous store operation.

3: This is performed purely syntactically based on functions defined in the domain library.

At present, the process of instrumentation depends on the domain to define its own domain-specific operations. However, it might be worth exploring techniques for the automatic synthesis of abstract transformers that enable us to determine the most suitable approximations of specific compound operations. For example, adapting the approach described in [Kal+22] to compilation-based techniques could enable the compiler to determine the most appropriate abstraction for the given program.

[Kal+22]: Kalita et al. (2022), "Synthesizing Abstract Transformers"

Example 9.4.2 For example, let's consider the dispatch function for the `strchr` operation. The string abstraction $\mathcal{A}$ defines its own $\texttt{strchr}_{\mathcal{A}}$ function and specifies that the character should be in the value domain:

```
fn strchr_lifter(p, p̂, tp, v, v̂, tv):
    if tp:
        if not tv:
            v̂ ← α_Val̂(v)
        r̂ ← call strchr_A(p̂, v̂)
        stash r̂, T
        ret ⊥
    else
        stash v̂, tv
        r ← call ‾strchr‾(p, v)
        ret r
```

We utilize the abstract stack and stash operations to pass abstract parameters in instrumented original function $\overline{\texttt{strchr}}$. There is no need to unstash after calling $\overline{\texttt{strchr}}$ since we would immediately stash the same value back. The caller of the dispatch operation will then call unstash to retrieve the abstract result.

Example 9.4.3 The decision of whether to perform abstract `strchr` when only `v` is abstract is left to the implementation. If we choose to call `strchr` only when the aggregate is originally abstract, then we need to modify the original `strchr` function as the parameter `v` may still be abstract. Alternatively, we can use the instrumented function $\overline{\texttt{strchr}}$ if only the value is abstract or use the purely concrete function if none of the parameters are tainted:

```
fn strchr_lifter(p, p̂, tp, v, v̂, tv):
    if tp:
        if not tv:
            v̂ ← α_Val̂(v)
        r̂ ← call strchr_A(p̂, v̂)
        stash r̂, T
        ret ⊥
    else
        if tv:
            stash v̂, tv
            r ← call ‾strchr‾(p, v)
        else:
            r ← call strchr(p, v)
            stash ⊥, F
        ret r
```

9.5 LLVM Abstraction & Refinement Tool

The LART tool, or LLVM Abstraction & Refinement Tool, is presented in this thesis as a way to provide LLVM-to-LLVM transformations that implements compilation-based program abstractions. The abstraction is instrumented in terms of concrete LLVM instructions, which results in the program being a normal (concrete) LLVM bitcode that can be executed or analyzed. Extra information about the abstraction(s) in effect over (a fragment of) a program is inserted using special LLVM intrinsic functions and LLVM metadata nodes.

The tool provides both a standalone mode that processes on-disk bitcode files as well as a framework that can be integrated into complex LLVM-based tools. The main purpose of LART is to serve as a "*preprocessor*" for LLVM-based model checkers and other analysis tools, making their job easier by reducing the problem size without compromising the soundness of the analyses.

Moreover, LART allows for the refinement of its implemented abstractions by providing specific instructions or constraints on identifying which part of the abstraction is too rough. An abstraction that is too rough can lead to the creation of false alarms that are visible during subsequent analysis but not present in the original program.

Abstractions for LLVM Bitcode

The goal of LART is to abstract information from LLVM bitcode in order to make subsequent analyses more efficient, at the cost of some precision. To achieve this, LART needs to be able to encode nondeterministic choices in LLVM programs. This can be done through a special-purpose function, called `lart_choose`. This function implements the concrete semantics of the `amb` operator by taking a pair of bounds as arguments and nondeterministically returning a value that falls between those bounds.

In order for downstream tools to recognize this extension to LLVM semantics, it is necessary for them to support the `lart_choose` function. This is the main deviation from the standard LLVM bitcode. Many analysis tools already have similar mechanisms in place, either internally or through an external interface. Adapting tools without support for `lart_choose` to work with LART is usually straightforward.[4]

4: For instance, all tools in SV-COMP support `__VERIFIER_nondet` operations that can be used to implement nondeterministic choice.

In addition to the `lart_choose` function, there are other special-purpose functions provided by LART required for taint and shadow manipulation. To access shadow memory, the program runtime needs to implement the `peek` and `poke` functions. The `test_taint` and `make_taint` functions are used to determine whether a value is tainted.

In order to perform native abstraction execution, LART also provides a concrete implementation of a native runtime. This can be partially linked to the program if a tool does not support a particular feature, such as shadow memory. Similarly, taint propagation instrumentation can be enabled or disabled as needed.

On Domain Implementation

Bringing a domain into practice can be a challenging task, especially in the context of compilation-based abstraction. The theoretical description of domains must be adapted carefully to avoid interfering with program optimizations. The implementation of operations in domains often generates a large number of abstract instructions, potentially hundreds instead of just one, as seen in matrix multiplication in the octagon domain. To maximize efficiency, it is essential to minimize the number of allocations and interactions with the global state of the analyzed program, as these complexities can affect the performance of the underlying analysis. This section will provide a more in-depth technical overview of domain implementation.

As previously discussed in the previous chapters, the implementation consists of two components:

1. LAVA– A Library of Abstract VAlues is a header-only C++ library that provides an implementation of abstract (value) domains.
2. LAMP– A Library of Abstract Metadomains Packages is responsible for domain interaction and interfacing with the analyses.

[GN21]: Gurfinkel et al. (2021), "Abstract interpretation of LLVM with a region-based memory model"

The design of LAVA domains is similar to that of Crab domains [GN21]. Each domain must implement a minimal set of operations for creating new abstract values, constraining them, and lowering them to the tristate domain. However, most operations are not mandatory, and if a program tries to execute one of the unsupported operations, a base implementation will yield an error. The set of operations is based on the LLVM IR bitcode, with a few additional mandatory operations to deal with conversions and control flow interactions. Moreover, LART allows for whole function instrumentation, and a domain may provide full function implementation. Otherwise, domains also can provide effective transformers to other abstract domains. These are used in LAMP to resolve domain interactions. For example, one can describe how to lift the interval domain to symbolic term analysis.

A key aspect of our design is the ability to seamlessly use different domains (link to abstracted program) without requiring a recompilation of the abstract program. To achieve this, we cannot embed domain types during the instrumentation process. Instead, we utilize type-erased abstract objects in C respectively LLVM IR, which are pure pointers to a memory fragment. To identify a domain in that fragment, the first byte of the memory fragment contains an identifier tag and tells the runtime how to interpret the memory section. This mechanism is abstracted from the domain implementation and handled by the LAMP package, ensuring that the information about the used domain does not leak into the instrumented program.

The LAMP domain package manages all employed domains and defines an ordering on these domains so that when values from multiple domains are combined in operation, their common ancestor can be easily identified. This ordering is mainly inferred from transformers defined between domains, but users also have the option to define their own custom or-

ders. Moreover, the LAMP package also specifies which domain should be applied to which values, such as scalar, pointer, or array abstraction.

This section gives a brief overview of implemented and proposed abstractions and their applications.

Variable Elimination (Unit Domain). Eliminating variables from a program can be a simple yet effective abstraction technique. This can be done by mapping variables to a one-element abstract domain.

Concrete Domain. In contrast to a unit domain, the concrete domain represents values precisely and computes with sets of values. It is handy for transitioning between more abstract domains and concrete computation.

Sign Domain. Primarily suites as an example domain to demonstrate principles in this thesis, but it can be used to refine *unit* abstraction.

Interval Domain. A more refined abstract domain chops up the concrete domain into a set of disjoint intervals. Again, particularly large intervals may need to be treated specially when used as a dimension.

Modulus Domain. One approach to abstraction is to map variables with a limited range of values to a particular modulus to the same abstract value. This can lead to significant performance gains in some instances, particularly in benchmarks that involve non-deterministic choices with a small range of possible values.

Term domain. It is clearly advantageous to represent values symbolically using formulas. The term domain suits this purpose in the compilation-based abstraction. It lets the program build an SMT representation of program values at let the runtime interpret it as needed. This allows us to recreate traditional symbolic execution or analysis of uninterpreted programs.

Array domains. Particular objects in programs benefit from special care from abstraction. For instance, a dedicated abstraction of arrays can allow for symbolically large arrays or abstract indexing. That would not be possible if we performed a simple per-value abstraction. In contrast to value domains, object domains are usually parametric, so we can describe by what abstraction to represent the content or indices to arrays.

String domains. In contrast to ordinary array domains, strings represent a specific object category. For strings, it is beneficial to leverage the knowledge of the domain, what operations are commonly used and how strings are represented. So, in addition to array access operations, we can abstract standard functions that operate on strings like `strlen` or `strcmp`.

Pointer domains. Memory abstraction requires special care since most naive abstractions can explode the state space due to insufficient overapproximation. Similarly to objects, we can parametrize how we abstract pointer arithmetic and memory manipulation in pointer abstraction.

There are also a few special-purpose domains, as we already mentioned, a **tristate domain** suites to abstract boolean manipulations. Moreover, we provide a few handy domain adaptors (parametric domains):

Product domain is a parametrizable domain that represents a value in multiple domains simultaneously. It is implemented as a direct product

of two domains and allows for the specification of behavior on branching and concretization.

Relational domain. We want to compute the abstraction as precisely as possible. For that, we augment domains with relational refinement. The relational domain comprises multiple subdomains that maintain data dependencies and perform backward contraint propagation.

Domains can be used to implement various analysis algorithms. For instance, a *concolic execution* can be implemented using a concrete and term domain product. Similarly, for a bounded analysis, one can create a counting domain that cancels the execution after reaching a certain point. A guided symbolic execution is achieved by providing hints to branches on tristate values. Also, counter-example-guided abstraction can leverage an iterative compilation-based strategy.

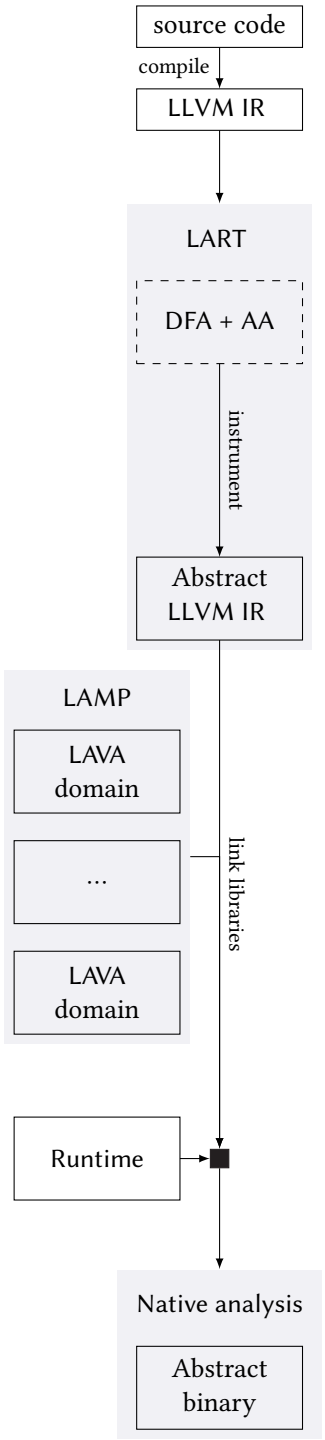


Figure 9.14: To-native LART workflow.

LART Infrastructure

LART is primarily used to generate an LLVM bitcode as close as possible to the regular LLVM semantics. This allows for the use of standard LLVM tools with the transformed bitcode. With the use of a suitable implementation of `lart_choose`, such as random generation or external test vectors, it is possible to produce native code from the abstracted program and execute it natively.

For that LART implements IR-based and IR-to-native abstract analysis using multiple components. This allows for flexibility in usage, as it can be used both as a stand-alone tool for producing self-analyzable binaries and test drivers, as well as in conjunction with other LLVM-based tools at runtime. The components are designed to be decoupled based on their responsibilities to facilitate easy configuration of the desired analysis:

- LART– An LLVM Abstraction & Refinement Tool is instrumentation framework that handles the process of static abstraction, dataflow analysis, constraint propagation, and generates code that is compatible with the desired runtime environment. It can instrument intrinsics for IR-based tools or generate their native implementation to perform binary analysis.
- LAVA– A Library of Abstract VAlues is a header-only `C++` library that provides an implementation of abstract (value) domains. Each domain is implemented as a class (or class template, for parametric domains) and provides a number of methods that implement the individual operations of the domain.
- LAMP– A Library of Abstract Metadomains Packages is responsible for domain interaction. Each `C++` source file (package) implements a single abstract metadomain, which mediates the interaction between abstract domains from LAVA (which represent values at runtime), and the abstract program as generated (transformed) by LART.

  A program abstracted by LART, before it is executed or further analyzed in some way, needs to be linked to exactly one of the LAMP packages (in bitcode form). Interface-wise, all modules are equivalent and LART does not need to know which is going to be

used. The difference lies in which abstract domains they combine and how, i.e., different LAMP modules will give different abstract semantics to the same program (syntactically) abstracted by LART.

- Runtime – The final ingredient of the analysis is a runtime that implements instincts instrument by LART. This can be an interpreter or native library linked to the instrumented bitcode. The LART comes with a simple, configurable runtime library that takes care of shadow memory, tainting, and nondeterministic choices.

9.6 Abstraction Optimization

One of the key motivations for the compilation-based approach in program analysis is the ability to use compiler optimizations on the abstracted code. The advantage of having complete syntactic abstraction prior to program analysis is that optimizations can be performed on a syntactic level. Similar to traditional compilation, it is expected that the compiler will be able to execute optimizations such as constant folding, code deduplication, and inlining on the output of syntactic abstraction.

The optimization of code can always be performed prior to syntactic abstraction. However, we conjecture that syntactic abstraction provides the compiler with an even greater ability to optimize the code. One of the fundamental optimizations executed by the compiler is constant folding, which statically evaluates program expressions using known operands. This results in a program that only operates with ambiguous values or performs too complex computations. By replacing ambiguous inputs with abstract values, they are effectively represented as constants, either interval, sign, or unit value, allowing for additional summarization.

Additionally, through comprehending the compiler optimizations, we anticipate that the compiler will be capable of summarizing taint propagation and eliminating redundant checks in operation lifters. This would involve optimizing the query process, so instead of inquiring about the execution of an abstract or concrete operation individually, a single query would suffice, continuing in a block of either abstract or concrete operations. Therefore, we state the following two hypotheses:

RQ1 Compiler optimizations allow to perform of abstraction summarization on the syntactically abstracted programs.

RQ2 A compiler performs a compile-time taint propagation and inference?

In order to explore these hypotheses, we look more in-depth at the compiler-produced abstract programs and intermediate results of compilation. In this endeavor, to examine the concrete outputs of the pipeline, we use standard C and LLVM IR with any unnecessary elements such as debug information or metadata abbreviated.

A program to be abstracted. It creates an abstract value using lamp function, which effectively implements abstraction of `amb[x, y]`. Recall, the `lamp` library is part of LART. It provides the interface to the implemented domain.

```
int compute() {
    int a = __lamp_any_range_i32(0, 1);
    int b = __lamp_any_range_i32(1, 10);
    int c = a * b;
    int d = a * c;
    int e = c * d;
    return e;
}
```

The optimized LLVM bitcode resembles almost precisely the original source code.

```
define i32 @compute() {
    %1 = call i32 @__lamp_any_range_i32(i32 0, i32 0)
    %2 = call i32 @__lamp_any_range_i32(i32 1, i32 10)
    %3 = mul i32 %2, %1
    %4 = mul i32 %3, %1
    %5 = mul i32 %4, %3
    ret i32 %5
}
```

To begin with, we conduct a syntactic abstraction that computes data dependencies on ambiguous values and instruments abstract operations and propagation of taints. As the instrumented bitcode becomes unwieldy, we will showcase only the noteworthy snippets in the following part. The whole intermediate steps are presented in Chapter E.

First of all, it is important to observe the distinction between the simplified language examples and real LLVM IR bitcode, for instance, the lamp functions generate taint internally and stash it to an abstract stack. Consequently, we need to instrument `unstash` following every `any` call:

```
%4 = call i32 @__lamp_any_range_i32(i32 1, i32 10)
%5 = call i1  @__lart_unstash_taint()
%6 = call i8* @__lart_unstash()
```

For each operation, LART instruments taint propagation (`%7`), it preserves the original instruction and creates an abstract `@lart.test.taint` call, which decides whether call `@lart.lifter`. In the lifter, we elevate the concrete values to the abstract domain and invoke the actual `__lamp_mul` operation. To achieve this, `@lart.test.taint` takes triplets of taint, concrete, and abstract value. The subsequent code segment is essentially reiterated thrice (refer to the complete bitcode in Section E).

```
%7 = or i1 %2, %5 // taints
%c = mul i32 %a, %b
%9 = call i8* @lart.test.taint.mul(
  @lart.lifter.mul, i1 %2, i32 %a, i8* %6, i1 %5, i32 %b, i8* %3
)
```

Subsequently, we can optimize the code in the usual LLVM IR manner. This entails inlining calls to `@lart.test.taint` and `@lart.lifter`, with code deduplication and constraint propagation taking place. It is worth noting that if the compiler determines that none of the values is tainted, it skips all the lifting and abstract computation and solely carries out concrete multiplication:

```
%7 = or i1 %2, %5 // taints
%c = mul i32 %a, %b
br i1 %7, label %abstract.path, label %concrete.path.exit
abstract.path:
    /* ... */
concrete.path.exit:
    %abs = phi i8* [ %18, %lart.lifter ], [ null, %0 ]
    %d = mul i32 %a, %c
```

```
    %e = mul i32 %c, %d
    tail call void @__lart_stash(i1 %7, i8* %abs)
    ret i32 %e
```

Moreover, the compiler could deduce that if the second multiplication is abstract, the third one must be abstract as well. Consequently, we can entirely eliminate taint checks and value lifting. The complete LLVM IR is available in Section E, here we show only a part of the inlined lifter:

```
lart.lifter.exit:
    %15 = phi i8* [ %13, %lift ], [ %11, %merge ]
    %16 = phi i8* [ %14, %lift ], [ %3,  %merge ]
    %17 = tail call i8* @__lamp_mul(i8* %15, i8* %16)
    %18 = tail call i8* @__lamp_mul(i8* %17, i8* %15)
    br label %concrete.path.exit
```

In order to optimize the abstract code, it is necessary to link the abstract domain and perform link-time optimization. In LLVM, link-time optimization essentially merges two LLVM IR modules and applies standard bitcode optimizations, with particular emphasis on inlining.

Link-time optimization grants the compiler complete visibility into the program's abstract operations. For instance, it is able to recognize that some operations might be side-effect-free, and thus the compiler can combine redundant computations. In the current instance, we linked the program with the unit domain, which essentially slices away the computations because its operations can be constant-folded to a single unit. The only recognizable elements of the resulting code are the allocations of abstract storage and abstract stack manipulations, which cannot be eliminated since they have side effects. Nonetheless, this case might be optimized further if we design a better allocation strategy for abstract domains. Otherwise, the operations are essentially summarized to return `0` and a stash of an abstract unit (see Section E).

```
// creation of an abstract value
%1 = tail call i8* @new(i64 1)
%2 = bitcast i8* %1 to %lamp::storage*
tail call void @__lart_stash(i1 zeroext true, i8* nonnull %2)
// unstash from the conctructor
%4 = tail call i1 @__lart_unstash_taint()
%5 = tail call i8* @__lart_unstash()
```

If we link the program with a more intricate domain, such as the interval domain, the optimizations once again attempt to simplify the code to the point where it is nearly unrecognizable in comparison to the original computation. This is accomplished by the compiler scalarizing the computation, processing each boundary separately, and eventually reconstructing the final interval. Here is a fragment of such computation:

```
// store boundaries as scalars
%l = extractvalue { i64, i64 } %16, 0
%r = extractvalue { i64, i64 } %16, 1
// detect bottom value
%1 = icmp slt i64 %l, %r
```

As can be observed, the compiler provides a substantial advantage over dynamic analysis. Interpreters struggle to optimize code and deduce dependencies during runtime. However, optimizations also have their limitations and cannot completely summarize computations that extend beyond function boundaries and affect a global state and memory.

Without inlining, the compiler has no access to information about function arguments, and as a result, it must preserve all computations with them. One possible solution would be to employ just-in-time compilation, allowing for the utilization of current function parameters to optimize away concrete or abstract paths.

The implementation can be optimized for better transparency. For example, if domain values are small enough, a small value optimization can be made to store them directly in place of a pointer. This way, they are present on the stack, and unnecessary allocations can be avoided.

Similarly, the initial taint values could be inlined, which would assist the compiler in summarizing taint propagation. Currently, the compiler does not have any knowledge of the initial abstract stack value, so it has to perform all `stash` and `unstash` operations. Eliminating pairs of `stash` and `unstash` operations after the function is inlined can also further enhance the transparency of taint propagation.

Essentially, taint propagation is the most crucial aspect, as it allows the complete slicing away of concrete or abstract paths. These are just suggestions and have not been implemented in our current implementation of LART.

Summary

In this chapter, we have presented a technical overview of compilation-based abstraction with a focus on syntactic abstraction. Syntactic abstraction is a key component of the proposed approach as it is part of the compilation step. Its primary purpose is to resolve interaction with a concrete environment, which involves three key components: direct computation, control flow, and memory interaction. The proposed approach was designed to resolve all of these interactions in a domain-agnostic manner. Additionally, the approach places a strong emphasis on minimizing the required abstraction, whether it is achieved statically or dynamically using instrumented taint propagation techniques.

To implement this approach, we have developed LART: LLVM Abstraction and Refinement Tool, also described in this chapter. Lastly, we have demonstrated the advantages of using syntactic abstraction and its ability to work collaboratively with concrete program optimizations.

Applications & Future Work 10

In this chapter, we conclude our discussion of compilation-based abstraction by examining its integration into various approaches, including explicit state model checking with DIVINE and native execution. Additionally, we delve into techniques that capitalize on compilation-based abstraction, such as refinement of pointer relational domain, compositional program analysis, and syntactic refinement for program decompilation. We conclude the chapter with possible future research directions.

10.1 Model Checking & Abstract Execution

The main applications presented in this thesis were developed as extensions to the DIVINE tool. Here, we provide a more detailed description of its architecture, as it offers greater insight into the applicability of compilation-based abstraction in model checking and modular program analysis.

DIVINE is an explicit state model checker designed for the verification of real-world code written in high-level programming languages, particularly in C/C++. Its verification process includes common tasks such as checking for assertion violations, ensuring memory safety, and handling data nondeterminism. Moreover, since it primarily targets full-featured C++ code, the tool also supports exception handling and simulates parallelism under various memory models. It would be impractical to handle all of these aspects in one monolithic verifier, and hence DIVINE is partitioned into multiple dedicated components (see Figure 10.1). In the core, DIVINE uses an LLVM IR interpreter combined with a parallel state-space exploration algorithm. The runtime environment and parallelism are modeled by a lightweight verification-oriented operating system and its scheduler, whereas a set of libraries covers C and C++ runtime.

The verification process of DIVINE is performed in two phases: the *static analysis* where preprocessing happens; and the *dynamic analysis* where the model checking (MC) based on program interpretation (DiVM) is performed.

In the preprocessing phase, the LLVM IR input is analyzed by LLVM abstraction and Refinement Tool (LART). Its main purpose is to instrument LLVM IR for the verification; this includes stubbing of undefined functions, generating metadata for the verifier (e.g., annotating atomic blocks and lifetime markers), instrumenting a verifier-friendly exception handling, and creating interrupt calls for rescheduling. Moreover, it serves as instrumentation engine of compilation-based abstraction for DIVINE to handle data nondeterminism.

The preprocessed bitcode is linked together with DIVINE's operating system (DiOS) and standard C and C++ libraries. DiOS is a lightweight operating system that provides a POSIX environment for verification, including virtual file system (VFS) and system call emulation (SYS). It manages process and thread scheduling, and is responsible for detecting program faults

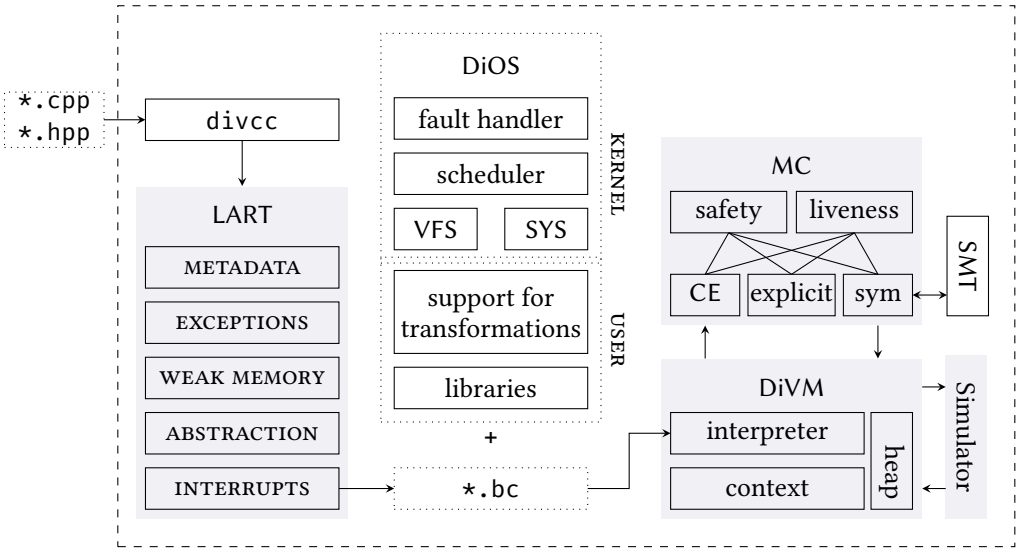


Figure 10.1: DIVINE consists of the five main components `divcc` (verification dedicated compiler), LART (LLVM Abstraction and Refinement Tool), DiOS (operating system), DiVM (virtual machine) and MC (model-checking algorithms).

such as assertion failures, out-of-bound access, memory leaks, arithmetic errors, concurrency issues, and user-defined errors.

In the dynamic phase, DIVINE employs two key components, a virtual machine (DiVM) and a toolset of model checking algorithms (MC). Essentially, DiVM is a state-space generator. It handles LLVM IR interpretation and memory (heap) representation. During verification, DiVM executes the whole DiOS with an input program and maintains a hash table of reached states (memory configurations).

Generated transitions are processed by a selected model-checking algorithm (either explicit or semi-symbolic, which uses an SMT solver to deal with symbolic data). The model checking algorithm performs a concurrent exhaustive search of program state-space and stores only hashed states in the concurrent hash table [Bar+15]. For memory efficiency, the MC engine only stores hashed states and does not keep the path explicitly during the verification, therefore when model checking encounters an error trace, DIVINE needs to recompute the path by the counterexample generation component (CE).

[Bar+15]: Barnat et al. (2015), "Fast, Dynamically-Sized Concurrent Hash Table"

Past experience has repeatedly shown that a successful explicit-state model checker needs to combine a fast interpreter (the component which computes successor states), partial order [Pel98] and/or symmetry reductions [Cla+98] and efficient means to store the visited and open sets [BBR10; Laa14].

To obtain a fast interpreter of LLVM IR in DIVINE, we have reduced its responsibilities to the bare minimum. In this way, we did not just accelerate evaluation, but additionally, by keeping the interpreter simple, we have minimized the potential of errors in divine. This design aligns with the compilation-based abstraction that leverages the fast interpreter of abstraction represented in the means of concrete LLVM IR.

[Pel98]: Peled (1998), "Ten Years of Partial Order Reduction"

[Cla+98]: Clarke et al. (1998), "Symmetry reductions in model checking"

[BBR10]: Barnat et al. (2010), "Scalable shared memory LTL model checking"

[Laa14]: Laarman (2014), "Scalable multi-core model checking"

With the separation of the system environment (DiOS) and abstractions (LART) from the interpreter, we have also achieved potential reusability of components for other verification tools. Since the preprocessed linked bitcode is an ordinary LLVM bitcode with just a few operations that need to be supported by a virtual machine (e.g., memory allocation and nondeterministic choice). In fact, the DiOS was successfully ported with a small amount of glue to a symbolic executor KLEE [Roč+21].

However, such component-wise design is always a trade-off between efficiency and reusability. The verification would be faster with an integrated operating system and symbolic evaluator in the virtual machine, but for the price of complexity and zero reusability. Through a separation of concerns, we follow the current trend that components of tools are reusable for other researchers and open source developers in a similar fashion as SMT solvers [DB08; Det+14] or automata manipulating libraries [Dur+16; Dur+22].

[Roč+21]: Ročkai et al. (2021), "Reproducible Execution of POSIX Programs with DiOS"

[DB08]: De Moura et al. (2008), "Z3: An Efficient SMT Solver"
[Det+14]: Deters et al. (2014), "A tour of CVC4: How it works, and how to use it"

[Dur+16]: Duret-Lutz et al. (2016), "Spot 2.0 — a framework for LTL and ω-automata manipulation"
[Dur+22]: Duret-Lutz et al. (2022), "From Spot 2.0 to Spot 2.10: What's New?"

10.1.1 Symbolic Computation

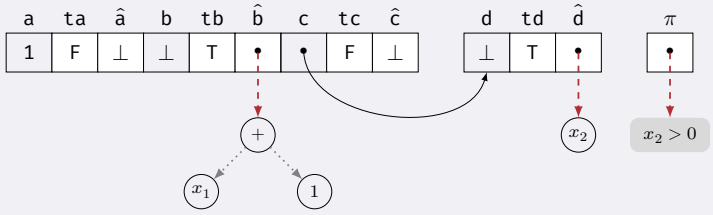
The initial focus of applying compilation-based abstraction was on symbolic computation, specifically the control-explicit data-symbolic approach, achieved through the use of a term domain. To integrate this into verification, we developed a procedure for performing feasibility and equality checks. During execution, the term domain collects path constraints into a global path condition, which is then checked for satisfiability on program branches. We inform DIVINE whenever a new conjunct is appended to the path condition in order to test its satisfiability. To enhance efficiency, we maintain a root to the path condition in a designated location, making it easily accessible for feasibility checks.

The equality check in compilation-based abstraction is more intricate, as we need to exclude the abstract part of the state representation from concrete comparison, as abstract values are rarely comparable purely syntactically. However, in compilation-based abstraction, abstract values are encoded using concrete representation, so we need a mechanism to differentiate between abstract and concrete values in the program states.

In DIVINE, abstract values are always pointers to abstract representation. These pointers are marked with a special identifier to distinguish them from concrete values – imagine a special pool allocator for abstract values. This enables the memory content of abstract values to be excluded from concrete comparison when performing equality checks. Additionally, these markings are used to determine which values to extract as symbolic formulas from the program.

In the context of symbolic execution, we distinguish between three types of memory blocks: concrete blocks, weak block, and marked blocks, and similarly, we refer to the pointers that point to them as concrete, weak, and marked pointers. Concrete memory blocks are considered in the equality comparison of concrete memory, whereas weak and marked blocks are omitted (cf. Example 10.1.1).

Example 10.1.1 Consider semi-symbolic state $\hat{\sigma}$ represented as DIVINE's memory graph:



The presented state consists of a concrete value `a`, a term tree $\hat{\texttt{b}}$, a concrete pointer `c` to the location of `d`, which contains an abstract value $\hat{\texttt{d}}$, and a path condition π. In reality, taints and abstract pointers are stored in a shadow memory (illustrated with white background), but for brevity, we present them next to their concrete-related place.

We distinguish three types of pointers:

- **concrete pointers**,
- **weak pointers** are unobservable (ignored) by equality checks.
- **marked pointers** point to the root of abstract values and are unobservable in the same way as weak pointers.

In the equality check, we compare concrete values in memory directly. However, if any of the compared places contain a marked value, we gather the abstract representation stored behind marked pointers for abstract equality check. In addition, to conduct a feasibility check, we always have a specific location of the path condition π likewise stored as a marked value.

Marked pointers are used to collect all abstract values, which are passed to domain-specific equality comparison operations. These pointers only point to the root of abstract values. However, in cases where abstract values consist of more complex dynamic structures, such as term trees, we need to annotate all their memory blocks to be excluded from the concrete comparison. We do not use marked pointers to annotate internal blocks, as we do not want to consider them separately in abstract equality checks. Instead, we use the third category of weak pointers, indicating that block should not be used in concrete equality comparison.

Notably, this approach is domain-independent, as all metadomains are configured to be allocated as marked storage by default. This allows for abstract model checking with any domain, as long as an appropriate equality procedure is provided. For instance, in the case of the term domain, the equality procedure builds an SMT query and invokes an SMT solver.

We provide an implementation and evaluation of control-explicit data-symbolic model checking in Appendix A, which was adapted from the original publication [LRB18]. Furthermore, we have successfully competed with DIVINE, employing the term domain [Lau+19], multiple times in SV-COMP as detailed in Appendix B.

[LRB18]: Lauko et al. (2018), "Symbolic Computation via Program Transformation"

[Lau+19]: Lauko et al. (2019), "Extending DIVINE with Symbolic Verification Using SMT"

10.1.2 Native Execution

The goal was not only to enhance interpretation-based tools with abstraction capabilities but also to enable the native execution of abstract programs. To achieve this, we implemented instrumentation for taint propagation and runtime management of shadow memory, which were originally handled by the DIVINE virtual machine. Additionally, the runtime allows for the configuration of nondeterministic choice operations, giving us the flexibility to specify a specific path for program execution or fork on ambiguous branch conditions.

Our focus was solely on abstract execution, as performing native equality checks and capturing the state of the system in a reasonable manner for model checking is a challenging task, if not impossible. In contrast to DIVINE, where feasibility checks are invoked from the interpreter, and the interaction with SMT solver does not leak to program interpretation, in the native execution, we augmented the term domain to compute directly in the representation of the Z3 solver and invoke satisfiability queries from the program on branch conditions. If a condition is not satisfied, we gracefully terminate the program run. This approach serves as a solid foundation for the future development of native abstraction, as we have proved by successfully competing with a native configuration of LART in SV-COMP (cf. Appendix B), demonstrating its superiority over the interpretation-based approach of DIVINE.

Native execution opens up interesting applications, such as recording choices made during program execution and later loading the abstract program into a debugger to simulate the abstract execution, allowing developers to explore program behavior and produced counterexamples. Moreover, by simply replacing the abstract domain with a concrete domain that specifies inputs for a specific execution, we can determine the feasibility of counterexamples natively without the need to rerun the static analysis. This approach provides an elegant solution to the general problem faced by verification tools on how to generate executable counterexamples, which are essential for understanding the verification results [Gen+18].

[Gen+18]: Gennari et al. (2018), "Executable Counterexamples in Software Model Checking"

To facilitate a better understanding of abstract execution, we also implemented a tracing functor domain, which logs the computation of each operation during program execution to a log file. It is worth noting that this can be achieved purely by introducing a new "tracing" domain without changing the syntactic abstraction or the actual domain in use. Similarly, we developed a term domain that generates a graphic representation of term trees during its execution, providing further insights into program behavior.

10.1.3 String Abstraction

In Chapter 5, we presented our work on verifying C string manipulating programs, which was discussed in detail in our papers [Roč+19] and [Lau+20]. Our proposed abstraction for string manipulation was the M-String segmentation functor domain, which was parametrized by two domains: an array of bounds (offsets) and an array of characters.

In terms of the LAVA domain, the M-String domain is a C++ library that defines the abstract semantics of string operations and encoding of string representations as concrete data. This domain is defined as a C++ template, and a specific instantiation is automatically derived by the C++ compiler from the template and classes that represent the type parameters for the domain.

The M-String domain implements all the necessary aggregate operations, including lift, update, and access. Additionally, it provides an optimized version of common string operations such as `strlen`, `strcpy`, `strcat`, `strcmp`, and `strchr`. These optimized operations help reduce the loss of abstraction precision that would occur if only the abstraction of accesses and updates from strings were used.

Since C strings are stored as shared, mutable character arrays, the implementation of the M-String domain needs to account for the sharing semantics of such arrays. If multiple pointers exist to the same abstract string, any modifications made through one pointer should be visible when the string is accessed through another pointer. Furthermore, the pointers may not be equal, as they can point to different suffixes of the same string. Therefore, the representation of pointers to abstract strings must separately handle the object and offset components, and the representation of the offset component must be compatible with the bound domain.

In practice, the M-String domain is used as part of a larger metadomain, which also defines the domains for bounds and characters as described in the previous example. The implementation of the M-String domain, along with examples and documentation on domain usage, can be accessed online on the supplementary page at https://divine.fi.muni.cz/2020/mstring. In addition, we provide a detailed description of multiple domain instantiations and their evaluation in Appendix C.

Example 10.1.2 This example illustrates a simple metadomain called `symstring`, implemented in C++ using the LAMP framework.

```
using mstring = lava::mstring< term, term >;

struct symstring {
    using doms = domain_list< term, mstring >;

    using scalar_lift_dom = term;
    using scalar_any_dom  = term;
    using array_lift_dom  = mstring;
    using array_any_dom   = mstring;

    static constexpr int join(int a, int b) noexcept {
        auto mstring_idx = doms::idx< mstring >;
        if ( a == mstring_idx || b == mstring_idx )
            return mstring_idx;
        else
            return doms::idx< term >;
    }
};
using meta_domain = semilattice< symstring >;
```

This metadomain consists of two domains: `term` and `mstring` instantiated with `term` bounds and `term` characters. Within the `symstring` metadomain, we define a list of domains, specify entry domains for scalar and aggregate abstraction (lifting) when converting concrete to abstract values, and define domains for abstract constructors (amb operations) referred to as `any` in the implementation. Furthermore, we establish the order of domains using the `join` function, indicating that the `mstring` domain is ordered above the `term` domain.

10.1.4 Atomicity of Operations

The well-defined atomicity of our concrete semantics and LLVM IR bitcode is advantageous for explicit state model-checking, as each operation represents a step in the transition system. However, this is not the case in the compilation-based abstraction, where operations on more complex domains may involve hundreds of instructions. This poses a problem, as programs during the execution of an abstract operation are typically in an invalid state – the values may be in a partially constructed form, or the path condition may be incomplete.[1] Performing state comparison or subsumption check in these cases may lead to unpredictable state comparisons, possibly causing the interpreter to crash if it fails to load the symbolic formulae from program memory.

1: This problem does not arise in interpretation-based approaches, as the interpreter inherently performs each transformation atomically.

In addition, when performing concurrency analysis, which is typically the goal of software model checkers, it is necessary to faithfully explore all possible thread interleavings. If the operation being abstracted is atomic, we need to make the interpreter to perform the sequence of instructions that implements the abstract operation atomically. This also means that the model checker cannot perform subsumption and feasibility checks within the atomic section.

Tools that analyze concurrent behavior often provide specific operations to mark the beginning and end of atomic sections. For example, tools that compete in the concurrency category of SV-COMP must support `__VERIFIER_atomic_begin` and `__VERIFIER_atomic_end` to model atomic sections. In DIVINE, interrupts can be masked using the `__dios_mask` function, which turns on or off the interrupts of the verification-dedicated operating system. Suppose we want to mark specific points in non-atomic operations, such as those that manipulate non-scalar abstract values like arrays. In that case, we can use the `__dios_reschedule` operation to allow the operating system to switch contexts at that specific point, for example, after each modified field of an abstract array.

We do not need to perform this task manually while implementing the domains since all elementary domains are accessed through the metadomain interface. Therefore, we only need to wrap the interfacing functions into atomic sections. To achieve this, we annotate all scalar metadomain functions to be atomic and utilize a dedicated pass in LART to add specific masks automatically at the start and end of the functions. However, implementing more complex (non-atomic) operations requires us to insert masks manually.

10.2 Syntactic Refinement

Frequently, when employing syntactic abstraction, the level of granularity can be too coarse, resulting in a significant portion of programs being syntactically abstracted. It can also be challenging to statically identify the right abstract domain on the first attempt.

To address this issue, it can be beneficial to improve syntactic abstraction through refinement and a counterexample-guided approach. In counterexample-guided syntactic abstraction, we often underapproximate the part of the program to be instrumented (leaving it in its concrete representation) and let the runtime determine whether we have reached a location that should have been abstracted but was not. Once we have identified such a location, we can obtain a counterexample and use it to refine our syntactic abstraction.

Similarly, it is possible to recognize during runtime that we have utilized an inadequate abstract domain. For example, we may discover that certain loop indices or memory blocks should be represented abstractly, as they are currently being used in abstract string manipulations. Armed with this knowledge, we can adjust the entry domains, designate certain allocations as abstract, or link different metadomains that align better with the identified entries.

10.2.1 Heap Layout Abstraction

Let us revisit the pointer arithmetic domain $\widehat{\mathsf{Pa}}(\widehat{\mathsf{D}}, \widehat{\mathsf{N}})$ discussed in Chapter 6. Its purpose was to allow for an analysis to consider all pertinent heap orderings. To accomplish this, the pointer arithmetic domain separated the duties of relational pointer reasoning and memory manipulations into two domains, which were both maintained simultaneously as a product of the numerical relational domain $\widehat{\mathsf{N}}$ and the memory (dereference) domain $\widehat{\mathsf{D}}$. By using these two representations, we were able to conduct symbolic relational reasoning while maintaining a straightforward memory representation that did not require us to consider its layout ordering.

While we have utilized this analysis in the DIVINE model checker, it is not attainable to abstract all allocations to the $\widehat{\mathsf{Pa}}(\widehat{\mathsf{D}}, \widehat{\mathsf{N}})$ domain. Moreover, since only a few, if any, are utilized in ambiguous operations, we do not need to abstract most of the allocations.

Although the $\widehat{\mathsf{Pa}}(\widehat{\mathsf{D}}, \widehat{\mathsf{N}})$ domain is relatively lightweight, the same cannot be said for the $\widehat{\mathsf{N}}$, as keeping track of numeric values can result in significant overhead. This is particularly true when using $\widehat{\mathsf{N}} = \mathcal{T}$ (the term domain). It is advantageous to store as many pointers as possible in the concrete domain, avoiding the additional cost of $\widehat{\mathsf{Pa}}(\widehat{\mathsf{D}}, \widehat{\mathsf{N}})$ and hence $\widehat{\mathsf{N}}$.

Ambiguity and Soundness

Intuitively, if ambiguous operations are absent, we do not need to use $\widehat{\mathsf{Pa}}(\widehat{\mathsf{D}}, \widehat{\mathsf{N}})$ to keep track of the ambiguity. Let us restate this a little more formally.

Lemma 10.2.1 *If* $\sigma \xrightarrow{r_p \longleftarrow \texttt{malloc s}} \sigma'$*, then all possible* σ' *are equivalent to each other, in the sense of Definition 6.4.1.*

Proof. Follows from the semantics of operations given in Section 3.3 and from Definition 6.4.1. □

Theorem 10.2.2 *Consider a reduced state space that is constructed by selecting a single base address for each* `malloc`*. If the program only executes unambiguous operations, for each state that is reachable in the full state space, an equivalent state is reachable in the reduced state space.*

In other words, the under-approximation of simply selecting a fixed base address for every allocation is not an under-approximation at all, unless the program executes an ambiguous operation.

Proof. Follows by induction from Corollary 6.4.1 (unambiguous operations preserve state equivalence) and Lemma 10.2.1 (successors of `malloc` are equivalent to each other). □

We have observed that until an ambiguous operation is executed, we can get away with a very simple approach to modeling pointers, without compromising soundness. In normal circumstances, this approach is an under-approximation: errors that are reachable in real execution might be missed in analysis. However, by means of a simple trick, we can reverse the situation. If we mark every ambiguous operation on pointers as an *error*, the resulting analysis becomes an over-approximation – it is impossible to miss errors hidden behind ambiguous operations, because the ambiguous operation itself is an error. Of course, this might cause spurious errors to appear (this obviously happens, for instance, if the program executes an ambiguous operation, but is in fact error-free).

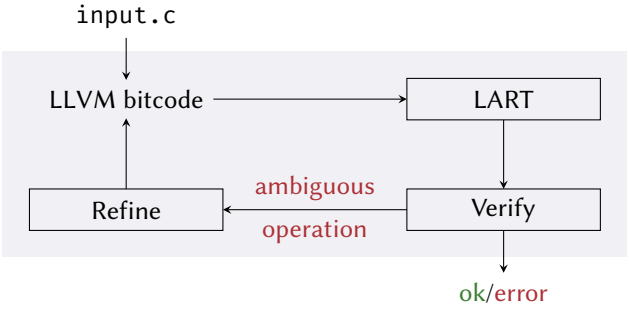
A straightforward and well-established approach to handle false positive errors is through CEGAR [Cla+00] (counterexample-guided abstraction refinement). To apply this method to the current problem, we follow these steps:

1. as outlined above, modify $\mathcal{C}$ so that every ambiguous operation is an error: this requires $\mathcal{C}$ to be able to identify ambiguous operations – fortunately, Theorems 6.4.2 to 6.4.5 give us a guide on how to do that,
2. when the decision procedure finds a spurious error (these are easy to identify, since they are exactly the ambiguous operations), make a note of the offending operation,
3. perform backward dataflow analysis starting from the operands of the offending operation, to identify the `malloc` instructions which created the pointers in question,

[Cla+00]: Clarke et al. (2000), "Counterexample-Guided Abstraction Refinement"

4. replace the `malloc` (which is from the concrete domain $\mathcal{C}$) with its $\widehat{\mathrm{Pa}}(\widehat{\mathrm{D}}, \widehat{\mathrm{N}})$ version and adjust any operations identified by LART's reaching abstraction analysis,
5. restart the analysis.

A high-level overview of the process is given in the following picture and an example of its application to a simple program is shown in Figure 10.2.



Pointer refinement loop incrementally abstracts ambiguous operations. LART takes care of program transformation and propagation of abstraction, while Refine annotates sources of abstract pointers. Annotations are kept for future runs in the form of LLVM metadata.

Implementation

Of course, to make implementation of the refinement loop possible, the verifier (or the concrete pointer domain) must perform the ambiguity check. Errors reported by the verifier must be accompanied by a counterexample trace which can then be used to identify the instruction, which caused the ambiguous operation and hence to compute the refined abstraction. In DIVINE, this has been implemented in the virtual machine, which can efficiently recover a base address from any valid pointer. Since the base address is known, Theorems 6.4.2 to 6.4.5 can be directly used to decide whether an operation is ambiguous or not.

Currently, the pointer arithmetic domain employs a very basic form of counterexample-guided refinement, which involves beginning with a fully concrete program and refining pointers one by one. However, there is potential for future research to improve the efficiency of the refinement loop by incorporating multiple counterexamples in each iteration or by utilizing the static analysis to obtain an initial approximation of the set of ambiguous operations.

We have implemented the $\widehat{\mathrm{Pa}}(\widehat{\mathrm{D}}, \widehat{\mathrm{N}})$ functor domain and evaluated it against several existing software verification tools on a collection of pointer-manipulating programs in [LKR22]. In many cases, existing tools only consider a single fixed heap order, which is a source of unsoundness. In Appendix D, we demonstrate that using our abstract domain, this unsoundness can be repaired at only a very modest performance cost. Additionally, we show that, even though many

```
x ← malloc(sizeof(i32)) ❷
y ← malloc(sizeof(i32)) ❸
if (x < y) ❶
    /* ... */
free(y)
w ← malloc(sizeof(i32))
z ← malloc(sizeof(i32))
len ← w - z
if (len > 10)
    /* ... */
```

During the initial iteration of the refinement loop, the verifier identifies an ambiguous operation at position ❶. Subsequently, the refinement procedure commences, wherein allocations that participated in the ambiguous operation, i.e., allocations ❷ and ❸, are annotated using LLVM metadata. Next, a dataflow analysis is executed by LART from the annotated allocations. The outcome of this analysis is the following abstracted program with abstract operations represented by underlined symbols, while the remaining operations are carried out concretely.

```
x ← malloc(sizeof(i32))
y ← malloc(sizeof(i32))
if (x < y)
    /* ... */
free(y)
w ← malloc(sizeof(i32)) ❺
z ← malloc(sizeof(i32)) ❻
len ← w - z ❹
if (len > 10)
    /* ... */
```

During the second iteration, the verifier executes the abstracted operations represented by underlined symbols in the $\widehat{\mathrm{Pa}}$ domain and detects the next ambiguous operation at position ❹. Subsequently, the refinement process annotates the allocations at positions ❺, and ❻. Note that the abstract comparison of variables `x` and `y` does not produce an ambiguity error during the second iteration as it has already been performed in the $\widehat{\mathrm{Pa}}$ domain, and ambiguity errors are not produced from this domain. Finally, the third iteration concludes without discovering any additional ambiguous operations.

Figure 10.2: An example of refinement.

verifiers ignore it, ambiguous behavior is present in a considerable fraction of programs from software verification competition (SV-COMP).

Both the proposed abstraction and the refinement loop were implemented in the software model checker DIVINE [Bar+17], which sources can be found at the supplementary web page: `https://divine.fi.muni.cz/2021/pointers/`.

10.2.2 Programs Decompilation

We employed a similar syntactic refinement approach in a newly proposed technique of program decompilation presented in [Kor+20]. Specifically, we focused on the decompilation of x86_64 machine code into LLVM IR, which entailed translating instructions to recover their operation semantics, extracting control flow, and address identification. Our main contribution in this paper was refining the indirect control flow by using data flow analysis, which made the resulting bitcode much more suitable for formal analysis. This involved utilizing both syntactic abstraction and symbolic execution techniques to infer program control flow.

[Kor+20]: Korenčik et al. (2020), "On Symbolic Execution of Decompiled Programs"

One of the major obstacles to understanding program behavior (whether by humans or algorithmically) is dynamic control flow: the simplest case are conditional jumps, which are quite well understood and comparatively easy to reason about. Function-local control flow without indirect jumps is typically encoded in control flow graphs, which are a simple, static objects which give a very good picture of the behavior of the given function.

Subroutine calls and indirect jumps both constitute a slightly different type of additional inconvenience: in case of indirect jumps, especially in machine-level programs, the set of their targets is not always clear or even possible to compute reliably and the ideal of a static control-flow graph breaks down. Subroutines, on the other hand, pose a similar problem, but not at the subroutine entry point, since in direct calls, this could be easily captured with a standard control flow graph. Instead, the problem arises when control flow returns from the subroutine to the call site: the return address is, in this case, dynamic, since there are possibly many call sites which call into the same subroutine (that is, after all, the reason subroutines exist). As long as the calls themselves are direct (and hence the target of the call instruction is statically known), it is at least possible to enumerate the call sites to which a function could return, making inter-procedural data flow analysis feasible, if not easy.

Unfortunately, when indirect calls are involved,[2] the situation becomes much more problematic: the forward edge (the call itself) is no longer easily resolved (and again, in machine-level programs, the forward edges cannot be reliably enumerated for the same reasons as with indirect intra-procedural jumps). This also means that enumerating call sites for a given subroutine becomes much harder, since every indirect call site could possibly call any of the functions in the program.[3]

Since reconstruction of vtables and similar artefacts from machine code is error-prone and specific to a combination of platform, operating system and C++ compiler (in case of C++ vtables – with hand-coded function pointer tables, this becomes even more of a hit-and-miss affair), it is

2: Indirect calls often arise due to late binding in C++ programs: object instances which are capable of late binding carry a pointer to a so-called vtable, which, for each late-bound method, lists the address of the subroutine which implements the particular method in the given object instance.

3: A simple heuristic that can improve situation in this case is that functions whose address never appears in the program outside of direct call instructions cannot be invoked using an indirect jump. However, the heuristic is not completely reliable and should be avoided in rigorous verification scenarios (it is entirely possible to store, for example, just an offset of the entry point from another known address, e.g., from another function, and reconstruct the entry point address at runtime, without the literal address ever appearing in program text). Nonetheless, in practice and for non-obfuscated programs, the heuristic works very well. Unfortunately, it does not cover some common patterns, like C++ virtual functions, or analogous constructs used in C programs.

desirable to resolve indirect calls in a more general and automated fashion. To this end, we have devised an approach based on dynamic methods and gradual refinement, which replaces each indirect call with a direct call to a synthetic helper function, which only uses conditional branches and direct calls to replicate the effects of the original indirect call. The resulting structure is much more transparent to further inter-procedural control-flow and data-flow analyses.

The algorithm proceeds as follows:

1. replace each indirect call with a call to a switch box, that is, a call-site-specific synthetic function which takes the indirect address as an argument,
2. synthesize empty switch boxes for each of the call sites replaced in step 1 – an empty switch box indicates the desired target address it has been passed, and aborts execution,
3. abstract away all input values and explore the state space of the program, noting any errors raised by the switch boxes,
4. amend each switch box which appears in a counterexample as follows:
   a) add a conditional branch which checks the argument for equality with the address indicated in the counter-example,
   b) if they match, proceed to directly call this target function,
   c) otherwise proceed with the previous version of the switch box;
5. repeat steps 3 and 4 until no further counter-examples in the switch boxes appear.

The choice of abstract domain in step 3 then controls the trade-off between precision and cost: coarser abstractions lead to smaller state spaces and the algorithm proceeds more quickly. However, they also produce larger switch boxes, which in turn cause downstream analyses to give coarser results. The output program after the process no longer contains any indirect calls.

Similar to the heap layout abstraction approach, where we begin by underapproximating the pointers that need to be abstracted, in our static control flow recovery method, we underapproximate the potential call destinations and only expand the switch box when presented with a specific counterexample, which yields an error on missing case of a switch box. To achieve this, we utilized LART for switch box instrumentation and general program abstraction, which is required to examine all possible program paths. As a result of this devirtualization, the analysis of reaching abstraction becomes also more precise, requiring a smaller portion of a program to be abstracted.

10.2.3 Analysis of Incomplete Program Fragments

Our bachelor student J. Šárnik [Šár22] used a similar refinement technique to perform an analysis of program fragments. *Program fragments* are programs that do not contain the entire system under test. They do not necessarily contain a main entry point, such as a single function whose arguments are not defined in the fragment. Program analysis can be performed on these fragments individually, similar to unit testing. However, unlike in unit testing, we take into account all ambiguous inputs and interactions with the rest of the system that are absent in the analyzed fragment.

[Šár22]: Šárník (2022), "Automatická analýza neúplných programů"

It is often advantageous to perform program analysis incrementally by fragments since partial results work as analysis summaries that can be reused in later stages of analysis. For example, a function may be invoked several times, but there is no need to re-interpret it each time; we can simply utilize its summary.

In work presented by J. Šárnik, we employed LART and its metadomains to infer such fragment summaries. We leveraged LART to generate "program drivers" that wrapped functions as self-contained programs. These drivers use abstract values to provide function arguments and resolve other program interactions.

Obtaining a summary for the fragment is a straightforward process that involves interpreting the driver in a term domain and tracking the relationship between function arguments and results (assuming the fragment is side-effect free). An illustrative example is provided in Figure 10.3.

```
fn foo(a, b)
    if a != 0
        error
    c = b + 1
    if c == 0
        error
```

Figure 10.3: An example of program fragment with error in case `a` $\neq$ `0` or `a` $=$ `0` and `b` $=$ `-1`.

In contrast to a complete program, we usually cannot expect to find errors independent of parameter values in a program fragment. The total amount of possible parameter combinations is, however, often intractably large. The goal of the discussed approach is to identify those parameter values of program fragments that lead to an error in a given fragment. [Šár22]

We utilized a combination of the unit domain ($\mathcal{A}_\star$) and the slicing domain ($\mathcal{A}_{\star\!\!\!/}$), as discussed in Chapter 4, to identify the parameters in the fragments that cause errors. The intuition behind the algorithm is the following. We iteratively test all possible combinations of parameter abstractions using these two domains. For example, if we abstracted `a` as $\star$ and `b` as $\star\!\!\!/$ in the program shown in Figure 10.3, only the first error would be reachable, since we sliced away the part of the program dependent on the second parameter. Using a product of term domain and unit domain, we accumulate the term precondition for an error to be reachable, which in this case is `a` $\neq$ `0`. This condition can serve as a summary precondition, allowing us to determine whether an error will occur without reinterpreting the program. By testing various parameter combinations, we can deduce which errors are dependent on which parameters, as well as which parameters cannot trigger any errors at all. In such cases, we could safely ignore those parameters in the summary.

This approach enabled us to reduce the amount of symbolic reasoning required by extracting only the preconditions necessary for safe analysis. To avoid the need for new infrastructure to handle summaries, we synthesize summaries as programs that construct and verify computed preconditions and then substitute the corresponding program fragments with those preconditions. This approach can be viewed as a form of

syntactic refinement, where functions are gradually rewritten to their simplified forms that represent their summaries.

10.3 Future Work

The central concept behind compilation-based abstraction is to reframe interpretation-based methods as a domain and employ syntactic abstraction and refinement to reduce abstraction costs. The design through syntactic abstraction might improve dynamic techniques as some program invariants can be inferred statically, allowing the compiler to summarize parts of abstract programs. Consequently, the primary areas for further exploration involve the adaptation of further abstraction-based techniques as domains. Exploring techniques like lazy abstraction, runtime widening and narrowing, and policy iteration could prove worthwhile.

Throughout the thesis, we delved into various prospects for future research, particularly in Section 8.2, where we examined potential modifications to the techniques employed in symbolic execution. We believe that further promising directions are to adapt concolic execution, as well as other precision-loss countermeasures, such as state merging, runtime function summaries, and bounded analysis. These techniques could complement syntactic abstraction, enabling the compiler to deduce loop unrolling or employ concrete values gleaned from concolic execution.

One interesting avenue to explore is also guided program exploration, where we assign weights to branches or enable the domain to guide uncertain path exploration.

Another area for potential research is adapting abstract domains for analyzing concurrent programs with weak memory models in a sound manner. To accomplish this, it is necessary to account for store buffers and all possible orderings of reads/writes from shared memory in the program. One approach is to utilize aggregate domains, which will model the various potential memory manipulations in their implementation of `access` and `update` operations.

Previously, we implemented a similar technique in DIVINE; however, it lacked the benefit of an abstraction framework and only permitted analysis with concrete values for different memory models [ŠRB16].

[ŠRB16]: Štill et al. (2016), "Weak Memory Models as LLVM-to-LLVM Transformations"

The primary focus of this thesis was on error reachability and safety. However, it may be advantageous to broaden the scope of compilation-based abstraction to encompass more expressive property checking, such as LTL model checking. One potential strategy involves recasting LTL model checking as a domain, where the domain represents the specification automaton. During program execution, the steps of the automaton can be performed synchronously with program steps, effectively implementing the product of the automaton and program execution.

One advantage of utilizing compilation-based abstraction for LTL is that it allows for easy reference from the automaton (abstract domain) to program variables since both use the same language. Furthermore, employing syntactic abstraction makes it possible to identify all potential locations that may modify the atomic propositions of the property being examined, which allows us not to perform an automaton step after each

program instruction. Furthermore, we could create a monitor domain to simply verify local properties after specific operations or modifications to certain variables.

Finally, DIVINE offers the capability of system abstraction. Since DiOS, the verification-dedicated operating system, is a component of the system under verification – especially it is compiled to bitcode, which is processed by LART, the abstraction can be extended to and from specific parts of DiOS. For instance, by utilizing nondeterministic values, it is feasible to simulate abstractly ambiguous steps of the system clocks, and LART propagates all nondeterministic information throughout DiOS and the system's userspace. Consequently, an intriguing avenue for future research involves investigating potential system domains that abstract the environment's behavior and primitives, such as the filesystem, networking, clock, or system calls.

Summary

In conclusion, this chapter described technical aspects of compilation-based abstraction. We discussed its implementation in various applications and explored future directions for its development.

Notably, we examined the integration of compilation-based abstraction into the explicit-state model checker DIVINE and provided a comprehensive overview of its architecture. The technical implementation of runtime for abstractions in DIVINE was also described, along with its application to symbolic model checking.

We illustrated the importance of syntactic refinement for efficient program abstraction and showcased its application in the refinement of heap abstraction using the pointer arithmetic domain, the refinement of dynamic control flow in decompilation, and the verification of partial programs.

Finally, we discussed possible future research directions for compilation-based abstraction. We emphasized the potential for recasting interpretation-based techniques as program domains, such as concolic execution, backward constraint propagation, LTL model checking, or system abstraction.

11 Epilogue

In this thesis, we have studied program abstraction techniques in the context of program verification. A particular focus was given to a novel technique called compilation-based abstraction, which was introduced in this research. This technique involves compiling (instrumenting) abstract semantics directly into the program. The thesis presented both theoretical and practical advancements related to this subject.

The motivation behind compilation-based abstraction was to create a self-contained program abstraction solution that simplifies the design of program analysis tools. By using program transformation during compilation, abstraction is directly integrated into the concrete program, which releases verification tools from the responsibility of abstraction. The fundamental concept of this approach is to express abstract semantics using concrete computation, which makes the abstracted program understandable by any explicit tool. Furthermore, it enabled us to take advantage of the benefits of compiled (static) program abstraction. By performing the abstraction once, we can analyze the program multiple times using various tools, different domains, or even running the abstraction natively.

We have successfully demonstrated the applicability of the technique by showcasing its integration with the off-the-shelf model checker DIVINE, as well as its ability to perform abstract execution. Further, we were able to compile abstraction into native binaries, which binary-analysis tools and debuggers can examine. This highlights the versatility of the technique and its potential to be utilized in various practical applications. Furthermore, we have provided a library for designing custom domains and a tool called LART– the LLVM Abstraction & Refinement Tool, which enables automatic program abstraction.

In addition to the aforementioned results, this thesis has also focused on the general topic of abstraction. We have discussed three main areas of program abstraction: scalar abstraction, aggregate abstraction, and dynamic memory abstraction. In scalar abstraction, we have delved into how to reframe symbolic computation into program semantics. We explored a novel control-explicit data-abstract approach of software analysis. In aggregate abstraction, we have focused explicitly on string abstraction and its extension to abstract semantics of C library functions. We devised a novel efficient string abstract domain, which considers specificities of C string manipulating programs. Lastly, we have addressed the challenge of ambiguous dynamic memory layout and demonstrated how pointers abstraction could effectively address this problem. The novel abstractions were also covered in accompanying publications.

The scalability of abstractions to larger problems heavily relies on refinement. Therefore, this thesis has also delved into topics of both interdomain and intradomain refinement. We have discussed a relational enhancement for non-relational abstract domains that can be directly integrated into the program. Additionally, we have studied multidomain analysis and explored ways to effectively resolve interactions between various

domains. Finally, we have introduced counterexample-guided abstraction refinement for syntactic abstraction and applied it to multiple scenarios like dynamic memory analysis, program decompilation, or incremental program analysis.

Moreover, the approach of compilation-based abstraction can be further generalized beyond value abstractions. We can view this technique as a form of semantic translation, allowing us to apply the approach to other program analysis techniques. If we can describe the relationship between concrete computation and the desired technique, then we can potentially apply the compilation-based approach.

In this thesis, we have discussed the possibility of translating various techniques to this approach, such as concolic execution, domain refinement, or program slicing. These areas open up exciting possibilities for future research and development and offer the potential for improving the accuracy and efficiency of program analysis in self-contained and innovative ways.

In conclusion, this thesis has provided insights into various aspects of compilation-based abstraction, along with related research in this area. We have presented theoretical and technical advancements and demonstrated their applicability with real-world examples and implementations.

Appendix

Evaluation of Symbolic Abstraction in DIVINE

A

In the study of compilation-based symbolic computation for model checking, we incorporated the approach into the explicit-state model checker DIVINE. The results presented here are from the original paper published in 2018 [LRB18]. However, since then, the domain and tool have been further developed and improved. The evaluation aimed to address two research questions:

[LRB18]: Lauko et al. (2018), "Symbolic Computation via Program Transformation"

RQ1 To what extent does the use of compilation-based abstraction simplify the design and implementation of the model-checking tool?

RQ2 How does the IR-based analysis with compilation-based symbolic abstraction compare to traditional interpretation-based techniques in terms of performance?

First of all, we have measured the performance of the transformation itself. On C programs from the SV-COMP suite, the transformation time was negligible. On more complex C++ programs, it took at most a few seconds, which is still fast compared to subsequent analysis.

A.1 Code Complexity

One factor considered in the compilation-based abstraction design was the reduction of code complexity. While the number of lines of code is not a precise measure of complexity, it serves as a useful approximation and is easily obtained. The results of this metric are summarized in Table A.1.

Components	DIVINE	KLEE	SymDIVINE	CBMC
Transformation	3.2	0	0	(22)
Virtual machine	(10)	15	6	7.5
Exploration	(1.5)	1.2	1	2.3
Solver integration	1.2	8	0	14
SAT solver	(45)	(45)	(23)	(5.5)
SMT solver	(80)	(80)	(400)	16
Runtime support	1	0	0	0
Total unique	5.4	24.2	7	39.8
Total shared	136.5	125	423	27.5

Table A.1: Summary of component sizes (thousands of lines of code) in a few symbolic verification and symbolic execution tools. Numbers in parentheses represent shared code (i.e. code not specific to the given approach to symbolic computation).

SV-COMP Benchmarks

We evaluated our compilation-based abstraction approach using a subset of the svcomp [Bey16] benchmarks, specifically 7 categories, summarised

[Bey16]: Beyer (2016), "Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016)"

Table A.2: Summary of benchmarks from SV-COMP. The time limit was 10 min and memory limit was 10 GiB. The *oot/oom* column is the number of test cases that did not finish within the limits, while *solved* are those that gave the expected result; *states* gives the number of states stored, *search* gives the state space exploration time and *ce* gives the counterexample generation time.

category	solved	oot/oom	states	search	ce
array	96	94	170.3k	52:00	54:15
bitvector	17	15	3166	3:12	2:33
loops	72	106	14.0k	53:52	11:40
product-lines	336	239	20.2M	4:36:44	43:11
pthread	9	36	609.4k	3:31	0:54
recursion	47	34	3955	16:16	7:41
systemc	14	45	25.0k	3:29	1:34
total	591	569			

Table A.3: The number of benchmarks correctly solved by each of the evaluated tools. The best result in each category is rendered in boldface.

category	total	DIVINE	SymDIVINE	CBMC
array	190	**96**	68	93
bitvector	32	**17**	9	2
loops	178	**72**	67	9
product-lines	575	336	**411**	234
pthread	45	**9**	0	1
recursion	81	**47**	43	22
systemc	59	14	**27**	0
total	1160	591	**625**	361

in Table A.2, along with statistics from our prototype tool. We have only taken examples with finite state spaces since the prototype could not handle infinite recursion or infinite accumulation loops. In total, we have selected 1160 SV-COMP inputs. In many cases (especially in the array category), the benchmarks are parametric: we have included both the original SV-COMP instance and smaller instances to check that the approach works correctly, even if it takes a long time or exceeds the memory limit on the instances included in SV-COMP. In all cases, the time limit, for each test case separately, was 10 minutes (wall time) and the memory limit was 10 GiB. The test machines were equipped with 4 Intel Xeon 5130 cores clocked at 2 GHz and 16 GiB of RAM. In addition to the presented approach, we have measured two additional tools: CBMC 5.8 and SymDIVINE, both of which are symbolic model checkers targeting C code. The overall results of the comparison, in terms of the number of cases solved, are presented in Table A.3.

[CKL04]: Clarke et al. (2004), "A Tool for Checking ANSI-C Programs"

Comparison 1: CBMC The results from CBMC 5.8 were obtained using the tool's default configuration. CBMC [CKL04] is a mature bounded model checker for C programs with a good track record in SV-COMP and is built around a symbolic interpreter for *goto programs*, its own intermediate form, not entirely dissimilar to CIL or LLVM in its spirit. Besides KLEE, the CBMC toolkit is among the best established members of the interpretation-based school of symbolic computation.

In addition to analyzing the overall number of benchmarks solved within the given time limit, we also sought to compare the amount of time required to solve each case. These findings are presented in Table A.4.

Table A.4: Speed comparison: the columns models_1 and models_2 show the number of models which the respective pair of tools finished in common. In most cases, CBMC is substantially faster than the proposed approach, while SymDIVINE is significantly slower. The time shown is a sum across all the models in a given category.

category	models_1	DIVINE	CBMC	models_2	DIVINE	SymDIVINE
array	73	34:16	13:58	58	3:18	42:54
bitvector	2	0:37	0:01	9	0:55	2:30
loops	4	0:03	0:02	62	22:25	19:04
product-lines	182	4:08:24	7:25	183	0:30	28:33
pthread	0	0	0	0	0	0
recursion	22	0:01	0:13	43	4:02	13:58
systemc	0	0	0	14	3:29	6:43

With regard to its state space exploration strategy, CBMC can be thought of as the middle ground between the approach taken by KLEE and that of our proposed tool. On one hand, KLEE, being a symbolic executor, does not attempt to identify already-visited program states. CBMC is a bounded model checker, which means it stores a single formula representing the entire set of reachable states. Our present approach, being based on an explicit-state model checker, stores sets of program states and compares them for equality using an SMT solver.

Comparison 2: SymDIVINE [Mrá+16] is a pre-existing, interpretation-based symbolic model checker which also works with LLVM bitcode. Similar to our approach, SymDIVINE relies on a state equality checker, in this case based on quantified bit-vector formulae. In theory, this yields coarser state equivalence and consequently smaller state spaces, but we could not confirm this in our set of benchmarks: the total number of states stored across the benchmarks that finished using both tools was 802 thousand for SymDIVINE and 93 thousand with the approach described in this paper. Additionally, QBV satisfiability queries are typically much more expensive than those used by our prototype tool, which can help explain the speed difference between the tools.

A.2 Supplementary Materials

The additional resources for can be found at:

- The original paper [LRB18]:
  `https://divine.fi.muni.cz/2018/sym/`

[LRB18]: Lauko et al. (2018), "Symbolic Computation via Program Transformation"

These resources include a binary distribution, benchmarks, additional graphs, detailed results, tutorials on how to use the symbolic abstraction, and sources of the extended model-checker DIVINE, which is open source software distributed under the ISC license.

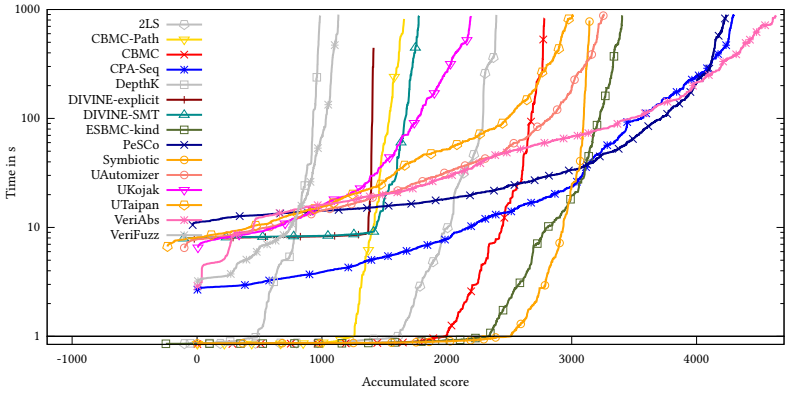
Software Verification Competition Participations

B

We have competed with various instances of compilation-based abstraction in the software verification competition (SV-COMP) over the years. Specifically with the explicit state model checker DIVINE enhanced by various domains and a native abstract execution variant. The competition provides an independent evaluation of tools on different benchmark categories. While we summarize our results here, a more detailed description can be found in the competition reports and accompanying papers of the respective submissions.

B.1 SV-COMP 2019

In 2019, we participated in a software verification competition using only symbolic abstraction in the form of the term domain. One can observe the comparison of results of explicit DIVINE and the same tool employing compilation-based abstraction. The results indicate that our approach did not impact the tool's performance, and it improved the tool's ability to verify benchmarks that involved data nondeterminism.

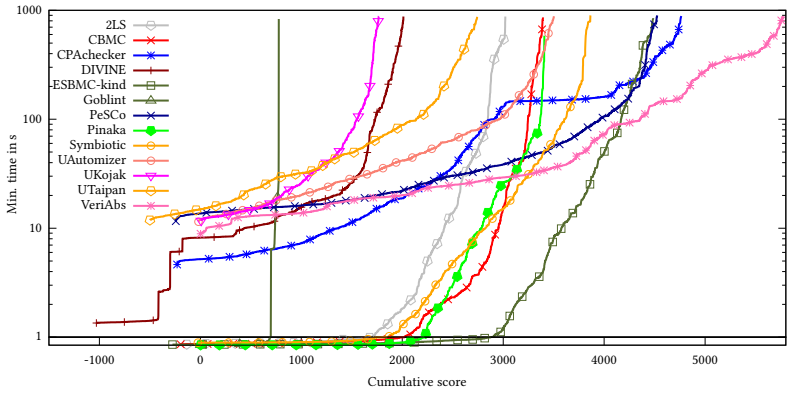


For a more detailed description, readers can refer to the competition report [Bey19] and accompanying submission papers [Lau+19]. Supplementary materials for the submission can be found at:

▶ https://divine.fi.muni.cz/2019/sv-comp-smt/

B.2 SV-COMP 2020 & 2021

In subsequent years, we expanded our abstraction capabilities to include support for floating-point values, array abstraction, and memory safety analysis. Additionally, we utilized a portfolio of domains, specifically the unit domain and the term domain. However, due to minor implementation errors and changes in benchmark specifications, comparing the results with previous ones is difficult.

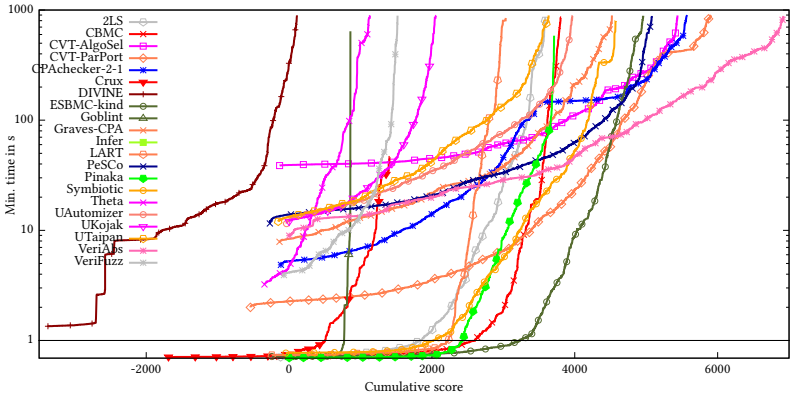


For a more detailed description, readers can refer to the competition reports [Bey20; Bey21]. Supplementary materials for the submission can be found at:

- SV-COMP 2020: `https://divine.fi.muni.cz/2020/sv-comp/`
- SV-COMP 2021: `https://divine.fi.muni.cz/2021/sv-comp/`

B.3 SV-COMP 2022

In 2022, we participated in the competition with a native runtime for program abstraction denoted as LART in the graph. It was performant on programs that execute smoothly with nearly concrete values, but it struggled with programs that create difficult instances for ⊠solvers or contained infinite loops. However, the native execution did not support floating points and memory safety. Unfortunately, changes in the benchmark specification made it difficult to compare the results with DIVINE, which participated in the competition as hors concours with the same configuration as the previous year and announced results in a different format.



For a more detailed description, readers can refer to the competition reports [Bey22b] and accompanying submission papers [LR22].

Evaluation of M-String in DIVINE

C

We conducted an evaluation of the M-String abstraction in several scenarios to demonstrate its properties:

1. We showed that abstract versions of standard functions are more efficient than transforming concrete versions using only abstract string accesses and updates.
2. We investigated several implementations of standard library functions and transformed them automatically to ensure their results agreed with those generated by M-String library operations.
3. We evaluated M-String instantiation with symbolic characters on a set of benchmarks from real software that contained buffer-overflow errors. In this scenario, we showed that M-String could efficiently detect real-world bugs and prove that a program did not contain them after fixing them.

The final benchmark demonstrated the use of abstractions on more complex C programs. Specifically, we analyzed automatically generated parsers from bison and flex tools on abstract (M-String) inputs. The resource limits for all scenarios were consistent, with each verification run limited to 4 processing units (cores), 80 GB of memory, and 1 hour of CPU time. The processor used to run the benchmarks was an AMD EPYC 7371 clocked at 2.60GHz.

C.1 M-String Operations

We conducted benchmarks to compare the efficiency of abstract domain operations with automatically abstracted implementations of standard library functions from PDCLib, a public-domain libc implementation, using only essential abstract operations: `lift`, `update`, and `access`. The benchmarks were divided into two groups, and for each standard library function and input type, we created an isolated benchmark in two variants. The first variant used an abstract semantics of M-String operations, as shown in Table C.1, while the second variant only used an automatic abstraction of essential aggregate operations, as shown in Table C.2.

The results, presented in Table C.1, were measured using parametrized M-String inputs of two types, where l represents the input's parametric length:

1. **Word** w is a string of the form: $w = c_1^{i_1} \cdot c_2^{i_2} \cdot \cdots \cdot c_l^{i_l}$ where $\sum k = 1^l i_k \leq l$ and c_j for $1 \leq j \leq l$ is an arbitrary character from character domain $\widehat{\mathsf{Ch}}$.
2. **Sequence** w is a string of the form $w = c^i$, where $0 \leq i \leq l$ and c is an element of a character domain $\widehat{\mathsf{Ch}}$.

Table C.1: The table presents measurements of M-String operations on two input types, **Word** and **Sequence**, as described in Section C.1. Each benchmark measures the state space size and verification time (in seconds) for M-String inputs of a bounded length. The table also includes the average transformation time (LART), measured in seconds. It is worth noting that the state space size does not vary with the input length, as explained in more detail in Section C.1.

	Word					
		Verification (s)				
	States	8	64	1024	4096	LART (s)
`strcmp`	3562	480.0	498.0	472.0	481.0	1.70
`strcpy`	368	9.8	9.1	9.3	9.4	1.70
`strcat`	7398	898.0	873.0	865.0	843.0	1.72
`strchr`	49	0.3	0.4	0.3	0.3	1.71
`strlen`	78	1.1	1.2	1.0	1.3	1.71

	Sequence					
		Verification (s)				
	States	8	64	1024	4096	LART (s)
`strcmp`	70	0.26	0.24	0.21	0.25	1.76
`strcpy`	48	0.20	0.20	0.21	0.20	1.71
`strcat`	105	0.51	0.52	0.53	0.51	1.71
`strchr`	15	0.04	0.04	0.03	0.04	1.70
`strlen`	16	0.05	0.04	0.05	0.06	1.81

The first notable difference between automatically abstracted implementations of library functions and M-String operations is that the analysis of the former timeouts for input strings longer than 64 characters. The main cause of the lifted implementation's inefficiency is that it has to iterate over all characters, while M-String operations leverage iteration over larger segments. This difference also causes a blow-up of the model checker's state space for the lifted implementations while the state space size does not change for M-String operations. The reason for this is the fact that the number of segments does not change with the length of the input. Therefore, M-String operations always perform the same computation independently of the M-String length.

Table C.2: The performance evaluation of standard library functions was performed by abstracting them using only the M-String definitions of access and update operations. It focused on input Sequences of 8, 64, and 1024 characters in size. Most instances timed out when attempting to conduct benchmarks with Word strings.

	Sequence					
	8		64		1024	
	Time(s)	States	Time(s)	States	Time(s)	States
`strcmp`	1.24	197	260.00	1597	T	–
`strcpy`	0.70	122	61.50	962	T	–
`strcat`	15.80	1102	T	–	T	–
`strchr`	0.04	16	0.05	16	0.05	16
`strlen`	0.19	46	9.57	326	T	–

C.2 C Standard Libraries

In the second group of benchmarks (shown in Table C.3 and Table C.4), we investigate whether the implementation from several standard libraries matches the expected results of abstract implementation. In other words, we perform an equivalence check of results obtained from M-String operations with the results of the automatically abstracted (originally concrete) standard library functions. We expect that both give the same results. For the evaluation, we picked three open-source libraries: PDClib, musl-libc

and μCLibc. Since results for the libraries are rather similar, we present here only an evaluation of PDClib functions. The remaining results are provided in the Supplementary Material. All benchmarks showed that our implementation matches the standard one.

Table C.3: Verification results of functions from PDCLib with timeout of 1 hour with Word input. Measurements show the size of state space and verification time for the parametric length of the input.

	Word					
	4		8		16	
	Time(s)	States	Time(s)	States	Time(s)	States
`strcmp`	14.3	1005	105.0	2989	1350.0	9741
`strcpy`	5.2	515	57.4	1823	912.0	6935
`strcat`	468.0	5748	T	–	T	–
`strchr`	0.1	22	0.1	22	0.1	22
`strlen`	0.7	91	4.1	259	68.8	883

Table C.4: Verification results of functions from PDCLib with timeout of 1 hour with Sequence input. Measurements show the size of state space and verification time for the parametric length of the input.

	Sequence					
	4		8		16	
	Time(s)	States	Time(s)	States	Time(s)	States
`strcmp`	2.2	204	5.1	376	16.5	720
`strcpy`	0.8	183	2.5	347	9.1	675
`strcat`	8.6	751	113	2535	1940	9463
`strchr`	0.3	17	0.3	17	0.4	17
`strlen`	0.2	34	0.3	54	0.7	94

As in the previous case, the benchmarks presented in this section also suffer from a state space blow-up due to the exponential number of possible character combinations. Therefore, to mitigate this issue, we decreased the size of the input strings. Additionally, the concrete implementations of string accesses and updates result in large SMT formulae, causing long solver times. It is worth noting that the computation analysis with Word input, which has more segments, results in longer execution times than the analysis with Sequence. This is because more segments naturally cause overhead for the analyses. For example, M-String needs to consider cases where some segments have zero length, which leads to hard SMT queries since, in the worst case, it needs to check all possible strings for given segment bounds and characters.

C.3 Veriabs Overflow Benchmarks

In this scenario, presented in Table C.5, we demonstrate that the M-String domain is capable of efficiently detecting overflow bugs. The Veriabs benchmarks exhibit overflow errors and corrected variants of real-world software. To ensure the soundness of the analysis, we instantiate the M-String with a term domain for characters, enabling reasoning about strings of symbolic length. However, a drawback of this instantiation is that when the length of the string bounds a loop, we may need to infinitely unroll the loop in the analysis, resulting in timeouts in the correct benchmarks.

Table C.5: The Veriabs overflow benchmarks consist of several categories of programs that exhibit overflow errors and their corresponding corrected variants. The table presents the number of successfully solved benchmarks (tests), along with the total time taken for each category. For each category, the table depicts correctly verified benchmarks, benchmarks where the verifier was able to find an error and number of timeouts

	Correct		Error Found		
	Tests	Time(s)	Tests	Time(s)	Timeouts
`apache`	0	–	26	384.26	24
`openser`	46	234.13	45	105.93	6
`wu-ftpd`	8	35.78	14	2461.27	19
`libgd`	4	9.01	4	1.85	0
`madwifi`	5	0.51	5	0.55	0
`gxine`	1	0.53	1	0.25	0

C.4 Parsers

Lastly, we evaluate our implementation on more complex programs: automatically generated parsers. For the generation, we use a tool Bison. It reads a language specification in the form of context-free grammar and produces a C parser that accepts the language. In the benchmarks, we generate two such parsers. The first one accepts a language of numerical expressions (mathematical expressions that consist of numbers and binary operators). The second parser is of a simple programming language with variables and branching. We present a evaluation for both parsers in Table C.6. As with the previous benchmark sets, the M-String inputs with a smaller number of segments outperformed other analyses. In these benchmarks, we use specifically hand-crafted M-String inputs for parsers. For parsing of mathematical expressions, it was: addition input had a form of two arbitrary numbers with a plus sign between them, ones was a simple input of a single digit sequence, and lastly, alternation was input that produced complicated M-Strings by alternating digits inside of expressions. The other parser of simple programming language was evaluated on: value was in input that created a variable and assigned a constant to it, loop was a short program with some control flow and wrong was a program that contained a syntax error.

Table C.6: Measurements of time and size of state space for analyses of automatically generated parsers.

	Numeric Expressions Grammar					
	10		20		35	
	Time(s)	States	Time(s)	States	Time(s)	States
`add`	40.2	416	319.0	3548	T	–
`ones`	5.5	62	8.1	196	189	2186
`alter`	708	105	1582.0	11k	T	–
`value`	6.6	38	90.4	488	1100	4988
`loop`	1.5	23	4.9	23	33	23
`wrong`	7.3	82	67.7	892	311	8992

C.5 Supplementary Materials

The additional resources for can be found at:

- The original paper [Roč+19]: `https://divine.fi.muni.cz/2019/mstring/`
- The extended version [Lau+20]: `https://divine.fi.muni.cz/2020/mstring/`

[Roč+19]: Ročkai et al. (2019), "String Abstraction for Model Checking of C Programs"

[Lau+20]: Lauko et al. (2020), "Abstracting Strings for Model Checking of C Programs"

These resources include a binary distribution, benchmarks, tutorials on how to use the string abstraction, and sources of the extended model-checker DIVINE, which is open source software distributed under the ISC license.

Evaluation of Heap Layout Abstraction

D

In this section, we examine properties of the proposed domain and of our refinement approach. First, in Section D.1, we present a qualitative comparison of the new approach with existing tools on synthetic, layout-sensitive programs. These programs were inspired by the usage of pointer comparisons in real-world code, as outlined in Section 6.3. In Section D.2, we discuss the impact of the abstraction and of the refinement loop on verification performance. All benchmarks are accessible from the supplementary page.*

D.1 Tool Limitations

For the purpose of tool comparison, we split the test programs into multiple categories described below. Each program included an ambiguous operation that determined the accessibility of an error location. When a tool failed to detect an error location in a category or reported an infeasible counterexample, we marked it with (✘). This often meant that the tool explored only a subset of possible paths, resulting in a false negative, although in some cases, tools imprecisely represented possible memory locations, leading to a false positive. In the following, we describe the features tested in the benchmark categories.[1] The evaluation of tools is summarized subsequently in Figure D.1.

1: All comparisons were performed in a defined way, i.e., pointers are converted into `uintptr_t` before ambiguous operation. In the examples, these conversions are omitted for brevity.

1. Allocation order benchmarks verify that for two allocations:

```
void *x = malloc(sizeof(int));
void *y = malloc(sizeof(int));
```

1. both `x < y` and `x > y` are feasible,
2. and `x == y` is infeasible.

This is when the analysis detects failures of all the following assertions: `assert(x < y)`, `assert(x > y)` and, `assert(x == y)`.

---

2. Memory layout benchmarks verify for all types of pointers, including heap pointers, stack pointers, and global pointers that:

```
void *g = &some_global;
void *x = malloc(sizeof(int));
void *y = malloc(sizeof(int));
char a[1] = {0}; // stack pointer
```

1. all addresses are unique,
2. heap, stack, and global memory locations can be in an arbitrary order, i.e. we verify all possible relations on values of `g`, `x`, `y` and `a` as in *allocation order* category.

---

* https://divine.fi.muni.cz/2021/pointers/

3. Transitive relations benchmarks verify in multiple scenarios with:

```
void *x = malloc(sizeof(int));
void *y = malloc(sizeof(int));
void *z = malloc(sizeof(int));
```

whether pointers keep their relations, i.e., if `x < y` and `y < z` hold than `x < z` is required to hold.

---

4. Subtraction benchmarks verify that subtraction of pointers:

```
void *x = malloc(sizeof(int));
void *y = malloc(sizeof(int));
if (x - y > 0)
    /* ... */
```

results in both possibilities, i.e., one where the result of subtraction is positive and the other when it is negative.

---

5. Reconstruction benchmarks verify that reconstruction of a pointer from an array bytes keeps the pointer relations:

```
void *x = malloc(sizeof(int));
void *y = malloc(sizeof(int));
void *new_x = to_pointer(fragments(x));
void *new_y = to_pointer(fragments(y));
```

where `fragments` function stores argument pointer to a returned byte array and function `to_pointer` reconstructs a pointer from an array of bytes. Finally, we check, that a tool finds all feasible orderings on reconstructed pointers `new_x`, `new_y`.

---

6. Fragmentation benchmarks verify that relational operation on pointer bytes (fragments) is also treated as ambiguous:

```
void *x = malloc(sizeof(int));
void *y = malloc(sizeof(int));
char *fx = fragments(x);
char *fy = fragments(y);
if (fx[5] != fy[7])
    /* ... */
```

where `fragments` function stores argument pointer to a returned byte array.

---

7. Lifetime analysis benchmarks verify whether allocations are capable of reusing addresses:

```
void *x = malloc(sizeof(int));
uintptr_t xv = uintptr_t(x);
free(x);
void *y = malloc(sizeof(int));
uintptr_t yv = uintptr_t(y);
assert(xv != yv); // fails
```

8. Overlapping benchmarks check that pointer locations overlap after some addition:

```
void *x = malloc(sizeof(int));
uintptr_t xv = uintptr_t(x);
void *y = malloc(sizeof(int));
uintptr_t yv = uintptr_t(y);
long offset = __VERIFIER_nondet_ulong();
assert(xv != yv + offset); // fails
```

9. Nondeterministic pointer benchmarks check that allocation can return an arbitrary pointer:

```
void *x = malloc(sizeof(char));
void *y = __VERIFIER_nondet_pointer();
assert(x != y); // fails
```

10. Reallocation benchmarks check that reallocation can use the same address:

```
void *x = malloc(100);
void *y = realloc(x, 200);
assert(x != y); // fails
```

	DIVINE-$\widehat{\text{Pa}}$ [LKR22]	DIVINE [Bar+17]	KLEE [CDE08]	CBMC [CKL04]	ESBMC [Mor+14]	CPAchecker [BK11]	Symbiotic [Cha+20]	Predator [DPV11]	Nidhugg [KS17]	2LS [SK16]
version	4.2	4.2	2.2	5.12.3	6.2	1.9	7	0.0.3	0.2	0.8.2
allocation order	✓	✘	✘	✘	✘	✓	✓	✓	✘	✘
memory layout	✓	✘	✘	✘	✓	✘	✓	✓	✘	✘
trans. relations	✓	✘	✘	✘	✓	✘	✘	✘	✘	✘
subtraction	✓	✘	✘	✘	✓	✓	✓	U	✘	✘
reconstruction	✓	✘	✘	U	✘	✘	U	✘	✘	✘
fragmentation	✘	✘	✘	✘	✘	✘	U	✘	✘	✘
lifetime analysis	✓	✘	✓	✘	✘	✓	✘	✘	✘	✘
reallocation	✓	✘	✘	✘	✘	✓	✘	✓	✘	✘
nondet pointer	✓	✓	✓	✓	✓	✓	✓	✓	U	✘
overlapping	✓	✓	✓	✘	✘	✓	✘	✓	U	✘

Figure D.1: Comparison of sound reasoning about programs with errors dependent on ambiguous behavior. The ✓ symbol indicates that a tool was able to detect errors on all possible paths. The ✘ mark denotes that at least one error path was missed or the tool has detected an unreachable error. The U symbol marks an unknown result when a tool reported that it does not know how to proceed. For evaluation, we have used the listed version of tools. Links to relevant sources and used options can be found in the supplementary material: https://divine.fi.muni.cz/2021/pointers/.

We have evaluated the instantiation $\widehat{\text{Pa}}(\mathcal{C}, \mathcal{T})$ – the dereference part of pointers was represented concretely, while the numeric part was represented using the term domain. The verifier made use of an SMT solver for bit-vector logic, as discussed in Section 6.5.2. To the best of our knowledge (and our evaluation supports this conclusion) this setup provides sound treatment of pointer arithmetic and comparisons.

Unfortunately, in our current implementation, the modifications which allow the model checker to detect ambiguous operations are somewhat incomplete. In particular, bitwise operations which decompose a pointer may go unreported, and hence a refinement step may be incorrectly skipped – this is what causes the failure of the `fragmentation` benchmark. However, the problem only affects refinement and is not present when all pointer operations are performed in the $\widehat{\text{Pa}}$ domain unconditionally.

D.2 Abstraction & Refinement Performance

```
void insert(node *n, void *v)
{
  if (std::less(v, n->value))
    {
    if (!n->left)
      n->left = make_node(v);
    else
      insert(n->left, v);
  } else {
    if (!n->right)
      n->right = make_node(v)
   ;
    else
      insert(n->right, v);
  }
}
```

Data structures that have a data layout dependent on pointer values pose a challenging problem for $\widehat{\text{Pa}}$ domain reasoning. For instance, while performing abstract execution of insertion into a binary search tree, as illustrated in the code, the program explores all possible tree shapes, resulting in a combinatorial explosion of $f(n) = \sum_{i=1}^{n} f(i-1)f(n-i)$ possibilities for n nodes. One potential solution to this problem is to enhance the abstract domain with a shape analysis [Yan+08] that simultaneously represents multiple abstract trees.

Figure D.2: An example of hard to reason program.

To demonstrate the usability of the domain, we evaluate DIVINE on benchmarks from the software verification competition SV-COMP.[†] We have also used DIVINE to count benchmarks which execute ambiguous operations (using the option `-o abort:ptrcmp`). The results are summarized in column **tasks** in Table D.1. For memory safety, 64 of 303 tasks were classified to contain an ambiguous operation. For the reachability category, it was 419 from 2210, and the least amount of ambiguous operations was in the concurrency category, which contains only a few benchmarks with dynamic memory allocation.

Overall, the benchmark set consisted of 3587 benchmarks, with 531 containing some behavior dependent on ambiguous pointer operation, which is 15%. All experiments were performed with an identical set of resource constraints: 10 minutes of CPU time, 15 GB of RAM and 2 CPU cores.[‡] This part of the evaluation consists of two scenarios.

In the first scenario, we demonstrate the effect of refinement on verification performance (Table D.1) by comparing DIVINE with the refinement loop enabled against a version of DIVINE that unconditionally abstracts all allocations in the program.

The first scenario shows that refinement substantially decreases the number of memory allocation operations that need to be abstracted (see Table D.1). For this reason, the refinement approach solves more benchmarks, especially those that do not contain any ambiguity, since no or only very little abstraction is necessary in those cases.

In the second scenario, we demonstrate that the proposed abstraction has only a modest cost in comparison to DIVINE without pointer abstraction (Figure D.3).

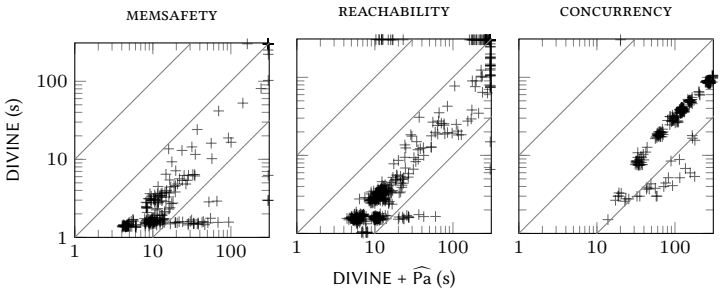


Figure D.3: Comparison: plain DIVINE vs DIVINE with the pointer arithmetic domain and refinement.

[†] https://github.com/sosy-lab/sv-benchmarks

[‡] The processor used to run the benchmarks was AMD EPYC 7371 clocked at 2.60GHz.

Table D.1: This table compares two configurations of DIVINE: one that leverages refinement loop and another that abstract all heap pointers unconditionally. For each category, we also separately show results on the subset of benchmarks that contain at least one ambiguous operation. Results include the number of correctly **solved** benchmarks, the average number of refinement iterations in those benchmarks (**iter.**) and the number of abstract allocations (**abs.**; μ – average, $\tilde{x}$ – median, σ – standard deviation). Abstraction statistics only include benchmarks solved by both configurations.

	DIVINE-$\widehat{\text{Pa}}$ + Refine						DIVINE-$\widehat{\text{Pa}}$ Abstract All					
	solved	**iter.**	**abs.**	μ	$\tilde{x}$	σ	**solved**	**abs.**	μ	$\tilde{x}$	σ	**tasks**
MEMSAFETY	263	0.19	1369	7.4	0	30.0	185	6242	33.7	18	49.4	303
ambiguous	49	1.30	1369	27.9	10	53.5	46	2531	55.0	36	61.1	64
REACHABILITY	685	0.27	373	0.9	0	4.1	319	848	2.6	0	7.7	2210
ambiguous	140	1.32	373	3.3	1	7.3	111	584	5.3	1	11.6	419
CONCURRENCY	289	0.12	2571	8.9	0	42.0	237	3569	15.1	0	46.2	1074
ambiguous	22	1.63	2571	116.8	68.5	104.7	20	2654	132.7	109.5	98.7	48

The performance results from Figure D.3 show that the proposed abstraction does slow down DIVINE verification, as expected. Most of the overhead measured in these benchmarks is due to program transformation performed by LART. Most of the remainder can be attributed to an exponential slowdown which appears in benchmarks where pointers are repeatedly compared in recursive data structures (e.g. in binary search trees – see Figure D.2). In these cases, pointer abstraction has to consider all possible configurations of the data structures, causing exponential blowup of the state space size.

In theory, a similar path explosion problem might occur when a program contains a loop whose termination condition refers to an ambiguous value, even though this problem appears to be rare in practice. However, if this case is a concern for a particular application, existing techniques compatible with the domain chosen for the numeric parts of pointers may be applicable, e.g. widening, loop summarization or k-induction.

D.3 Supplementary Materials

The additional resources for can be found at:

- The original paper [LKR22]:
  `https://divine.fi.muni.cz/2021/pointers/`

[LKR22]: Lauko et al. (2022), "Verification of Programs Sensitive to Heap Layout"

These resources include a binary distribution, benchmarks, detailed results, tutorials on how to use the pointer refinement loop, and sources of the extended model-checker DIVINE, which is open source software distributed under the ISC license.

Abstraction Optimization Artifacts

E

Optimized original LLVM bitcode:

```
define i32 @compute() {
    %1 = call i32 @__lamp_any_range_i32(i32 0, i32 0)
    %2 = call i32 @__lamp_any_range_i32(i32 1, i32 10)
    %3 = mul i32 %2, %1
    %4 = mul i32 %3, %1
    %5 = mul i32 %4, %3
    ret i32 %5
}
```

Unoptimized instrumented LLVM bitcode:

```
define i32 @compute() {
    %1 = call i32 @__lamp_any_range_i32(i32 0, i32 0)
    %2 = call i1  @__lart_unstash_taint()
    %3 = call i8* @__lart_unstash()
    %4 = call i32 @__lamp_any_range_i32(i32 1, i32 10)
    %5 = call i1  @__lart_unstash_taint()
    %6 = call i8* @__lart_unstash()
    %7 = or i1 %5, %2
    %8 = mul i32 %4, %1
    %9 = call i8* @lart.test.taint.mul.i32.i32(
        i8* (i1, i32, i8*, i1, i32, i8*)* @lart.lifter.mul.i32.i32,
        i1 %5, i32 %4, i8* %6, i1 %2, i32 %1, i8* %3
    )
    %10 = or i1 %7, %2
    %11 = mul i32 %8, %1
    %12 = call i8* @lart.test.taint.mul.i32.i32(
        i8* (i1, i32, i8*, i1, i32, i8*)* @lart.lifter.mul.i32.i32,
        i1 %7, i32 %8, i8* %9, i1 %2, i32 %1, i8* %3
    )
    %13 = or i1 %10, %7
    %14 = mul i32 %11, %8
    %15 = call i8* @lart.test.taint.mul.i32.i32(
        i8* (i1, i32, i8*, i1, i32, i8*)* @lart.lifter.mul.i32.i32,
        i1 %10, i32 %11, i8* %12, i1 %7, i32 %8, i8* %9
    )
    call void @__lart_stash(i1 %13, i8* %15)
    ret i32 %14
}
```

Optimized instrumented LLVM bitcode:

```
define i32 @compute() {
    %1 = call i32 @__lamp_any_range_i32(i32 0, i32 0)
    %2 = call i1  @__lart_unstash_taint()
    %3 = call i8* @__lart_unstash()
    %4 = call i32 @__lamp_any_range_i32(i32 1, i32 10)
    %5 = call i1  @__lart_unstash_taint()
    %6 = call i8* @__lart_unstash()
    %7 = or i1 %2, %5
    %8 = mul i32 %4, %1
    br i1 %7, label %abstract.path, label %lart.test.taint.mul.i32.i32

    abstract.path:
        br i1 %5, label %merge.arg.1, label %lift.arg.1
```

```
    lift.arg.1:
        %9 = call i8* @__lamp_wrap_i32(i32 %4)
        br label %merge.arg.1

    merge.arg.1:
        %10 = phi i8* [%6, %abstract.path], [%9, %lift.arg.1]
        br i1 %2, label %merge.arg.1.mul, label %lift.arg.2

    merge.arg.1.mul:
        %11 = call i8* @__lamp_mul(i8* %10, i8* %3)
        br label %lart.lifter.mul.exit

    lift.arg.2:
        %12 = call i8* @__lamp_wrap_i32(i32 %1)
        %13 = call i8* @__lamp_mul(i8* %10, i8* %12)
        %14 = call i8* @__lamp_wrap_i32(i32 %1)
        br label %lart.lifter.mul.exit

    lart.lifter.mul.exit:
        %15 = phi i8* [%13, %lift.arg.2], [%11, %merge.arg.1.mul]
        %16 = phi i8* [%14, %lift.arg.2], [%3, %merge.arg.1.mul]
        %17 = call i8* @__lamp_mul(i8* %15, i8* %16)
        %18 = call i8* @__lamp_mul(i8* %17, i8* %15)
        br label %lart.test.taint.mul.i32.i32

    lart.test.taint.mul.i32.i32:
        %ret.op.i4 = phi i8* [%18, %lart.lifter.mul.exit], [null, %0]
        %.pn = mul i32 %8, %1
        %19 = mul i32 %.pn, %8
        call void @__lart_stash(i1 %7, i8* %ret.op.i4)
        ret i32 %19, !dbg !26
}
```

Optimized LLVM bitcode linked with unit domain:

```
define i32 @compute() {
    %1 = call i8* @new(i64 1)
    %2 = bitcast i8* %1 to %lamp::storage*
    %3 = getelementptr %lamp::storage, %lamp::storage* %2,
         i64 0, i32 0, i32 0, i32 0
    call void @__lart_stash(i1 zeroext true, i8* nonnull %3)
    %4 = call i1  @__lart_unstash_taint()
    %5 = call i8* @__lart_unstash()
    %6 = call i8* @new(i64 1)
    %7 = bitcast i8* %6 to %lamp::storage*
    %8 = getelementptr %lamp::storage, %lamp::storage* %7,
         i64 0, i32 0, i32 0, i32 0
    call void @__lart_stash(i1 zeroext true, i8* nonnull %8)
    %9  = call i1  @__lart_unstash_taint()
    %10 = call i8* @__lart_unstash()
    %11 = or i1 %4, %9
    br i1 %11, label %abstract.path, label %lart.test.taint.mul.i32.i32

    abstract.path:
        %12 = call i8* @new(i64 1)
        %13 = bitcast i8* %12 to %lamp::storage*
        %14 = getelementptr %lamp::storage, %lamp::storage* %13,
              i64 0, i32 0, i32 0, i32 0
        br label %lart.test.taint.mul.i32.i32

    lart.test.taint.mul.i32.i32:
        %ret.op.i4 = phi i8* [%14, %abstract.path], [null, %0]
        call void @__lart_stash(i1 %11, i8* %ret.op.i4)
        ret i32 0
}
```

Bibliography

[Ahr+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, eds. *Deductive Software Verification - The KeY Book. From Theory to Practice*. Springer, Dec. 16, 2016 (cited on pages 132, 136).

[AGC12] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. "Craig interpretation." In: *Static Analysis. 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*. Springer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 300–316. DOI: 10.1007/978-3-642-33125-1_21 (cited on page 118).

[Alb+12a] Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. "Ufo: A Framework for Abstraction- and Interpolation-Based Software Verification." In: *Computer Aided Verification. 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Ed. by Parthasarathy Madhusudan and Sanjit A. Seshia. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 672–678. DOI: 10.1007/978-3-642-31424-7_48 (cited on page 131).

[Alb+12b] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. "Lazy abstraction with interpolants for arrays." In: *Logic for Programming, Artificial Intelligence, and Reasoning. 18th International Conference, LPAR-18, Merida, Venezuela, March 11-15, 2012, Proceedings*. Springer. 2012, pp. 46–61. DOI: 10.1007/978-3-642-28717-6_7 (cited on pages 82, 117).

[Alb+12c] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. "SAFARI: SMT-Based Abstraction for Arrays with Interpolants." In: *Computer Aided Verification. 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Ed. by Parthasarathy Madhusudan and Sanjit A. Seshia. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 679–685. DOI: 10.1007/978-3-642-31424-7_49 (cited on pages 82, 117).

[ALS06] Ali Almossawi, Kelvin Lim, and Tanmay Sinha. *Analysis tool evaluation: Coverity prevent*. Tech. rep. Carnegie Mellon University, 2006 (cited on page 103).

[Ama+17] Roberto Amadini, Alexander Jordan, Graeme Gange, François Gauthier, Peter Schachte, Harald Søndergaard, Peter J. Stuckey, and Chenyi Zhang. "Combining String Abstract Domains for JavaScript Analysis: An Evaluation." In: *Tools and Algorithms for the Construction and Analysis of Systems. 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. Springer Berlin Heidelberg, 2017, pp. 41–57. DOI: 10.1007/978-3-662-54577-5_3 (cited on page 86).

[AMS20] Gianluca Amato, Maria Chiara Meo, and Francesca Scozzari. "On collecting semantics for program analysis." In: *Theoretical Computer Science* 823 (2020), pp. 1–25. DOI: 10.1016/j.tcs.2020.02.021 (cited on page 35).

[AGT08] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. "Demand-Driven Compositional Symbolic Execution." In: *Tools and Algorithms for the Construction and Analysis of Systems. 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008, Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Springer Berlin Heidelberg, 2008, pp. 367–381. DOI: 10.1007/978-3-540-78800-3_28 (cited on page 137).

[And+17] Pavel Andrianov, Karlheinz Friedberger, Mikhail Mandrykin, Vadim Mutilin, and Anton Volkov. "CPA-BAM-BnB: Block-abstraction memoization and region-based memory models for predicate abstractions. (Competition Contribution)." In: *Tools and Algorithms for the Construction and Analysis of Systems. 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*. Springer. Springer Berlin Heidelberg, 2017, pp. 355–359. DOI: 10.1007/978-3-662-54580-5_22 (cited on page 126).

[AMK21] Pavel Andrianov, Vadim Mutilin, and Alexey Khoroshilov. "CPALockator: Thread-Modular Analysis with Projections." In: *Tools and Algorithms for the Construction and Analysis of Systems. 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings, Part II*. Springer. Springer, 2021, pp. 423–427. DOI: 10.1007/978-3-030-72013-1_25 (cited on page 126).

[Avg+14] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. "Automatic exploit generation." In: *Communications of the ACM* 57.2 (2014), pp. 74–84. DOI: 10.1145/2560217.2560219 (cited on page 157).

[Bag+05] Roberto Bagnara, Patricia M. Hill, Elisa Ricci, and Enea Zaffanella. "Precise widening operators for convex polyhedra." In: *Science of Computer Programming* 58.1 (2005). Special Issue on the Static Analysis Symposium 2003, pp. 28–56. DOI: 10.1016/j.scico.2005.02.003 (cited on page 48).

[BHZ04] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. "Widening Operators for Powerset Domains." In: *Verification, Model Checking, and Abstract Interpretation. 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*. Ed. by Bernhard Steffen and Giorgio Levi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 135–148. DOI: 10.1007/978-3-540-24622-0_13 (cited on page 67).

[BHZ10] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. "Exact join detection for convex polyhedra and other numerical abstractions." In: *Computational Geometry* 43.5 (2010), pp. 453–473. DOI: 10.1016/j.comgeo.2009.09.002 (cited on page 67).

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008 (cited on page 27).

[Bal+18] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. "A survey of symbolic execution techniques." In: *ACM Computing Surveys (CSUR)* 51.3 (2018), p. 50. DOI: 10.1145/3182657 (cited on pages 97, 138).

[Bar+17] Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. "Model Checking of C and C++ with DIVINE 4." In: *Automated Technology for Verification and Analysis. 15th International Symposium, ATVA 2017, Pune, India, October 3–6, 2017, Proceedings*. Ed. by K. Narayan Kumar Deepak D'Souza. Cham: Springer, 2017, pp. 201–207. DOI: 10.1007/978-3-319-68167-2_14 (cited on pages 9, 95, 183, 207).

[BBR10] Jiří Barnat, Luboš Brim, and Peter Ročkai. "Scalable shared memory LTL model checking." In: *International Journal on Software Tools for Technology Transfer* 12.2 (May 2010), pp. 139–153. DOI: 10.1007/s10009-010-0136-z (cited on page 174).

[Bar+15] Jiří Barnat, Peter Ročkai, Vladimír Štill, and Jiří Weiser. "Fast, Dynamically-Sized Concurrent Hash Table." In: *Model Checking Software*. 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings. Ed. by Bernd Fischer and Jaco Geldenhuys. Cham: Springer, 2015, pp. 49–65. DOI: 10.1007/978-3-319-23404-5_5 (cited on page 174).

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. 2016. URL: www.SMT-LIB.org (cited on page 126).

[BBH18] Damián Barsotti, Andrés M Bordese, and Tomás Hayes. "PEF: Python Error Finder." In: *Electronic Notes in Theoretical Computer Science* 339 (2018), pp. 21–41. DOI: 10.1016/j.entcs.2018.06.003 (cited on page 132).

[BZ20] Anna Becchi and Enea Zaffanella. "PPLite: Zero-overhead encoding of NNC polyhedra." In: *Information and Computation* 275 (2020). DOI: 10.1016/j.ic.2020.104620 (cited on page 132).

[Bel05] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator." In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, p. 41 (cited on page 141).

[BC10] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. 1st. Springer Publishing Company, Incorporated, 2010 (cited on page 14).

[BW04] Jeannet Bertrand and Serwe Wendelin. "Abstracting Call-Stacks for Interprocedural Verification of Imperative Programs." In: *Algebraic Methodology and Software Technology. 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16, 2004, Proceedings*. 2004. DOI: 10.1007/978-3-540-27815-3_22 (cited on page 49).

[Ber+10] Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. "Static Analysis and Verification of Aerospace Software by Abstract Interpretation." In: *American Institute of Aeronautics and Astronautics (AIAA) Infotech@Aerospace 2010* 2 (Apr. 2010). DOI: 10.2514/6.2010-3385 (cited on pages 68, 118).

[Ber+15] Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. "Static Analysis and Verification of Aerospace Software by Abstract Interpretation." In: *Found. Trends Program. Lang.* 2.2-3 (Dec. 2015), pp. 71–190. DOI: 10.1561/2500000002 (cited on pages 68, 118, 131, 132).

[Bey16] Dirk Beyer. "Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016)." In: *Tools and Algorithms for the Construction and Analysis of Systems. 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Marsha Chechik and Jean-François Raskin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 887–904. DOI: 10.1007/978-3-662-49674-9_55 (cited on page 193).

[Bey19] Dirk Beyer. "Automatic Verification of C and Java Programs: SV-COMP 2019." In: *Tools and Algorithms for the Construction and Analysis of Systems. 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part III*. Ed. by Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen. Cham: Springer, 2019, pp. 133–155. DOI: 10.1007/978-3-030-17502-3_9 (cited on pages 2, 3, 197).

[Bey20] Dirk Beyer. "Advances in Automatic Software Verification: SV-COMP 2020." In: *Tools and Algorithms for the Construction and Analysis of Systems. 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II*. Springer, 2020, pp. 347–367. DOI: 10.1007/978-3-030-45237-7_21 (cited on page 198).

[Bey21] Dirk Beyer. "Software Verification: 10th Comparative Evaluation (SV-COMP 2021)." In: *Tools and Algorithms for the Construction and Analysis of Systems. 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings, Part II*. Springer, 2021, pp. 401–422. DOI: 10.1007/978-3-030-72013-1_24 (cited on page 198).

[Bey22a] Dirk Beyer. "Cooperative verification: Towards reliable safety-critical systems (invited talk)." In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems*. 2022, pp. 1–2. DOI: 10.1145/3563822.3572548 (cited on page 126).

[Bey22b] Dirk Beyer. "Progress on Software Verification: SV-COMP 2022." In: *Tools and Algorithms for the Construction and Analysis of Systems. 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part II*. Springer, 2022, pp. 375–402. DOI: 10.1007/978-3-030-99527-0_20 (cited on page 198).

[BF18] Dirk Beyer and Karlheinz Friedberger. "In-Place vs. Copy-on-Write CEGAR Refinement for Block Summarization with Caching." In: *Leveraging Applications of Formal Methods, Verification and Validation. Verification* (Jan. 1, 2018). DOI: 10.1007/978-3-030-03421-4_14 (cited on page 117).

[BGS18] Dirk Beyer, Sumit Gulwani, and David A. Schmidt. "Combining Model Checking and Data-Flow Analysis." In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Springer, Jan. 1, 2018. DOI: 10.1007/978-3-319-10575-8_16 (cited on page 117).

[BHT08] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. "Program analysis with dynamic precision adjustment." In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2008, pp. 29–38. DOI: 10.1109/ase.2008.13 (cited on page 117).

[Bey+05] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. "Checking memory safety with Blast." In: *Fundamental Approaches to Software Engineering. 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Springer. 2005, pp. 2–18. DOI: 10.1007/978-3-540-31984-9_2 (cited on page 117).

[BHT06] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. "Lazy Shape Analysis." In: *Computer Aided Verification. 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Ed. by Thomas Ball and Robert B. Jones. Springer Berlin Heidelberg, Jan. 1, 2006. DOI: 10.1007/11817963_48 (cited on page 117).

[BK11] Dirk Beyer and M. Erkan Keremoglu. "CPAchecker: A Tool for Configurable Software Verification." In: *Computer Aided Verification. 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011, Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 184–190. DOI: 10.1007/978-3-642-22110-1_16 (cited on pages 99, 126, 127, 207).

[BKW10] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. "Predicate Abstraction with Adjustable-Block Encoding." In: *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*. FMCAD '10. Lugano, Switzerland: FMCAD Inc, 2010, pp. 189–198 (cited on page 125).

[BL13] Dirk Beyer and Stefan Löwe. "Explicit-State Software Model Checking Based on CEGAR and Interpolation." In: *Fundamental Approaches to Software Engineering. 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013, Proceedings*. Ed. by Vittorio Cortellessa and Dániel Varró. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 146–162. DOI: 10.1007/978-3-642-37057-1_11 (cited on page 117).

[BP22] Dirk Beyer and Andreas Podelski. "Software Model Checking: 20 Years and Beyond." In: *Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*. Ed. by Jean-François Raskin, Krishnendu Chatterjee, Laurent Doyen, and Rupak Majumdar. Cham: Springer Nature Switzerland, 2022, pp. 554–582. DOI: 10.1007/978-3-031-22337-2_27 (cited on page 126).

[BW20] Dirk Beyer and Heike Wehrheim. "Verification artifacts in cooperative verification: Survey and unifying component framework." In: *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles. 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part I*. Springer. 2020, pp. 143–167. DOI: 10.1007/978-3-030-61362-4_8 (cited on page 126).

[Bie+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. "Bounded model checking." In: *Advances in computers* 58.11 (2003), pp. 117–148. DOI: 10.1016/s0065-2458(03)58003-2 (cited on pages 126, 139).

[Bie+09] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. "Bounded model checking." In: *Handbook of satisfiability*. 2009, pp. 457–481 (cited on page 38).

[Bir40] Garrett Birkhoff. *Lattice Theory*. American Mathematical Society: Colloquium publications v. 25; v. 1940. American Mathematical Society, 1940 (cited on page 15).

[BBM97] Nikolaj Bjørner, Anca Browne, and Zohar Manna. "Automatic generation of invariants and intermediate assertions." In: *Theoretical Computer Science* 173.1 (1997), pp. 49–87. DOI: 10.1016/s0304-3975(96)00191-0 (cited on page 113).

[BMR12] Nikolaj S. Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. "Program Verification as Satisfiability Modulo Theories." In: *IJCAR* 20 (2012), pp. 3–11 (cited on page 126).

[Bla+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "A Static Analyzer for Large Safety-critical Software." In: *SIGPLAN Not.* 38.5 (May 2003), pp. 196–207. DOI: 10.1145/780822.781153 (cited on pages 48, 132).

[BCD22] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. "Handling Memory-Intensive Operations in Symbolic Execution." In: *15th Innovations in Software Engineering Conference*. ISEC 2022. Gandhinagar, India: Association for Computing Machinery, 2022. DOI: 10.1145/3511430.3511453 (cited on page 89).

[Bou12] Mehdi Bouaziz. "TreeKs: A Functor to Make Numerical Abstract Domains Scalable." In: *Electronic Notes in Theoretical Computer Science* 287 (2012). Proceedings of the Fourth International Workshop on Numerical and Symbolic Abstract Domains, NSAD 2012, pp. 41–52. DOI: 10.1016/j.entcs.2012.09.005 (cited on pages 68, 118).

[Bou+09] Olivier Bouissou, Eric Conquet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Khalil Ghorbal, E Goubault, David Lesens, Laurent Mauborgne, Antoine Miné, Sylvie Putot, Xavier Rival, and Michel Turin. "Space Software Validation using Abstract Interpretation." In: *The International Space System Engineering Conference : Data Systems in Aerospace - DASIA 2009* (May 2009) (cited on page 132).

[BH19] Rémy Boutonnet and Nicolas Halbwachs. "Disjunctive Relational Abstract Interpretation for Interprocedural Program Analysis: 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13–15, 2019, Proceedings." In: Jan. 2019, pp. 136–159. DOI: 10.1007/978-3-030-11245-5_7 (cited on page 49).

[Bra+14] Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet. "IKOS: A Framework for Static Analysis Based on Abstract Interpretation." In: *Software Engineering and Formal Methods. 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014, Proceedings*. Ed. by Dimitra Giannakopoulou and Gwen Salaün. Cham: Springer, 2014, pp. 271–277. DOI: 10.1007/978-3-319-10431-7_20 (cited on pages 37, 130, 132).

[BHW09] Richard Bubel, Reiner Hähnle, and Benjamin Weiß. "Abstract Interpretation of Symbolic Execution with Explicit State Updates." In: *Formal Methods for Components and Objects: 7th International Symposium, FMCO 2008, Sophia Antipolis, France, October 21-23, 2008, Revised Lectures*. Berlin, Heidelberg: Springer, 2009, pp. 247–277 (cited on pages 137, 138).

[Bul+17] Tevfik Bultan, Fang Yu, Muath Alkhalaf, and Abdulbaki Aydin. *String Analysis for Software Verification and Security*. Springer, 2017 (cited on page 85).

[Bur+90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. "Symbolic model checking: 10^{20} states and beyond." In: *5th Symposium in Logic in Computer Science (LICS)*. June 1990, pp. 428–439. DOI: 10.1109/LICS.1990.113767 (cited on pages 3, 38).

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224 (cited on pages 3, 102, 126, 127, 131, 132, 141, 157, 207).

[Cad+08] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. "EXE: Automatically Generating Inputs of Death." In: *ACM Trans. Inf. Syst. Secur.* 12.2 (2008). DOI: 10.1145/1455518.1455522 (cited on pages 132, 133, 138).

[CE20] Cruz Camacho and Alejandro Elkin. "Static Analysis of Python Programs using Abstract Interpretation: An Application to Tensor Shape Analysis." In: 2020 (cited on page 133).

[Cas19] Stephen Cass. *The Top Programming Languages 2019. IEEE Spectrum Magazine.* Available: https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019. Accessed 10 February 2020. 2019 (cited on page 85).

[Cha+12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. "Unleashing Mayhem on Binary Code." In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy.* SP '12. USA: IEEE Computer Society, 2012, pp. 380–394. DOI: 10.1109/sp.2012.31 (cited on page 98).

[Cha+21] Marek Chalupa, Tomáš Jašek, Jakub Novák, Anna Řechtáčková, Jan Strejček, and Veronika Šoková. "Symbiotic 8: Beyond Symbolic Execution (Competition Contribution)." In: *Tools and Algorithms for the Construction and Analysis of Systems. 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings, Part II.* Vol. 12652. Cham: Springer, 2021, pp. 453–457. DOI: 10.1007/978-3-030-72013-1_31 (cited on pages 3, 126).

[Cha+20] Marek Chalupa, Tomáš Jašek, Lukáš Tomovič, Martin Hruška, Veronika Šoková, Paulína Ayaziová, Jan Strejček, and Tomáš Vojnar. "Symbiotic 7: Integration of Predator and More. (Competition Contribution)." In: *Tools and Algorithms for the Construction and Analysis of Systems. 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II.* Ed. by David Parker Armin Biere. Germany: Springer, 2020, pp. 413–417. DOI: 10.1007/978-3-030-45237-7_31 (cited on page 207).

[CSV19] Marek Chalupa, Jan Strejček, and Martina Vitovská. "Joint forces for memory safety checking revisited." In: *International Journal on Software Tools for Technology Transfer* (Aug. 2019). DOI: 10.1007/s10009-019-00526-2 (cited on page 102).

[CR08] Bor-Yuh Evan Chang and Xavier Rival. "Relational inductive shape analysis." In: *ACM SIGPLAN Notices.* Vol. 43. 1. ACM. 2008, pp. 247–260. DOI: 10.1145/1328897.1328469 (cited on pages 77, 99).

[Che+15] Yu-Fang Chen, Chiao Hsieh, Ming-Hsien Tsai, Bow-Yaw Wang, and Farn Wang. "CPArec: Verifying Recursive Programs via Source-to-Source Program Transformation. (Competition Contribution)." In: *Tools and Algorithms for the Construction and Analysis of Systems. 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings.* Ed. by Christel Baier and Cesare Tinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 426–428. DOI: 10.1007/978-3-662-46681-0_35 (cited on page 126).

[Che+22] Ju Chen, WookHyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. "SYMSAN: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis." In: *31st USENIX Security Symposium (USENIX Security 22).* Boston, MA: USENIX Association, Aug. 2022, pp. 2531–2548 (cited on pages 3, 5, 46, 133).

[Chi+09] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. "Selective Symbolic Execution." In: (Jan. 2009) (cited on page 140).

[CKC12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "The S2E Platform: Design, Implementation, and Applications." In: *ACM Trans. Comput. Syst.* 30.1 (2012). DOI: 10.1145/2110356.2110358 (cited on pages 126, 132, 138, 140, 141).

[Cho+96] Fred Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. "Effective representation of aliases and indirect memory operations in SSA form." In: *Compiler Construction. 6th International Conference, CC '96, Linköping, Sweden, April 24 - 26, 1996. Proceedings.* Ed. by Tibor Gyimóthy. Springer. 1996, pp. 253–267. DOI: 10.1007/3-540-61053-7_66 (cited on page 40).

[CMS03] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. "Precise analysis of string expressions." In: *International Static Analysis Symposium.* Springer. 2003, pp. 1–18. DOI: 10.7146/brics.v10i5.21776 (cited on page 85).

[Cim+16] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. "Infinite-state invariant checking with IC3 and predicate abstraction." In: *Formal Methods in System Design* 49.3 (Dec. 2016), pp. 190–218. DOI: 10.1007/s10703-016-0257-4 (cited on pages 117, 118).

[Cla+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-Guided Abstraction Refinement." In: *Computer Aided Verification. 12th International Conference, CAV 2000 Chicago, IL, USA, July 15-19, 2000 Proceedings*. Ed. by E. Allen Emerson and Aravinda Prasad Sistla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 154–169. DOI: 10.1007/10722167_15 (cited on pages 3, 38, 116, 128, 139, 181).

[Cla+98] Edmund M. Clarke, Ernest Allen Emerson, Somesh Jha, and Aravinda Prasad Sistla. "Symmetry reductions in model checking." In: *Computer Aided Verification. 10th International Conference, CAV'98, Vancouver, BC, Canada, June 28-July 2, 1998, Proceedings*. Ed. by Alan J. Hu and Moshe Y. Vardi. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 147–158. DOI: 10.1007/3-540-36384-x_5 (cited on page 174).

[CES86] Edmund M. Clarke, Ernest Allen Emerson, and Aravinda Prasad Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications." In: *ACM Transactions on Programming Languages and Systems* 8.2 (1986), pp. 244–263. DOI: 10.1145/5397.5399 (cited on page 14).

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs." In: *Tools and Algorithms for the Construction and Analysis of Systems. 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. Ed. by Kurt Jensen and Andreas Podelski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15 (cited on pages 14, 102, 126, 127, 194, 207).

[Cla+05] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. "SATABS: SAT-based predicate abstraction for ANSI-C." In: *Tools and Algorithms for the Construction and Analysis of Systems. 11th International Conference, TACAS 2005, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2004, Proceedings*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Springer. Springer Berlin Heidelberg, 2005, pp. 570–574. DOI: 10.1007/978-3-540-31980-1_40 (cited on page 117).

[Cor+15] J. Robert M. Cornish, Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. "Analyzing Array Manipulating Programs by Program Transformation." In: *Logic-Based Program Synthesis and Transformation. 24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9-11, 2014. Revised Selected Papers*. Ed. by Maurizio Proietti and Hirohisa Seki. Cham: Springer, 2015, pp. 3–20. DOI: 10.1007/978-3-319-17822-6_1 (cited on page 82).

[Cor+95] Agostino Cortesi, Gilberto Filé, Roberto Giacobazzi, Catuscia Palamidessi, and Francesco Ranzato. "Complementation in abstract interpretation." In: *Static Analysis. Second International Symposium, SAS '95, Glasgow, UK, September 25 - 27, 1995. Proceedings*. Ed. by Alan Mycroft. Springer Berlin, Heidelberg, Jan. 1, 1995. DOI: 10.1007/3-540-60360-3_35 (cited on pages 67, 118).

[CO18] Agostino Cortesi and Martina Olliaro. "M-String Segmentation: A Refined Abstract Domain for String Analysis in C Programs." In: *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. Aug. 2018, pp. 1–8. DOI: 10.1109/TASE.2018.00009 (cited on pages 85, 86).

[CZ11] Agostino Cortesi and Matteo Zanioli. "Widening and narrowing operators for abstract interpretation." In: *Computer Languages, Systems & Structures* 37 (Apr. 2011), pp. 24–42. DOI: 10.1016/j.cl.2010.09.001 (cited on pages 48, 139).

[Cos+05] Alexandru Costan, Stéphane Gaubert, Éric Goubault, Matthieu Martel, and Sylvie Putot. "A Policy Iteration Algorithm for Computing Fixed Points in Static Analysis of Programs." In: *Computer Aided Verification. 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*. Ed. by Kousha Etessami and Sriram K. Rajamani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 462–475. DOI: 10.1007/11513988_46 (cited on page 48).

[CFC11] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. "Static analysis of string values." In: *Formal Methods and Software Engineering. 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings*. Springer. 2011, pp. 505–521. DOI: 10.1007/978-3-642-24559-6_34 (cited on page 85).

[CFC15] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. "A Suite of Abstract Domains for Static Analysis of String Values." In: *Software: Practice and Experience* 45.2 (2015), pp. 245–287. DOI: 10.1002/spe.2218 (cited on pages 85, 86).

[Cou12] Patrick Cousot. "Formal Verification by Abstract Interpretation." In: *NASA Formal Methods. 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012, Proceedings*. Ed. by Alwyn E. Goodloe and Suzette Person. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–7. DOI: 10.1007/978-3-642-28891-3_3 (cited on page 35).

[Cou21] Patrick Cousot. *Principles of Abstract Interpretation*. MIT Press, 2021 (cited on pages 16, 60).

[CC77a] Patrick Cousot and Radhia Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints." In: *Proceedings of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252. DOI: 10.1145/512950.512973 (cited on pages 27, 33, 35, 36, 48, 49, 85).

[CC77b] Patrick Cousot and Radhia Cousot. "Static determination of dynamic properties of generalized type unions." In: *Proceedings of an ACM Conference on Language Design for Reliable Software*. Vol. 12. 3. ACM. ACM Press, New York, NY, 1977, pp. 77–94. DOI: 10.1145/390018.808314 (cited on page 77).

[CC77c] Patrick Cousot and Radhia Cousot. "Static determination of dynamic properties of recursive procedures." In: *IFIP Conference on Formal Description of Programming Concepts, St. Andrews, NB, Canada*. North-Holland Publishing Company. 1977, pp. 237–277 (cited on pages 49, 77).

[CC79] Patrick Cousot and Radhia Cousot. "Systematic Design of Program Analysis Frameworks." In: *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '79. San Antonio, Texas: ACM Press, New York, NY, 1979, pp. 269–282. DOI: 10.1145/567752.567778 (cited on page 67).

[CC92] Patrick Cousot and Radhia Cousot. "Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation." In: *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*. PLILP '92. Berlin, Heidelberg: Springer, 1992, pp. 269–295 (cited on pages 48, 139).

[CC01] Patrick Cousot and Radhia Cousot. "Compositional Separate Modular Static Analysis of Programs by Abstract Interpretation." In: *Proceedings of the Second International Conference on Advances in Infrastructure for E-Business, E-Science and E-Education on the Internet, SSGRR 2001*. Compact disk, L'Aquila, Italy: Scuola Superiore G. Reiss Romoli, 2001 (cited on page 49).

[CC10] Patrick Cousot and Radhia Cousot. "A gentle introduction to formal verification of computer systems by abstract interpretation." In: *Logics and Languages for Reliability and Security*. Ed. by Javier Esparza, Bernd Spanfelner, and Orna Grumberg. Vol. 25. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2010, pp. 1–29. DOI: 10.3233/978-1-60750-100-8-1 (cited on page 14).

[CC14] Patrick Cousot and Radhia Cousot. "Abstract interpretation: past, present and future." In: *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. CSL-LICS '14. Vienna, Austria: Association for Computing Machinery, July 2014. DOI: 10.1145/2603088.2603165 (cited on page 77).

[Cou+05] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "The ASTREÉ Analyzer." In: *Programming Languages and Systems. 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Ed. by Mooly Sagiv. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 21–30. DOI: 10.1007/978-3-540-31987-0_3 (cited on pages 103, 131).

[Cou+07] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "Combination of Abstractions in the ASTRÉE Static Analyzer." In: *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*. Ed. by Mitsu Okada and Ichiro Satoh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 272–300. DOI: 10.1007/978-3-540-77505-8_23 (cited on pages 59, 67, 119).

[CCL11a] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. "A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis." In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. POPL '11. 2011, pp. 105–118. DOI: 10.1145/1926385.1926399 (cited on pages 83, 84, 86).

[CCL11b] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. "A parametric segmentation functor for fully automatic and scalable array content analysis." In: *ACM SIGPLAN Notices* 46.1 (2011), pp. 105–118. DOI: 10.1145/1925844.1926399 (cited on pages 77, 82).

[CCM10] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. "A Scalable Segmented Decision Tree Abstract Domain." In: *Time for Verification: Essays in Memory of Amir Pnueli*. Ed. by Zohar Manna and Doron A. Peled. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 72–95. DOI: 10.1007/978-3-642-13754-9_5 (cited on pages 68, 118).

[CCM11] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. "The Reduced Product of Abstract Domains and the Combination of Decision Procedures." In: *Foundations of Software Science and Computational Structures. 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011, Proceedings*. Ed. by Martin Hofmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 456–472. DOI: 10.1007/978-3-642-19805-2_31 (cited on pages 49, 67, 119).

[Cyt+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph." In: *ACM Transactions on Programming Languages and Systems* 13.4 (Oct. 1991), pp. 451–490. DOI: 10.1145/115372.115320 (cited on page 22).

[Dan+13] Andrei Marian Dan, Yuri Meshman, Martin Vechev, and Eran Yahav. "Predicate abstraction for relaxed memory models." In: *Static Analysis. 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2012, Proceedings*. Springer. 2013, pp. 84–104. DOI: 10.1007/978-3-642-38856-9_7 (cited on page 50).

[DB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *Tools and Algorithms for the Construction and Analysis of Systems. 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008, Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Budapest, Hungary: Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24 (cited on pages 137, 175).

[DOM21] David Delmas, Abdelraouf Ouadjaout, and Antoine Miné. "Static Analysis of Endian Portability by Abstract Interpretation." In: *Static Analysis. 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings*. Ed. by Cezara Drăgoi, Suvam Mukherjee, and Kedar Namjoshi. Cham: Springer, 2021, pp. 102–123. DOI: 10.1007/978-3-030-88806-0_5 (cited on page 49).

[DS07] David Delmas and Jean Souyris. "Astrée: From Research to Industry." In: *Static Analysis*. Ed. by Hanne Riis Nielson and Gilberto Filé. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 437–451 (cited on page 132).

[Det+14] Morgan Deters, Andrew Reynolds, Tim King, Clark W. Barrett, and Cesare Tinelli. "A tour of CVC4: How it works, and how to use it." In: *2014 Formal Methods in Computer-Aided Design (FMCAD)* (2014), pp. 7–7. DOI: 10.1109/fmcad.2014.6987586 (cited on page 175).

[DOY06] Dino Distefano, Peter W. O'hearn, and Hongseok Yang. "A local shape analysis based on separation logic." In: *Tools and Algorithms for the Construction and Analysis of Systems. 12th International Conference, TACAS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*. Ed. by Holger Hermanns and Jens Palsberg. Springer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 287–302. DOI: 10.1007/11691372_19 (cited on page 99).

[DRS03] Nurit Dor, Michael Rodeh, and Shmuel Sagiv. "CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C." In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*. 2003, pp. 155–167. DOI: 10.1145/780822.781149 (cited on page 85).

[Dud+12] Kamil Dudka, Petr Müller, Petr Peringer, and Tomáš Vojnar. "PREDATOR: A verification tool for programs with dynamic linked data structures. (Competition Contribution)." In: *Tools and Algorithms for the Construction and Analysis of Systems. 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 – April 1, 2012, Proceedings*. Ed. by Cormac Flanagan and Barbara König. Springer. Springer Berlin Heidelberg, 2012, pp. 545–548. DOI: 10.1007/978-3-642-28756-5_45 (cited on page 99).

[DPV11] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. "Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic." In: *Computer Aided Verification. 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011, Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 372–378. DOI: 10.1007/978-3-642-22110-1_29 (cited on pages 99, 207).

[Dur+16] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. "Spot 2.0 — a framework for LTL and ω-automata manipulation." In: *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*. Vol. 9938. Springer, Oct. 2016, pp. 122–129. DOI: 10.1007/978-3-319-46520-3_8 (cited on page 175).

[Dur+22] Alexandre Duret-Lutz, Etienne Renault, Maximilien Colange, Florian Renkin, Alexandre Gbaguidi Aisse, Philipp Schlehuber-Caissier, Thomas Medioni, Antoine Martin, Jérôme Dubois, Clément Gillard, and Henrich Lauko. "From Spot 2.0 to Spot 2.10: What's New?" In: *Computer Aided Verification. 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part II*. Ed. by Sharon Shoham and Yakir Vizel. Cham: Springer, 2022, pp. 174–187. DOI: 10.1007/978-3-031-13188-2_9 (cited on pages 9, 175).

[Ern+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. "The Daikon system for dynamic detection of likely invariants." In: *Science of Computer Programming* 69.1 (2007). Special issue on Experimental Software and Toolkits, pp. 35–45. DOI: 10.1016/j.scico.2007.01.015 (cited on page 137).

[EL02] David Evans and David Larochelle. "Improving Security Using Extensible Lightweight Static Analysis." In: *IEEE Software* 19.1 (2002), pp. 42–51. DOI: 10.1109/52.976940 (cited on page 85).

[FG10] Paul Feautrier and Laure Gonnord. "Accelerated Invariant Generation for C Programs with Aspic and C2fsm." In: *Electronic Notes in Theoretical Computer Science* 267.2 (2010). Proceedings of the Tools for Automatic Program AnalysiS (TAPAS), pp. 3–13. DOI: 10.1016/j.entcs.2010.09.014 (cited on pages 48, 139).

[Fer04] Jérôme Feret. "Static Analysis of Digital Filters." In: *Programming Languages and Systems. 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 – April 2, 2004, Proceedings*. Ed. by David Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 33–48. DOI: 10.1007/978-3-540-24725-8_4 (cited on pages 50, 132).

[Fer05] Jérôme Feret. “The Arithmetic-Geometric Progression Abstract Domain.” In: *Verification, Model Checking, and Abstract Interpretation. 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*. Ed. by Radhia Cousot. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 42–58. DOI: 10.1007/978-3-540-30579-8_3 (cited on pages 50, 132).

[FR99] Gilberto Filé and Francesco Ranzato. “The powerset operator on abstract interpretations.” In: *Theoretical Computer Science* 222.1 (1999), pp. 77–111. DOI: 10.1016/S0304-3975(98)00007-3 (cited on pages 57, 58, 67, 118).

[FG05] Cormac Flanagan and Patrice Godefroid. “Dynamic Partial-Order Reduction for Model Checking Software.” In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’05. Long Beach, California, USA: Association for Computing Machinery, 2005, pp. 110–121. DOI: 10.1145/1040305.1040315 (cited on page 38).

[FQ02] Cormac Flanagan and Shaz Qadeer. “Predicate Abstraction for Software Verification.” Version POPL ’02. In: *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* 37.1 (Jan. 2002), pp. 191–202. DOI: 10.1145/503272.503291 (cited on page 3).

[Fle17] Matt Fleming. *A thorough introduction to eBPF*. https://lwn.net/Articles/740157/. Accessed: 2023-02-01. 2017 (cited on page 50).

[Flo93] Robert W. Floyd. “Assigning Meanings to Programs.” In: *Program Verification: Fundamental Issues in Computer Science*. Ed. by Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin. Dordrecht: Springer Netherlands, 1993, pp. 65–81. DOI: 10.1007/978-94-011-1793-7_4 (cited on page 14).

[FMV14] Carlo A. Furia, Bertrand Meyer, and Sergey Velder. “Loop Invariants: Analysis, Classification, and Examples.” In: *ACM Computing Surveys* 46.3 (2014). DOI: 10.1145/2506375 (cited on page 137).

[FM10] Carlo Alberto Furia and Bertrand Meyer. “Inferring Loop Invariants Using Postconditions.” In: *Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*. Ed. by Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 277–300. DOI: 10.1007/978-3-642-15025-8_15 (cited on page 137).

[Gan+13] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. “Abstract Interpretation over Non-lattice Abstract Domains.” In: *Static Analysis. 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2012, Proceedings*. Ed. by Francesco Logozzo and Manuel Fähndrich. Springer, 2013, pp. 6–24. DOI: 10.1007/978-3-642-38856-9_3 (cited on pages 68, 84, 118).

[Gan+16] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. “An Abstract Domain of Uninterpreted Functions.” In: *Verification, Model Checking, and Abstract Interpretation. 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Springer. 2016, pp. 85–103. DOI: 10.1007/978-3-662-49122-5_4 (cited on page 62).

[Gau+07] Stephane Gaubert, Eric Goubault, Ankur Taly, and Sarah Zennou. “Static Analysis by Policy Iteration on Relational Domains.” In: *Programming Languages and Systems*. Ed. by Rocco De Nicola. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 237–252. DOI: 10.1007/978-3-540-71316-6_17 (cited on page 48).

[GS07a] Thomas Gawlitza and Helmut Seidl. “Precise Fixpoint Computation Through Strategy Iteration.” In: *Programming Languages and Systems*. Ed. by Rocco De Nicola. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 300–315. DOI: 10.1007/978-3-540-71316-6_21 (cited on page 48).

[GS07b] Thomas Gawlitza and Helmut Seidl. “Precise Relational Invariants Through Strategy Iteration.” In: *Computer Science Logic*. Ed. by Jacques Duparc and Thomas A. Henzinger. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 23–40 (cited on page 48).

[GS11] Thomas Martin Gawlitza and Helmut Seidl. “Solving Systems of Rational Equations Through Strategy Iteration.” In: *ACM Trans. Program. Lang. Syst.* 33.3 (May 2011), 11:1–11:48. DOI: 10.1145/1961204.1961207 (cited on page 48).

[Gen+18] Jeffrey Gennari, Arie Gurfinkel, Temesghen Kahsai, Jorge A. Navas, and Edward J. Schwartz. "Executable Counterexamples in Software Model Checking." In: *Verified Software. Theories, Tools, and Experiments. 10th International Conference, VSTTE 2018, Oxford, UK, July 18–19, 2018, Revised Selected Papers*. Oxford, United Kingdom: Springer, 2018, pp. 17–37. DOI: 10.1007/978-3-030-03592-1_2 (cited on page 177).

[Ger+19] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. "Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: ACM, 2019, pp. 1069–1084. DOI: 10.1145/3314221.3314590 (cited on pages 50, 132).

[GQ01] Roberto Giacobazzi and Elisa Quintarelli. "Incompleteness, Counterexamples, and Refinements in Abstract Model-Checking." In: *Static Analysis. 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001. Proceedings*. Ed. by Patrick Cousot. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 356–373. DOI: 10.1007/3-540-47764-0_20 (cited on pages 67, 118).

[GRS00] Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. "Making Abstract Interpretations Complete." In: *Journal of the ACM* 47.2 (Mar. 2000), pp. 361–416. DOI: 10.1145/333979.333989 (cited on pages 67, 118).

[GW14] Jeremy Gibbons and Nicolas Wu. "Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl)." In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 339–347. DOI: 10.1145/2628136.2628138 (cited on page 135).

[God07] Patrice Godefroid. "Compositional dynamic test generation." In: *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2007, pp. 47–54. DOI: 10.1145/1190215.1190226 (cited on page 138).

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing." In: *SIGPLAN Not.* 40.6 (2005), pp. 213–223. DOI: 10.1145/1064978.1065036 (cited on pages 97, 132, 133, 138).

[GH06] Laure Gonnord and Nicolas Halbwachs. "Combining Widening and Acceleration in Linear Relation Analysis." In: *Static Analysis*. Ed. by Kwangkeun Yi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 144–160. DOI: 10.1007/11823230_10 (cited on pages 48, 139).

[GRS05] Denis Gopan, Thomas Reps, and Mooly Sagiv. "A framework for numeric analysis of array operations." In: *ACM SIGPLAN Notices* 40.1 (2005), pp. 338–350. DOI: 10.1145/1047659.1040333 (cited on page 82).

[Gul+08] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori, and Sriram K. Rajamani. "Automatically Refining Abstract Interpretations." In: *Tools and Algorithms for the Construction and Analysis of Systems. 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008, Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Springer. Springer Berlin Heidelberg, 2008, pp. 443–458. DOI: 10.1007/978-3-540-78800-3 (cited on page 118).

[GMT08] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. "Lifting abstract interpreters to quantified logical domains." In: *ACM SIGPLAN Notices*. Vol. 43. 1. ACM. 2008, pp. 235–246. DOI: 10.1145/1328897.1328468 (cited on page 82).

[Gur+15] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. "The SeaHorn Verification Framework." In: *Computer Aided Verification. 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Ed. by Daniel Kroening and Corina S. Păsăreanu. Cham: Springer, 2015, pp. 343–361. DOI: 10.1007/978-3-319-21690-4_20 (cited on pages 3, 126, 132).

[GN21] Arie Gurfinkel and Jorge A. Navas. "Abstract interpretation of LLVM with a region-based memory model." In: *Software Verification. 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18–19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18–19, 2021, Revised Selected Papers*. Springer, 2021, pp. 122–144. DOI: 10.1007/978-3-030-95561-8_8 (cited on pages 21, 83, 130, 166).

[Gut12] Marion Guthmuller. "State equality detection for implementation-level model-checking of distributed applications." In: Aug. 2012, p. 9 (cited on page 143).

[GQC15] Marion Guthmuller, Martin Quinson, and Gabriel Corona. "System-Level State Equality Detection for the Formal Dynamic Verification of Legacy Distributed Applications." In: *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. Mar. 2015, pp. 451–458. DOI: 10.1109/PDP.2015.95 (cited on page 143).

[Had15] Liana Hadarean. "An efficient and trustworthy theory solver for bit-vectors in satisfiability modulo theories." PhD thesis. New York University, 2015 (cited on page 63).

[HP08] Nicolas Halbwachs and Mathias Péron. "Discovering properties about arrays in simple programs." In: *ACM SIGPLAN Notices*. Vol. 43. 6. ACM. 2008, pp. 339–348. DOI: 10.1145/1375581.1375623 (cited on page 82).

[HC12] Raju Halder and Agostino Cortesi. "Abstract interpretation of database query languages." In: *Computer Languages, Systems & Structures* 38.2 (2012), pp. 123–157. DOI: 10.1016/j.cl.2011.10.004 (cited on page 50).

[HSS09] Trevor Hansen, Peter Schachte, and Harald Søndergaard. "State Joining and Splitting for the Symbolic Execution of Binaries." In: *Runtime Verification. 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009, Selected Papers*. Ed. by Saddek Bensalem and Doron A. Peled. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 76–92. DOI: 10.1007/978-3-642-04694-0_6 (cited on page 138).

[HHP13] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. "Software Model Checking for People Who Love Automata." In: *Computer Aided Verification. 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013, Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 36–52. DOI: 10.1007/978-3-642-39799-8_2 (cited on pages 102, 126, 127).

[Hen+04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. "Abstractions from proofs." In: *ACM SIGPLAN Notices*. Vol. 39. 1. ACM. 2004, pp. 232–244. DOI: 10.1145/982962.964021 (cited on page 118).

[Hen+03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. "Software Verification with BLAST." In: *Model Checking Software. 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*. Ed. by Thomas Ball and Sriram K. Rajamani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 235–239. DOI: 10.1007/3-540-44829-2_17 (cited on page 117).

[Hen+02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. "Lazy Abstraction." In: *ACM SIGPLAN Notices* 37 (Mar. 2002). DOI: 10.1145/565816.503279 (cited on pages 3, 116, 117, 128).

[Her30] Jacques Herbrand. "Recherches sur la théorie de la démonstration." In: (1930) (cited on page 61).

[HJV01] Timothy J. Hickey, Qun Ju, and Maarten H. Van Emden. "Interval Arithmetic: From Principles to Implementation." In: *Journal of the ACM* 48.5 (2001), pp. 1038–1068. DOI: 10.1145/502102.502106 (cited on page 60).

[Hoa69] Charles Antony Richard Hoare. "An Axiomatic Basis for Computer Programming." In: *Communications of the ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259 (cited on page 14).

[HB12] Kryštof Hoder and Nikolaj Bjørner. "Generalized property directed reachability." In: *Theory and Applications of Satisfiability Testing – SAT 2012. 15th International Conference, Trento, Italy, June 17-20, 2012, Proceedings*. Springer. 2012, pp. 157–171. DOI: 10.1007/978-3-642-31612-8_13 (cited on page 126).

[Hol+15] Lukáš Holík, Martin Hruška, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar. "Forester: Shape Analysis Using Tree Automata. (Competition Contribution)." In: *Tools and Algorithms for the Construction and Analysis of Systems. 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*. Ed. by Christel Baier and Cesare Tinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 432–435. DOI: 10.1007/978-3-662-46681-0_37 (cited on page 99).

[Hol97] Gerard J. Holzmann. "The model checker SPIN." In: *IEEE Transactions on Software Engineering* 23.5 (1997), pp. 279–295. DOI: 10.1109/32.588521 (cited on page 133).

[Hol02] Gerard J. Holzmann. "UNO: Static Source Code Checking for UserDefined Properties." In: *In 6th World Conf. on Integrated Design and Process Technology, IDPT '02*. 2002 (cited on page 85).

[HP16] Gábor Horváth and Norbert Pataki. "Source Language Representation of Function Summaries in Static Analysis." In: *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS '16. Rome, Italy: Association for Computing Machinery, 2016. DOI: 10.1145/3012408.3012414 (cited on page 137).

[Hsi+97] Cheng-Hsueh A. Hsieh, Marie T. Conte, Teresa L. Johnson, John C. Gyllenhaal, and Wen-mei W. Hwu. "Compilers for improved Java performance." In: *Computer* 30.6 (1997), pp. 67–75. DOI: 10.1109/2.587551 (cited on page 128).

[Hyv+17] Antti Hyvarinen, Sepideh Asadi, Karine Even Mendoza, Grigory Fedyukovich, Hana Chockler, and Natasha Sharygina. "Theory Refinement for Program Verification." In: *Theory and Applications of Satisfiability Testing – SAT 2017. 20th International Conference, Melbourne, VIC, Australia, August 28 – September 1, 2017, Proceedings*. Sept. 2017. DOI: 10.1007/978-3-319-66263-3_22 (cited on pages 116, 123).

[Ibi16] Andreas Ibing. "Dynamic symbolic execution with interpolation based path merging." In: *Int. Conf. Advances and Trends in Software Engineering*. 2016 (cited on page 118).

[Itz+14a] Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Ori Lahav, Aleksandar Nanevski, and Mooly Sagiv. "Modular reasoning about heap paths via effectively propositional formulas." In: *ACM SIGPLAN Notices* 49.1 (2014), pp. 385–396 (cited on page 99).

[Itz+14b] Shachar Itzhaky, Nikolaj Bjorner, Thomas Reps, Mooly Sagiv, and Aditya Thakur. "Property-Directed Shape Analysis." In: *Computer Aided Verification. 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014, Proceedings*. Ed. by Armin Biere and Roderick Bloem. Berlin, Heidelberg: Springer, 2014, pp. 35–51. DOI: 10.1007/978-3-319-08867-9_3 (cited on page 99).

[JMN13] Joxan Jaffar, Vijayaraghavan Murali, and Jorge A. Navas. "Boosting concolic testing via interpolation." In: *ESEC/SIGSOFT FSE. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 2013, pp. 48–58. DOI: 10.1145/2491411.2491425 (cited on page 118).

[JM09] Bertrand Jeannet and Antoine Miné. "Apron: A Library of Numerical Abstract Domains for Static Analysis." In: *Computer Aided Verification. 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009, Proceedings*. Ed. by Ahmed Bouajjani and Oded Maler. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 661–667. DOI: 10.1007/978-3-642-02658-4_52 (cited on pages 126, 132).

[JM06] Ranjit Jhala and Kenneth L. McMillan. "A Practical and Complete Approach to Predicate Refinement." In: *Tools and Algorithms for the Construction and Analysis of Systems. 12th International Conference, TACAS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*. Ed. by Holger Hermanns and Jens Palsberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 459–473. DOI: 10.1007/11691372_33 (cited on page 117).

[JM07] Ranjit Jhala and Kenneth L. McMillan. “Array Abstractions from Proofs.” In: *Computer Aided Verification. 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. Ed. by Werner Damm and Holger Hermanns. Springer. Springer Berlin Heidelberg, 2007, pp. 193–206. DOI: 10.1007/978-3-540-73368-3_23 (cited on page 118).

[JB23] Charu Johns and Jill Britton. *Do not subtract or compare two pointers – SEI CERT C Coding Standard – Confluence*. Apr. 20, 2023. URL: https://wiki.sei.cmu.edu/confluence/display/c/ARR36-C.+Do+not+subtract+or+compare+two+pointers+that+do+not+refer+to+the+same+array (visited on 07/10/2023) (cited on page 103).

[Jon19] Martin Jonáš. “Satisfiability of Quantified Bit-Vector Formulas: Theory and Practice.” PhD thesis. Brno: Masaryk University, Faculty of Informatics, 2019 (cited on page 64).

[JS18] Martin Jonáš and Jan Strejček. “Abstraction of Bit-Vector Operations for BDD-Based SMT Solvers.” In: *Theoretical Aspects of Computing – ICTAC 2018. 15th International Colloquium, Stellenbosch, South Africa, October 16–19, 2018, Proceedings*. Ed. by Bernd Fischer and Tarmo Uustalu. Springer. 2018, pp. 273–291. DOI: 10.1007/978-3-030-02508-3_15 (cited on page 116).

[JMO18] Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. “Modular Static Analysis of String Manipulations in C Programs.” In: *Static Analysis. 25th International Symposium, SAS 2018, Freiburg, Germany, August 29–31, 2018, Proceedings*. 2018, pp. 243–262. DOI: 10.1007/978-3-319-99725-4_16 (cited on page 86).

[JMO19] Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. “An Abstract Domain for Trees with Numeric Relations.” In: *Programming Languages and Systems. 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings*. Ed. by Luís Caires. Cham: Springer, 2019, pp. 724–751. DOI: 10.1007/978-3-030-17184-1_26 (cited on page 99).

[Kal+22] Pankaj Kumar Kalita, Sujit Kumar Muduli, Loris D’Antoni, Thomas Reps, and Subhajit Roy. “Synthesizing Abstract Transformers.” In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2 (2022). DOI: 10.1145/3563334 (cited on page 164).

[KK16] Vini Kanvar and Uday P. Khedker. “Heap Abstractions for Static Analysis.” In: *ACM Computing Surveys* 49.2 (June 2016), 29:1–29:47. DOI: 10.1145/2931098 (cited on pages 98, 99).

[Kar76] Michael Karr. “Affine relationships among variables of a program.” In: *Acta Informatica* 6.2 (June 1976), pp. 133–151. DOI: 10.1007/BF00268497 (cited on page 113).

[KPV03] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. “Generalized Symbolic Execution for Model Checking and Testing.” In: *Tools and Algorithms for the Construction and Analysis of Systems. 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*. Ed. by Hubert Garavel and John Hatcliff. TACAS’03. Warsaw, Poland: Springer Berlin Heidelberg, 2003, pp. 553–568. DOI: 10.1007/3-540-36577-x_40 (cited on page 98).

[Kil73] Gary A. Kildall. “A Unified Approach to Global Program Optimization.” In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’73. Boston, Massachusetts: ACM, 1973, pp. 194–206. DOI: 10.1145/512927.512945 (cited on page 48).

[Kin76] James C. King. “Symbolic Execution and Program Testing.” In: *Communications of the ACM* 19.7 (July 1976), pp. 385–394. DOI: 10.1145/360248.360252 (cited on page 3).

[KS17] Michalis Kokologiannakis and Konstantinos Sagonas. “Stateless model checking of the Linux kernel’s hierarchical read-copy-update (tree RCU).” In: *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. SPIN 2017. Santa Barbara, CA, USA, 2017, pp. 172–181. DOI: 10.1145/3092282.3092287 (cited on page 207).

[KGC14] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. "SMT-Based Model Checking for Recursive Programs." In: *Computer Aided Verification. 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014, Proceedings*. Ed. by Armin Biere and Roderick Bloem. Cham: Springer, 2014, pp. 17–34. DOI: 10.1007/978-3-319-08867-9_2 (cited on page 126).

[Kor+20] Lukáš Korenčik, Petr Ročkai, Henrich Lauko, and Jiří Barnat. "On Symbolic Execution of Decompiled Programs." In: *Proceedings – 2020 IEEE 20th International Conference on Software Quality, Reliability, and Security, QRS 2020*. Neuveden: IEEE Computer Society, 2020, pp. 265–272. DOI: 10.1109/QRS51102.2020.00044 (cited on pages 9, 142, 183).

[KV09] Laura Kovács and Andrei Voronkov. "Finding loop invariants for programs over arrays using a theorem prover." In: *Fundamental Approaches to Software Engineering. 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009, Proceedings*. Springer. 2009, pp. 470–485. DOI: 10.1007/978-3-642-00593-0_33 (cited on page 82).

[Kri07] Saul Kripke. "Semantical considerations of the modal logic." In: *Studia Philosophica* 1 (2007) (cited on page 38).

[KS16] Daniel Kroening and Ofer Strichman. "Equality Logic and Uninterpreted Functions." In: *Decision Procedures: An Algorithmic Point of View*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 77–95. DOI: 10.1007/978-3-662-50497-0_4 (cited on page 123).

[KT14] Daniel Kroening and Michael Tautschnig. "CBMC–C bounded model checker. (Competition Contribution)." In: *Tools and Algorithms for the Construction and Analysis of Systems. 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by Erika Ábrahám and Klaus Havelund. Springer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 389–391. DOI: 10.1007/978-3-642-54862-8_26 (cited on pages 102, 139).

[Kuz+12] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. "Efficient State Merging in Symbolic Execution." In: *SIGPLAN Not.* 47.6 (2012), pp. 193–204. DOI: 10.1145/2345156.2254088 (cited on page 138).

[Laa14] Alfons Laarman. "Scalable multi-core model checking." IPA Dissertation Series No. 2014-06. PhD thesis. Netherlands: University of Twente, May 2014 (cited on page 174).

[LJG11] Lies Lakhdar-Chaouch, Bertrand Jeannet, and Alain Girault. "Widening with Thresholds for Programs with Complex Control Graphs." In: *Automated Technology for Verification and Analysis. 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011, Proceedings*. Ed. by Tevfik Bultan and Pao-Ann Hsiung. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 492–502. DOI: 10.1007/978-3-642-24372-1_38 (cited on page 48).

[LA04] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, p. 75 (cited on pages 21, 126).

[Lau20] Henrich Lauko. *Abstractions via Program Transformations*. Brno, 2020. URL: https://is.muni.cz/th/uqf69/. (PhD Thesis Proposal) (cited on pages 21, 125).

[LKR22] Henrich Lauko, Lukáš Korenčik, and Petr Ročkai. "Verification of Programs Sensitive to Heap Layout." In: *ACM Transactions on Software Engineering and Methodology* 31 (4 2022). DOI: 10.1145/3508363 (cited on pages 8, 21, 97, 182, 207, 209).

[Lau+20] Henrich Lauko, Martina Olliaro, Agostino Cortesi, and Petr Ročkai. "Abstracting Strings for Model Checking of C Programs." In: *Applied Sciences. Special Issue Static Analysis Techniques: Recent Advances and New Horizons* 10.21 (2020). DOI: 10.3390/app10217853 (cited on pages 8, 77, 83, 86, 88, 90, 91, 93, 177, 202).

[LR22] Henrich Lauko and Petr Ročkai. “LART: Compiled Abstract Execution. (Competition Contribution).” In: *Tools and Algorithms for the Construction and Analysis of Systems. 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*. Munich, Germany: Springer, 2022, pp. 457–461. DOI: 10.1007/978-3-030-99527-0_31 (cited on pages 8, 198).

[LRB18] Henrich Lauko, Petr Ročkai, and Jiří Barnat. “Symbolic Computation via Program Transformation.” In: *Theoretical Aspects of Computing – ICTAC 2018. 15th International Colloquium, Stellenbosch, South Africa, October 16-19, 2018, Proceedings*. Ed. by Bernd Fischer and Tarmo Uustalu. Cham: Springer, 2018, pp. 313–332. DOI: 10.1007/978-3-030-02508-3_17 (cited on pages 7, 47, 95, 125, 132, 133, 176, 193, 195).

[Lau+ng] Henrich Lauko, Petr Ročkai, Vladimír Štill, and Jiří Barnat. “DIVINE – Model Checker for C++.” In: *Automatic Software Verification*. Ed. by Dirk Beyer. (forthcoming) (cited on pages 8, 21, 47, 125).

[Lau+19] Henrich Lauko, Vladimír Štill, Petr Ročkai, and Jiří Barnat. “Extending DIVINE with Symbolic Verification Using SMT. (Competition Contribution).” In: *Tools and Algorithms for the Construction and Analysis of Systems. 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part III*. Cham: Springer, 2019, pp. 204–208. DOI: 10.1007/978-3-030-17502-3_14 (cited on pages 8, 47, 176, 197).

[LSA16] Wonyeol Lee, Rahul Sharma, and Alex Aiken. “Verifying Bit-Manipulations of Floating-Point.” In: *ACM SIGPLAN Notices* 51.6 (2016), pp. 70–84. DOI: 10.1145/2980983.2908107 (cited on page 49).

[Lei08] K. Rustan M. Leino. “This is Boogie 2.” June 2008 (cited on page 21).

[Li+13] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. “Steering symbolic execution to less traveled paths.” In: *ACM SigPlan Notices* 48.10 (2013), pp. 19–32. DOI: 10.1145/2544173.2509553 (cited on page 157).

[MA14] Magnus Madsen and Esben Andreasen. “String Analysis for Dynamic Field Access.” In: *Compiler Construction. 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5–13, 2014, Proceedings*. 2014, pp. 197–217. DOI: 10.1007/978-3-642-54807-9_12 (cited on page 86).

[Mag+10] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. “Automatic numeric abstractions for heap-manipulating programs.” In: *ACM Sigplan Notices*. Vol. 45. 1. ACM. 2010, pp. 211–222. DOI: 10.1145/1707801.1706326 (cited on page 99).

[MR18] Rupak Majumdar and Jean-Francois Raskin. “Symbolic Model Checking in Non-Boolean Domains.” In: *Handbook of Model Checking*. Springer, Jan. 1, 2018. DOI: 10.1007/978-3-319-10575-8_31 (cited on page 3).

[Mal+18] Vikor Malík, Martin Hruška, Peter Schrammel, and Tomáš Vojnar. “Template-Based Verification of Heap-Manipulating Programs.” In: *2018 Formal Methods in Computer Aided Design (FMCAD)*. Oct. 2018, pp. 1–9. DOI: 10.23919/fmcad.2018.8603009 (cited on pages 99, 102).

[MZ17] Isabella Mastroeni and Damiano Zanardini. “Abstract Program Slicing: An Abstract Interpretation-Based Approach to Program Slicing.” In: *ACM Trans. Comput. Logic* 18.1 (2017). DOI: 10.1145/3029052 (cited on page 39).

[Mau00] Laurent Mauborgne. “An incremental unique representation for regular trees.” In: *Nordic Journal of Computing* 7.4 (2000), pp. 290–311 (cited on page 77).

[MR05] Laurent Mauborgne and Xavier Rival. “Trace Partitioning in Abstract Interpretation Based Static Analyzers.” In: *Proceedings of the 14th European Conference on Programming Languages and Systems*. ESOP’05. Edinburgh, UK: Springer, 2005, pp. 5–20. DOI: 10.1007/978-3-540-31987-0_2 (cited on page 132).

[MJ93] Steven McCanne and Van Jacobson. “The BSD Packet Filter: A New Architecture for User-Level Packet Capture.” In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX’93. San Diego, California: USENIX Association, 1993, p. 2 (cited on page 50).

[McM93] Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993 (cited on page 14).

[McM03] Kenneth L. McMillan. “Interpolation and SAT-based model checking.” In: *Computer Aided Verification. 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*. Ed. by Warren A. Hunt and Fabio Somenzi. Springer. Springer Berlin Heidelberg, 2003, pp. 1–13. DOI: 10.1007/978-3-540-45069-6_1 (cited on page 117).

[McM06] Kenneth L. McMillan. “Lazy Abstraction with Interpolants.” In: *Computer Aided Verification. 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Ed. by Thomas Ball and Robert B. Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 123–136. DOI: 10.1007/11817963_14 (cited on pages 117, 126).

[Mey92] Bertrand Meyer. “Applying’design by contract’.” In: *Computer* 25.10 (1992), pp. 40–51. DOI: 10.1109/2.161279 (cited on page 138).

[Min04] Antoine Miné. “Weakly relational numerical abstract domains.” PhD thesis. Ecole Polytechnique X, 2004 (cited on pages 18, 33, 139).

[Min06a] Antoine Miné. “Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics.” In: *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES’06), Ottawa, Ontario, Canada, June 14-16, 2006*. 2006, pp. 54–63. DOI: 10.1145/1134650.1134659 (cited on page 86).

[Min06b] Antoine Miné. “Symbolic Methods to Enhance the Precision of Numerical Abstract Domains.” In: *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation. 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*. Ed. by E. Allen Emerson and Kedar S. Namjoshi. VMCAI’06. Charleston, SC: Springer, 2006, pp. 348–363. DOI: 10.1007/11609773_23 (cited on page 69).

[Min06c] Antoine Miné. “The octagon abstract domain.” In: *Higher-Order and Symbolic Computation* 19.1 (Mar. 1, 2006), p. 31. DOI: 10.1007/s10990-006-8609-1 (cited on pages 68, 113, 118).

[Min12] Antoine Miné. “Abstract domains for bit-level machine integer and floating-point operations.” In: *WING’12-4th International Workshop on invariant Generation*. 2012, p. 16 (cited on page 49).

[Mor+14] Jeremy Morse, Mikhail Ramalho, Lucas Cordeiro, Denis Nicole, and Bernd Fischer. “ESBMC 1.22. (Competition Contribution).” In: *Tools and Algorithms for the Construction and Analysis of Systems. 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by Erika Ábrahám and Klaus Havelund. Springer. Springer Berlin Heidelberg, 2014, pp. 405–407. DOI: 10.1007/978-3-642-54862-8_31 (cited on page 207).

[MU21] Leonardo de Moura and Sebastian Ullrich. “The Lean 4 Theorem Prover and Programming Language.” In: *Automated Deduction – CADE 28. 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Ed. by André Platzer and Geoff Sutcliffe. Cham: Springer, 2021, pp. 625–635. DOI: 10.1007/978-3-030-79876-5_37 (cited on page 14).

[Mrá+16] Jan Mrázek, Petr Bauch, Henrich Lauko, and Jiří Barnat. “SymDIVINE: Tool for Control-Explicit Data-Symbolic State Space Exploration.” In: *Model Checking Software. 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7-8, 2016, Proceedings*. Ed. by Dragan Bošnački and Anton Wijs. Neuveden: Springer, 2016, pp. 208–213. DOI: 10.1007/978-3-319-32582-8_14 (cited on pages 9, 53, 65, 142, 144, 195).

[Mrá+17] Jan Mrázek, Martin Jonáš, Vladimír Štill, Henrich Lauko, and Jiří Barnat. "Optimizing and Caching SMT Queries in SymDIVINE. (Competition Contribution)." In: *Tools and Algorithms for the Construction and Analysis of Systems. 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*. Ed. by Tiziana Margaria Axel Legay. Berlin, Heidelberg: Springer, 2017, pp. 390–393. DOI: 10.1007/978-3-662-54580-5_29 (cited on page 9).

[MJ81] Steven S. Muchnick and Neil D. Jones. *Program Flow Analysis: Theory and Application*. Prentice Hall Professional Technical Reference, 1981 (cited on pages 68, 118).

[MV14] Petr Müller and Tomás Vojnar. "CPAlien: Shape Analyzer for CPAChecker." In: *Tools and Algorithms for the Construction and Analysis of Systems. 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5–13, 2014, Proceedings*. 2014, pp. 395–397. DOI: 10.1007/978-3-642-54862-8_28 (cited on page 126).

[Nav+12] Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. "Signedness-Agnostic Program Analysis: Precise Integer Bounds for Low-Level Code." In: *Programming Languages and Systems. 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012, Proceedings*. Ed. by Ranjit Jhala and Atsushi Igarashi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 115–130. DOI: 10.1007/978-3-642-35182-2_9 (cited on page 59).

[Nec+02] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. "CIL: Intermediate language and tools for analysis and transformation of C programs." In: *International Conference on Compiler Construction*. Springer. 2002, pp. 213–228. DOI: 10.1007/3-540-45937-5_16 (cited on page 21).

[NS07] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation." In: *SIGPLAN Not.* 42.6 (2007), pp. 89–100. DOI: 10.1145/1273442.1250746 (cited on pages 5, 140).

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media, 2002 (cited on page 14).

[One96] Aleph One. *Smashing The Stack For Fun And Profit*. Phrack Magazine. 1996 (cited on page 85).

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. "PVS: A prototype verification system." In: *Automated Deduction—CADE-11*. Ed. by Deepak Kapur. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 748–752. DOI: 10.1007/3-540-55602-8_217 (cited on page 14).

[PIR16] Changhee Park, Hyeonseung Im, and Sukyoung Ryu. "Precise and Scalable Static Analysis of jQuery Using a Regular Expression Domain." In: *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*. 2016, pp. 25–36. DOI: 10.1145/2989225.2989228 (cited on page 86).

[PS18] Etienne Payet and Fausto Spoto. "Checking Array Bounds by Abstract Interpretation and Symbolic Expressions." In: *Automated Reasoning. 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*. Ed. by Didier Galmiche, Stephan Schulz, and Roberto Sebastiani. Springer, Jan. 1, 2018. DOI: 10.1007/978-3-319-94205-6_46 (cited on page 82).

[Pel98] Doron Peled. "Ten Years of Partial Order Reduction." In: *Computer Aided Verification. 10th International Conference, CAV'98, Vancouver, BC, Canada, June 28-July 2, 1998, Proceedings*. Ed. by Alan J. Hu and Moshe Y. Vardi. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 17–28. DOI: 10.1007/bfb0028727 (cited on page 174).

[PWZ13] Ruzica Piskac, Thomas Wies, and Damien Zufferey. "Automating separation logic using SMT." In: *Computer Aided Verification. 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013, Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Springer. Springer Berlin Heidelberg, 2013, pp. 773–789. DOI: 10.1007/978-3-642-39799-8_54 (cited on page 99).

[Plo71] Gordon D. Plotkin. "A further note on inductive generalization." In: *Machine intelligence* 6 (1971), pp. 101–124 (cited on page 61).

[PF20] Sebastian Poeplau and Aurélien Francillon. “Symbolic execution with SymCC: Don’t interpret, compile!” In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 181–198 (cited on pages 3, 5, 46, 133, 140, 141).

[PF21] Sebastian Poeplau and Aurelien Francillon. “SymQEMU: Compilation-based symbolic execution for binaries.” In: *Network and Distributed System Security Symposium*. Network & Distributed System Security Symposium. Feb. 2021. DOI: 10.14722/ndss.2021.24118 (cited on pages 5, 133, 141).

[PGM04] Sylvie Putot, Eric Goubault, and Matthieu Martel. “Static Analysis-Based Validation of Floating-Point Computations.” In: *Numerical Software with Result Verification*. Ed. by René Alt, Andreas Frommer, R. Baker Kearfott, and Wolfram Luther. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 306–313. DOI: 10.1007/978-3-540-24738-8_18 (cited on page 50).

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. “Precise interprocedural dataflow analysis via graph reachability.” In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1995, pp. 49–61. DOI: 10.1145/199448.199462 (cited on page 138).

[Rep+10] Thomas Reps, Junghee Lim, Aditya Thakur, Gogul Balakrishnan, and Akash Lal. “There’s plenty of room at the bottom: Analyzing and verifying machine code.” In: *Computer Aided Verification. 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010, Proceedings*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Springer. Springer Berlin Heidelberg, 2010, pp. 41–56. DOI: 10.1007/978-3-642-14295-6_6 (cited on page 50).

[RSW02] Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. “Shape analysis and applications.” In: *The Compiler Design Handbook*. CRC Press, 2002, pp. 179–221 (cited on page 77).

[Rey70] John C. Reynolds. “Transformational systems and algebraic structure of atomic formulas.” In: *Machine intelligence* 5 (1970), pp. 135–151 (cited on page 61).

[Ric53] Henry Gordon Rice. “Classes of recursively enumerable sets and their decision problems.” In: *Transactions of the American Mathematical Society* 74 (1953), pp. 358–366 (cited on page 14).

[Roč15] Petr Ročkai. “Model Checking Software.” PhD thesis. Brno: Masaryk University, Faculty of Informatics, 2015. (Visited on 09/15/2022) (cited on page 2).

[Roč+21] Petr Ročkai, Zuzana Baranová, Jan Mrázek, Katarína Kejstová, and Jiří Barnat. “Reproducible Execution of POSIX Programs with DiOS.” In: *Software and Systems Modeling* 20.2 (2021), pp. 363–382. DOI: 10.1007/s10270-020-00837-y (cited on page 175).

[Roč+19] Petr Ročkai, Henrich Lauko, Martina Olliaro, and Agostino Cortesi. “String Abstraction for Model Checking of C Programs.” In: *Model Checking Software. 26th International Symposium, SPIN 2019, Beijing, China, July 15–16, 2019, Proceedings*. Ed. by Axel Legay Fabrizio Biondi Thomas Given-Wilson. Cham: Springer, 2019, pp. 74–93. DOI: 10.1007/978-3-030-30923-7_5 (cited on pages 8, 77, 86, 88, 177, 202).

[Roč+18] Petr Ročkai, Vladimír Štill, Ivana Černá, and Jiří Barnat. “DiVM: Model checking with LLVM and graph memory.” In: *Journal of Systems and Software* 143 (2018), pp. 1–13. DOI: 10.1016/j.jss.2018.04.026 (cited on pages 23, 143).

[Rua+21] Nicola Ruaro, Kyle Zeng, Lukas Dresel, Mario Polino, Tiffany Bao, Andrea Continella, Stefano Zanero, Christopher Kruegel, and Giovanni Vigna. “SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning.” In: *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID ’21. San Sebastian, Spain: Association for Computing Machinery, 2021, pp. 456–468. DOI: 10.1145/3471621.3471865 (cited on page 157).

[Šár22] Jakub Šárník. “Automatická analýza neúplných programů.” MA thesis. Brno: Masaryk University, Faculty of Informatics, 2022 (cited on page 185).

[SHB16] Dominic Scheurer, Reiner Hähnle, and Richard Bubel. “A General Lattice Model for Merging Symbolic Execution Branches.” In: *Formal Methods and Software Engineering. 18th International Conference on Formal Engineering Methods, ICFEM 2016, Tokyo, Japan, November 14–18, 2016, Proceedings*. Ed. by Kazuhiro Ogata, Mark Lawford, and Shaoying Liu. Cham: Springer, 2016, pp. 57–73. DOI: 10.1007/978-3-319-47846-3_5 (cited on page 138).

[Sch97] David A. Schmidt. "Abstract interpretation in the operational semantics hierarchy." In: *BRICS Report Series* 4.2 (1997). DOI: 10.7146/brics.v4i2.18781 (cited on page 27).

[Sch53] Jürgen Schmidt. "Beiträge zur Filtertheorie. II." In: *Mathematische Nachrichten* 10.3-4 (1953), pp. 197–232 (cited on page 16).

[SK16] Peter Schrammel and Daniel Kroening. "2LS for Program Analysis. (Competition Contribution)." In: *Tools and Algorithms for the Construction and Analysis of Systems. 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Marsha Chechik and Jean-François Raskin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 905–907. DOI: 10.1007/978-3-662-49674-9_56 (cited on pages 102, 207).

[Sea13] Robert C. Seacord. *Secure Coding in C and C++*. 2nd. Addison-Wesley Professional, 2013 (cited on page 89).

[SPW09] Mohamed Nassim Seghir, Andreas Podelski, and Thomas Wies. "Abstraction refinement for quantified array assertions." In: *Static Analysis. 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009, Proceedings*. Springer. 2009, pp. 3–18. DOI: 10.1007/978-3-642-03237-0_3 (cited on page 82).

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C." In: *ACM SIGSOFT Software Engineering Notes* 30.5 (Sept. 2005), pp. 263–272. DOI: 10.1145/1095430.1081750 (cited on pages 97, 132).

[Sen+15] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. "MultiSE: Multi-Path Symbolic Execution Using Value Summaries." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 842–853. DOI: 10.1145/2786805.2786830 (cited on page 138).

[SZ10] Hossain Shahriar and Mohammad Zulkernine. "Classification of Static Analysis-Based Buffer Overflow Detectors." In: *Fourth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2010, Singapore, June 9-11, 2010 - Companion Volume*. 2010, pp. 94–101. DOI: 10.1109/SSIRI-C.2010.28 (cited on page 85).

[Sha17] Tushar Sharma. *Abstract Interpretation Over Bitvectors*. The University of Wisconsin-Madison, 2017 (cited on page 60).

[SR17] Tushar Sharma and Thomas Reps. "Sound Bit-Precise Numerical Domains." In: *Verification, Model Checking, and Abstract Interpretation. 18th International Conference, VMCAI 2017, Paris, France, January 15–17, 2017, Proceedings*. Ed. by Ahmed Bouajjani and David Monniaux. Cham: Springer, 2017, pp. 500–520. DOI: 10.1007/978-3-319-52234-0_27 (cited on pages 49, 50).

[STR13] Tushar Sharma, Aditya Thakur, and Thomas Reps. *An abstract domain for bit-vector inequalities*. Tech. rep. Computer Science Department, University of Wisconsin, Madison, 2013 (cited on page 50).

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. "Checking safety properties using induction and a SAT-solver." In: *Formal Methods in Computer-Aided Design. Third International Conference, FMCAD 2000 Austin, TX, USA, November 1-3, 2000 Proceedings*. Springer. 2000, pp. 127–144. DOI: 10.1007/3-540-40922-x_8 (cited on page 126).

[Sho+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis." In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 138–157. DOI: 10.1109/SP.2016.17 (cited on pages 132, 140).

[SPV15] Gagandeep Singh, Markus Püschel, and Martin Vechev. "Making numerical program analysis fast." In: *ACM SIGPLAN Notices* 50.6 (2015), pp. 303–313. DOI: 10.1145/2813885.2738000 (cited on pages 126, 132).

[SPV17a] Gagandeep Singh, Markus Püschel, and Martin Vechev. “A practical construction for decomposing numerical abstract domains.” In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–28. DOI: 10.1145/3158143 (cited on page 132).

[SPV17b] Gagandeep Singh, Markus Püschel, and Martin Vechev. “Fast polyhedra abstract domain.” In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 2017, pp. 46–59. DOI: 10.1145/3009837.3009885 (cited on page 132).

[SJ11] Pascal Sotin and Bertrand Jeannet. “Precise Interprocedural Analysis in the Presence of Pointers to the Stack.” In: *Programming Languages and Systems. 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011, Proceedings*. Ed. by Gilles Barthe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 459–479. DOI: 10.1007/978-3-642-19718-5_24 (cited on page 49).

[SD07] Jean Souyris and David Delmas. “Experimental Assessment of Astrée on Safety-Critical Avionics Software.” In: vol. 4680. Sept. 2007, pp. 479–490. DOI: 10.1007/978-3-540-75101-4_45 (cited on page 132).

[Sta03] Richard M. Stallman. “The GCC developer community.” In: *Using GCC: The GNU Compiler Collection Reference Manual* (2003) (cited on page 103).

[Ste95] Bjarne Steensgaard. “Sparse functional stores for imperative programs.” In: *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*. 1995, pp. 62–70. DOI: 10.1145/202529.202536 (cited on page 40).

[Ste20] Dominic Steinhöfel. “Abstract Execution: Automatically Proving Infinitely Many Programs.” PhD thesis. Darmstadt: Technische Universität, 2020. DOI: 10.25534/tuprints-00008540 (cited on page 138).

[Ste22] Dominic Steinhöfel. “Symbolic Execution: Foundations, Techniques, Applications, and Future Perspectives.” In: *The Logic of Software. A Tasting Menu of Formal Methods. Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*. Ed. by Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, and Einar Broch Johnsen. Cham: Springer, 2022, pp. 446–480. DOI: 10.1007/978-3-031-08166-8_22 (cited on pages 135, 136).

[SS15] Evgeniy Stepanov and Konstantin Serebryany. “MemorySanitizer: fast detector of uninitialized memory use in C++.” In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society. 2015, pp. 46–55 (cited on page 129).

[ŠRB16] Vladimír Štill, Petr Ročkai, and Jiří Barnat. “Weak Memory Models as LLVM-to-LLVM Transformations.” In: *Mathematical and Engineering Methods in Computer Science. 10th International Doctoral Workshop, MEMICS 2015, Telč, Czech Republic, October 23-25, 2015, Revised Selected Papers*. Ed. by Jan Kofroň and Tomáš Vojnar. Neuveden: Springer, 2016, pp. 144–155. DOI: 10.1007/978-3-319-29817-7_13 (cited on page 186).

[SX18] Yulei Sui and Jingling Xue. “Value-flow-based demand-driven pointer analysis for C and C++.” In: *IEEE Transactions on Software Engineering* 46.8 (2018), pp. 812–835. DOI: 10.1109/tse.2018.2869336 (cited on page 45).

[SA13] Josef Svenningsson and Emil Axelsson. “Combining Deep and Shallow Embedding for EDSL.” In: *Trends in Functional Programming. 13th International Symposium, TFP 2012, St Andrews, UK, June 12-14, 2012, Revised Selected Papers*. Ed. by Hans-Wolfgang Loidl and Ricardo Peña. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 21–36. DOI: 10.1007/978-3-642-40447-4_2 (cited on page 135).

[TCR13] Antoine Toubhans, Bor-Yuh Evan Chang, and Xavier Rival. “Reduced Product Combination of Abstract Domains for Shapes.” In: *Verification, Model Checking, and Abstract Interpretation. 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013, Proceedings*. Ed. by Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 375–395. DOI: 10.1007/978-3-642-35873-9_23 (cited on pages 59, 67, 99, 119).

[Tur49] Alan Turing. “Checking a large routine.” In: *Report of a Conference on High Speed Automatic Calculating Machines*. University Mathematical Laboratory, 1949, pp. 67–69 (cited on page 14).

[UM14] Caterina Urban and Antoine Miné. “A Decision Tree Abstract Domain for Proving Conditional Termination.” In: *Static Analysis. 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*. Ed. by Markus Müller-Olm and Helmut Seidl. Cham: Springer, 2014, pp. 302–318. DOI: 10.1007/978-3-319-10936-7_19 (cited on pages 68, 118).

[VW86] Moshe Y. Vardi and Pierre Wolper. “An automata-theoretic approach to automatic program verification.” In: *1st Symposium in Logic in Computer Science (LICS)*. IEEE Computer Society. 1986 (cited on page 38).

[Vis+21] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. “Semantics, Verification, and Efficient Implementations for Tristate Numbers.” In: *CoRR* abs/2105.05398 (2021) (cited on page 50).

[Wag+00] David A. Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. “A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities.” In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2000, San Diego, California, USA*. 2000 (cited on page 85).

[WKP09] Martin Wehrle, Sebastian Kupferschmid, and Andreas Podelski. “Transition-based directed model checking.” In: *Tools and Algorithms for the Construction and Analysis of Systems. 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009, Proceedings*. Ed. by Stefan Kowalewski and Anna Philippou. Springer. Springer Berlin Heidelberg, 2009, pp. 186–200. DOI: 10.1007/978-3-642-00768-2_19 (cited on page 38).

[Wie83] Clark Wiedmann. “A Performance Comparison between an APL Interpreter and Compiler.” In: *SIGAPL APL Quote Quad* 13.3 (1983), pp. 211–217. DOI: 10.1145/390005.801219 (cited on page 128).

[XCE03] Yichen Xie, Andy Chou, and Dawson R. Engler. “ARCHER: using symbolic, path-sensitive analysis to detect memory access errors.” In: *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003*. 2003, pp. 327–336. DOI: 10.1145/940071.940115 (cited on page 85).

[Yan+08] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O’Hearn. “Scalable Shape Analysis for Systems Code.” In: *Computer Aided Verification. 20th International Conference, CAV 2008 Princeton, NJ, USA, July 7-14, 2008, Proceedings*. Ed. by Aarti Gupta and Sharad Malik. Springer Berlin Heidelberg, Jan. 1, 2008. DOI: 10.1007/978-3-540-70545-1_36 (cited on pages 3, 99, 208).

[Yao+21] Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. “Program Analysis via Efficient Symbolic Abstraction.” In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA (2021). DOI: 10.1145/3485495 (cited on page 145).

[Yun+18] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. “QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing.” In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761 (cited on pages 132, 140).