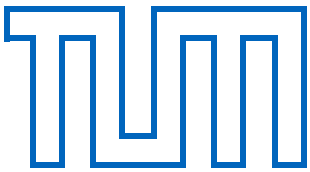


TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Adaptable Static Analysis of Executables for proving the Absence of Vulnerabilities

Dipl.-Inf. Univ. Bogdan Andrei Mihaila

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:

Univ.-Prof. Tobias Nipkow, Ph.D.

Prüfer der Dissertation:

1. TUM Junior Fellow Dr. Axel Simon
2. Univ.-Prof. Dr. Javier Esparza
3. Prof. David Pichardie, École normale supérieure de Rennes, Frankreich

Die Dissertation wurde am 13.11.2014 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 08.01.2015 angenommen.

Acknowledgements

I would like to thank my colleague Alexander Sepp for implementing the initial huge part of the framework and thus making this work possible; Holger Siegel for his contributions to the analysis framework and the implementation of the memory handling parts of the framework; Julian Kranz for his work on the disassembler frontend and GDSL; all the students that contributed code to the framework; my advisor Axel Simon as the initiator and head of the project for his guidance and support throughout my PhD; Helmut Seidl for the motivation and support he has given me after my graduation and my work colleagues for the recreational foosball matches, the valuable and not so valuable conversations.

I would like to thank Thomas Dullien for initiating the idea of a dynamically started static analysis, for interesting discussions and for providing us challenging analysis examples.

The biggest Thank You goes to my family and friends for the support throughout stressful times and the fun during good times.

Abstract

In a world where vulnerabilities in software pose an inherent threat for the networked society, analyzing third-party software, in the form of executable programs, becomes increasingly important. While program verification at the source code level has received much attention in the last decades, this thesis focuses on providing a sound and precise analysis framework for binaries, thereby enabling the understanding, auditing, and verification of executable programs.

The first challenge in analyzing machine code in binaries is to give a semantics to the many instructions of a modern processor. This semantics should be faithful to the computation of the concrete machine semantics in order to allow for a range of analyses. Moreover, the semantics should be concise to enable scalable and easy-to-implement analyses. To this end, we present the intermediate language RREIL and a corresponding analysis framework that takes RREIL programs as input, thereby being mostly agnostic to the actual architecture that the binary was compiled for.

Due to over-approximation, any analysis may warn about invalid memory accesses that cannot occur in the actual program; a so-called false positive. Our framework is therefore adaptable by featuring a plug-in architecture where code auditors can integrate special-purpose domains specific to their analysis needs. Specifically, we propose a hierarchy of three interfaces to abstract domains, namely for inferring memory layout, bit-level information and numeric information. We equip this framework with well-known abstract domains that reason about buffer overflows, pointer dereferences and possible integer wrap-arounds in arithmetic operations. We demonstrate the extensibility of our framework by proposing a set of novel abstract domains geared towards the precise analysis of binaries.
In particular, we introduce a set of abstract domains that improve the precision of widening, a technique to ensure termination and scalability in numeric analyses. In the context of machine code analysis, widening faces challenges that cannot be addressed by techniques currently used in source code analysis.
Furthermore, we introduce a novel domain that improves the precision of off-the-shelf convex numeric domains. This novel domain uses techniques from predicate abstraction, thus allowing to express complex, non-convex program invariants.
A third novel domain we introduce allows for combining initialized and non-initialized values in relational numeric analyses without undue loss of precision. This domain is

essential when representing program states where a memory region (a stack frame or heap cell) may or may not exist.

No matter how precise an analysis is, a code auditor may have to deal with a large number of warnings when analyzing large programs. In order to improve the usability of the analyzer, we detail how combining static and dynamic analysis allows an auditor to focus the analyzer on potentially vulnerable code parts.

Finally, we illustrate the analysis of particularly difficult examples. Specifically, we present the analysis of a subtle bug in Sendmail, where our analyzer is able to discover the fault in the vulnerable program version and verifies the corrected variant.

Zusammenfassung

In einer Welt in der Sicherheitslücken in Software eine latente Bedrohung der vernetzten Gesellschaft darstellt, wird die Analyse von fremder Software, in der Form von ausführbaren Programmen, immer wichtiger. Während die Verifikation von Quellcode in den letzten Jahrzehnten viel Aufmerksamkeit auf sich gezogen hat, ist das Ziel dieser Arbeit eine Infrastruktur für korrekte Analysen von Binärdateien bereitzustellen, um dadurch das Verstehen und die Verifikation von ausführbaren Programmen zu ermöglichen.

Die erste Herausforderung in der Analyse von Maschinencode ist es, Binärdateien eine Semantik zu geben, die die vielen Instruktionen moderner Prozessoren umfasst. Diese Semantik sollte sinngetreu die Berechnung der Instruktionen wiedergeben, um verschiedene Analysen zu ermöglichen. Weiterhin sollte die Semantik kompakt sein um skalierende und einfach zu implementierende Analysen zu ermöglichen. Wir präsentieren dafür die Zwischensprache RREIL und eine entsprechende Analyseinfrastruktur, die RREIL als Eingabe nimmt, sodass die Analyse unabhänging von der Architektur, für die die Binärdatei übersetzt wurde, arbeiten kann.

Aufgrund der Überapproximation kann eine Analyse über inkorrekte Speicherzugriffe warnen, obwohl diese im Programm gar nicht auftreten können; sogenannte Fehlalarme. Unsere Infrastruktur kann daher angepasst werden durch das Hinzufügen von speziellen Domänen, die auf das zu analysierende Programm zugeschnitten sind. Ganz konkret präsentieren wir eine Hierarchie von drei Schnittstellen, jeweils für die Inferenz von Variablenanordnungen im Speicher, für die Manipulation von Werten auf der Bit-Ebene und für die Manipulation von numerischer Information. Wir statten diese Infrastruktur mit etablierten abstrakten Domänen aus, die Speicherüberläufe, Zeiger und arithmetische Überläufe analysieren.
Die Erweiterbarkeit zeigen wir, indem wir eine neue abstrakte Domäne präsentieren, die präzises *widening* für Binärcode implementiert, was sicherstellt, dass numerische Analysen terminieren und skalieren. Im Rahmen der Analyse von Maschinencode ist *widening* insofern heikel, dass bestehende Techniken aus der Analyse von Quellcode nicht ausreichen.
Weiterhin schlagen wir eine neue Domäne vor, die die Präzision von etablierten numerischen Domänen verbessert. Diese neue Domäne benutzt Techniken aus dem Gebiet der Prädikatenabstraktion und kann komplexe, nicht-konvexe Zustandsräume auszudrücken.
Die dritte neue Domäne die wir präsentieren erlaubt die Kombination von initialisierten

und uninitialisierten Werten in relationalen numerischen Analysen ohne übermäßigen Präzisionsverlust. Diese Domäne ist essentiell um Programmzustände darzustellen, in denen ein Speicherbereich (ein Stapel oder Speicher auf der Halde) nicht unbedingt existiert.

Unabhängig davon, wie präzise eine Analyse ist, ein Nutzer der Analyse muss mit sehr vielen potentiellen Fehlern umgehen sobald Programme größer werden. Um die Benutzbarkeit der Analyse zu verbessern, zeigen wir, wie die Kombination von statischer und dynamischer Analyse dem Benutzer helfen kann, den Fokus der Analyse auf Bereiche mit potentiellen Sicherheitslücken zu beschränken.

Zu guter Letzt veranschaulichen wir die Infrastruktur, indem wir die Analyse herausfordernder Beispiele diskutieren. Hierzu nutzen wir einen bekannten Fehler in Sendmail. Unsere Analyse identifiziert diesen Fehler in der Originalversion und verifiziert die korrigierte Version als korrekt.

Contents

Part I

Static Analysis of Binary Code

1 Introduction

The need for software analysis is becoming more relevant as software is nowadays part of every technical product. Especially in security critical areas, such as aviation and the automobile industry the analysis of software to prove the absence of errors is part of the software development process. While many tools exist for source code analysis, the analysis of executables has only recently become the focus of attention, mostly due to prominent discoveries of vulnerabilities in widely deployed software.

1.1 Necessity of Machine Code Analysis

Despite the growing need to eliminate faults to prevent security breaches, security auditing of commercial software often has to rely on the inspection of the machine code of the software. Most vendors employ a closed-source policy, thereby forcing customers using their libraries to either trust the code or to perform an audit on the machine code level. However, the manual auditing of machine code is a daunting task, as it requires the reverse engineering and understanding of millions of machine code instructions. Given the ever growing size of software projects and the ubiquity of software in security critical devices, reverse engineering and security auditing requires automatic program analysis tools to support the security engineer.

A second major challenge for a security auditor is the analysis of malware, that is, malicious code that is designed to hide its intent. Malware often tries to resist reverse engineering by using code obfuscation techniques [CN09]. Most of these obfuscations are targeted at the human auditor, e.g. control flow obfuscation techniques try to thwart code understanding. An automated analysis tool may cope well with such rather simple obfuscations, thereby aiding reverse engineers with the task of disassembling and understanding of malicious code.

Besides helping with reverse engineering, a precise and sound analysis tool for executables has the benefit of extending source code analysis to machine level, where the actual code is executed. The translation process in compilers might introduce errors that are only visible at the machine code level [BR10]. Current approaches exist to improve compiler correctness [BDL06; Ler06; BDP12], or aid the analysis of low-level code using results from source level analyses [Nec97; Riv03]. However, the analysis of executables independently of the compiler and source language semantics is still a necessity for auditing security critical code.

1.2 Challenges in the Analysis of Executables

Security auditing has traditionally been forced to consult the program executables rather than its source code for finding vulnerabilities. The sheer size of today's software also leads to more defects. Since the corresponding binaries lack any structure that make a large project manageable, tool support is mandatory to reverse engineer any real-world software.
Current tools like IDA-Pro help with the reconstruction of the control-flow graph (CFG) but often require manual guidance when confronted with optimized or obfuscated code, malware, etc., due to the use of code patterns that can only capture a limited set of compiler idioms. Recent approaches address this shortcoming by inferring possible values of registers and memory locations which helps to resolve most indirect jumps [Fle+10; KVZ09; Tha+10; BR10].

The task of resolving indirect jumps is complicated by architectures such as Intel x86 that do not enforce a fixed length for all instructions but use a variable length encoding to obtain a denser instruction packing. On such architectures it is crucial to precisely infer the target of a computed jump since computing an incorrect target would decode instructions starting in the middle of other valid instructions or even in areas containing data [SDA02]. In contrast, the ability to jump into the middle of another instruction is a tool used in exploits to execute instructions that were never emitted by the compiler [Sha07]. Thus, it is a laudable goal for an analysis to be very precise when inferring targets for computed jumps.

The reconstruction of the CFG is only a first step to finding vulnerabilities. One way forward is the use of a precise and sound static analysis that is able to identify large parts of the code as correct so as to focus the security engineer on potentially vulnerable code fragments. In this work, we address this concern by presenting a static analysis that can infer precise information on memory regions and their content, thereby helping to prove memory accesses correct.

1.3 Our Goals

Our goal is to provide a framework for the analysis of binary code that produces comparable results to analysis frameworks designed for source code analysis. A static analysis of executables differs from the analysis of high-level languages in that the control flow graph of the program (CFG) is not known a priori. The CFG, in turn, is necessary to propagate states between basic blocks and to infer that a fixpoint is reached in each loop. The reconstruction of the CFG requires a precise inference of possible jump targets for indirect jumps and calls. In the case of **`switch`**-statements, the generated jump tables can often be resolved by tracking a finite set of values, a so-called value-set analysis. Our aim, however, is to implement more sophisticated analyses that are required to assert, for instance, the correctness of ordinary pointer accesses such as those to stack-local arrays which are of interest to a security auditor working on executables. In particular, our framework should be able to prove the absence of specific program errors, but not be limited to:

1. dereferences of uninitialized pointers
2. out of bounds array accesses
3. buffer overruns on the stack
4. division by zero
5. integer wraparounds

Our framework is not able to analyze self-modifying code but our analysis emits a warning if the program is trying to modify the code segment. Moreover, our analysis is robust enough to deal with some obfuscation [CN09] techniques. Especially, techniques that obfuscate the control flow, thus trying to make program comprehension more difficult, do not have an impact on our framework [SMS11] as we perform an abstract interpretation of the program. Even techniques dealing with the increasingly popular obfuscation by virtualization [CN09] may be integrated in our framework by, e.g. following the ideas proposed in [Kin12].

1.4 Thesis Organization and Contributions

Some of the contributions of this thesis have been published at refereed conferences. In particular, the published work is as follows: [SMS11] an overview of our analyzer; [MSS13] an improvement on widening; [MS14] a novel abstract domain combining predicates and numeric domain values and a novel abstract domain that deals with undefined values [SMS13]. This thesis expands on the published topics and adds new insights or modifications to the aforementioned work. Furthermore, some chapters contain novel, unpublished work which is pointed out below.

We will commence this thesis by introducing the theoretical framework (Abstract Interpretation) that our analyzer is based on. After this, we explain in detail our analysis and its components before describing novel ideas to improve the precision of binary analyses. Finally we discuss implementation details and show applications of the analyzer and conclude with ideas for future work. The thesis is structured as follows:

Chapter 1 introduces the necessity of machine code analysis and the problems faced in machine code analysis.

Chapter 2 describes in detail the structure of our analyzer [SMS11]. In particular it introduces the intermediate representation used in the analyses; defines the static analysis performed on this intermediate language; explains in detail the abstract domains that we use and how they are combined; elaborates on the analysis of procedures as performed in our framework.
The chapter is based on the work presented in [SMS11] which was co-authored with Alexander Sepp and Axel Simon. The RREIL language and the initial structure of the

analyzer is the work of the main author Alexander Sepp, where my contribution consists in parts of the implementation and writing. However, the analyzer framework has grown since the initial publication and new abstract domains have been added to the framework. The domains dealing with memory, pointers and the heap are the work of my colleague Holger Siegel whereas the *segment* domains and the *affine* domain are Axel Simon's contribution. The remaining domains, the *zeno* domains are my contribution. Additionally, various students have contributed code to our analyzer.

Chapter 3 discusses the problem of precision loss due to widening and our novel approach to improve widening. The chapter is based on the work presented in [MSS13] that extends the initial implementation of the thresholds widening domain by Alexander Sepp. The extension added new widening heuristics and improved the initial ideas behind widening with thresholds.
In addition to the widening domains presented in [MSS13] this chapter describes the inference of semantic constants and a new version of the *delay* domain that uses thresholds. Additionally, we simplified the *thresholds* domain and changed it to allow repeated application of thresholds, which improves the precision of widening when analyzing nested loops.

Chapter 4 introduces a novel abstract domain, the *predicate* domain [MS14], that allows to express non-convex invariants and recovers the precision loss due to the convex approximation in convex numeric domains.

Chapter 5 introduces a novel abstract domain [SMS13] that relates defined and undefined values of variables with flags in numeric domains. The domain thus allows to maintain the precise value of a variable on joins with program paths where the variable is not defined.
The chapter is based on the work presented in [SMS13] that is co-authored by Holger Siegel and Axel Simon. My contribution involves the implementation of the abstract domain with improvements on the practicality of the domain, whereas Holger Siegel came up with the theoretical background and the formulation of the domain.

Chapter 6 discusses a new and unpublished idea to improve the results of a static analysis by focusing it on parts of the program, a so-called "dynamically started static analysis". In addition to the scalability problems when using the static analysis on large programs, the precision loss would accumulate too many false positives before reaching the program point of interest. Employing this method we were able to precisely analyze certain parts of a larger program.

Chapter 7 elaborates on implementation details of the analyzer. Note that this describes the joint implementation work of our group consisting of Alexander Sepp, Holger Siegel, Axel Simon and myself.

Chapter 8 describes the tools developed for interacting with the analyzer. We realized early that for the successful development of a static analyzer it is paramount to be

able to debug complex abstract domains and the fixpoint computation. To this end we developed a GUI that visualizes the results and allows to easily sift through the amount of data that an analysis outputs.

Chapter 9 demonstrates the application of our analyzer on interesting case studies. In particular we show how the novel ideas to improve the precision of static analyses work together to prove the correctness of a particularly challenging example.

Chapter 10 concludes and proposes ideas for future work.

2 Binary Analysis Framework

2.1 Abstract Interpretation

Abstract Interpretation is a well established static analysis technique used to perform automated program analysis and verification. An introduction to abstract interpretation is given in [CC10] and [Cou01] where the latter elaborates on the current state of the art of existing tools.
Abstract Interpretation is a formal method that soundly approximates the program semantics. It is able to statically, i.e. without running the program, infer runtime properties of a program. Soundness means that errors that are statically proved to be absent by the analysis do not occur in any execution of the program. That is, no runtime error is missed by the analysis or, equivalently no false negatives exist. On the other hand, abstract interpretation is incomplete meaning that it may flag errors in correct programs. These so-called false positives originate from the abstraction of the program semantics. The abstraction is key to make an analysis of all possible program runs tractable (computable) but it results in precision loss and may thus flag errors that never happen at runtime. Next, we will give an intuition of how abstractions are used in reasoning about the program semantics.

2.1.1 Programs as States and Transitions

Program properties can be seen as a description of a set of program states for which the property holds. Thus we may formulate the static analysis of programs and the inference of program properties by using descriptions of sets and their manipulation.
A program can be characterized by the set S of states that occur at runtime and the transitions $F \subseteq S \times S$ between these states $s_i \in S$. A program state contains, for example, the value of each variable in the program and the transitions are the semantics of program instructions. In order to compute an invariant for all possible runtime states S of the program, which might be infinite, we need a finite representation of S, an abstraction. Depending on the property that needs to be proved, a tractable abstraction $S^\sharp$ of the program states and its execution semantics $F^\sharp$ must be defined. We use a Galois connection between the concrete semantics F and its abstract semantics $F^\sharp$ to relate properties of the program in the abstract with those in the concrete. We use a stricter form of a Galois connection called a Galois insertion that is defined as follows:

Definition 2.1 (Galois Insertion) Given two partially ordered sets $\langle S, \subseteq \rangle$ and $\langle S^\sharp, \sqsubseteq \rangle$. Two monotone functions $\alpha : S \to S^\sharp$ and $\gamma : S^\sharp \to S$ form a Galois insertion iff:

$$\begin{aligned} \forall x \in S &: x \subseteq \gamma(\alpha(x)) \\ \forall y \in S^\sharp &: \alpha(\gamma(y)) \sqsubseteq y \text{ and } y \sqsubseteq \alpha(\gamma(y)) \end{aligned}$$

Fig. 2.1 shows the Galois insertion, i.e. $\alpha \circ \gamma = id$, between the concrete domain and its abstraction $\langle S, \subseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle S^\sharp, \sqsubseteq \rangle$. The abstraction function α lifts values from the concrete set to values in the abstract set. Its counterpart γ maps back values from the abstract set to the concrete domain. Using function composition we can compute F as $\gamma \circ F^\sharp \circ \alpha$. The abstraction by its very nature is lossy in that it might not capture precisely all the properties of the original program, thus in general: $F(x) \subseteq \gamma \circ F^\sharp \circ \alpha(x)$. This means the result can be less precise when computed in the abstract. Nevertheless, we are now able to compute the result of complex mathematical functions, e.g. the program semantics, by defining and composing abstractions of the program semantics.

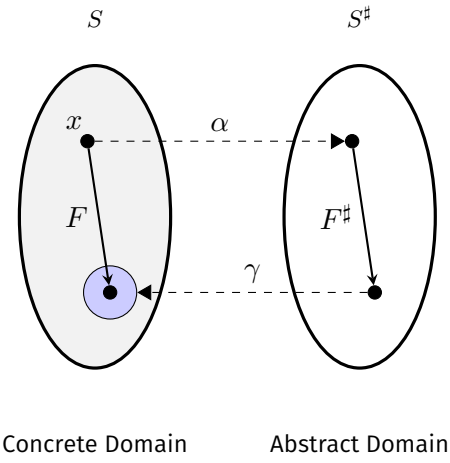


Figure 2.1: Galois insertion between the concrete and abstract domain.

If we choose $F^\sharp$ to only contain functions that are monotone with regard to a partial order $\sqsubseteq$ and an infimum exists in $S^\sharp$, then the solutions of $F^\sharp$ can be computed by iteratively applying $F^\sharp$ on the least element $\bot \in S^\sharp$ until we find a fixpoint [CC77; CC79a]. Note that intuitively a smaller element in $S^\sharp$ gives a more precise description about a state thus we are interested in finding the smallest fixpoint of $F^\sharp$.

2.1.2 Abstraction Examples

Let $\langle S^\sharp, \sqsubseteq \rangle$ be a lattice, thus satisfying above properties. If the lattice is of finite height and the abstract functions are monotone we always reach a fixpoint in finitely many steps due to the ascending chain condition [CC79b]. Take for example the lattice of constants shown in Fig. 2.2, a common abstraction used in constant propagation analyses. The lattice

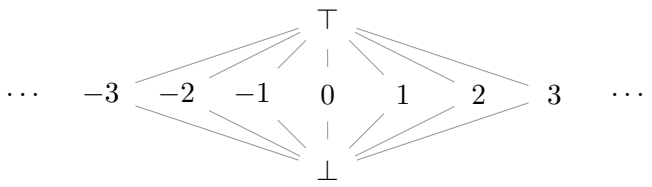


Figure 2.2: Flat lattice of constants with a finite height.

elements in the figure are ordered from bottom ($\bot$), the lowest element, to top ($\top$), the greatest element and lines denote that an order relation holds between two elements. As lattices are only partially ordered not every pair of elements is comparable, e.g. 0 is neither lower nor greater than 1. We define an abstract domain that maps each variable in the program to a lattice element. We start with the lowest value $\bot$ and by executing the abstract functions $F^\sharp$ of the program we reach in finitely many steps the least fixpoint *lfp* mapping each program variable either to a single value $n \in \mathbb{Z}$ or to $\top$. Here, $\top$ denotes that a variable may contain any of the possible constants, thus being the most imprecise description but still a sound abstraction. In particular at the entry point of a program the values of the program variables are set to $\top$ to model unknown program input.

A more powerful lattice is the lattice of intervals as shown in Fig. 2.3. The abstract domain of intervals maps each variable to a set of constants described by an interval. Compared to the constants lattice the interval lattice is a more precise abstraction of the values of a variable. However, the interval lattice has an infinite height, thus to ensure termination of the fixpoint computation a special operator, called widening must be used. Widening extrapolates from known states thus it can be seen as a generalization from bounded proofs to unbounded proofs. A more detailed description of this operator and how a static analysis using abstract interpretation infers the fixpoint for a program is given in Sect. 2.2.

The power of abstract interpretation is that it proves the absence of errors without requiring user supplied invariants of the program to be analyzed. However, the semantics of instructions must be specified as abstract transformers, a task that has to be done once per analyzed language (e.g. using the C99 language specification) and abstraction. Furthermore, existing abstractions, so-called abstract domains, may be used to express numeric program properties and prove the correctness of programs with regard to these properties and the specification. Nevertheless, the challenge in static analysis is to design practical abstractions that are computable, efficient and expressive enough to infer interesting properties for complex programs. In Sect. 2.5 we discuss the abstract domains that our analyzer uses and what program invariants are inferred by each domain.

Before that however, we will give a formal definition of program semantics and safety properties and how these are computable using abstractions.

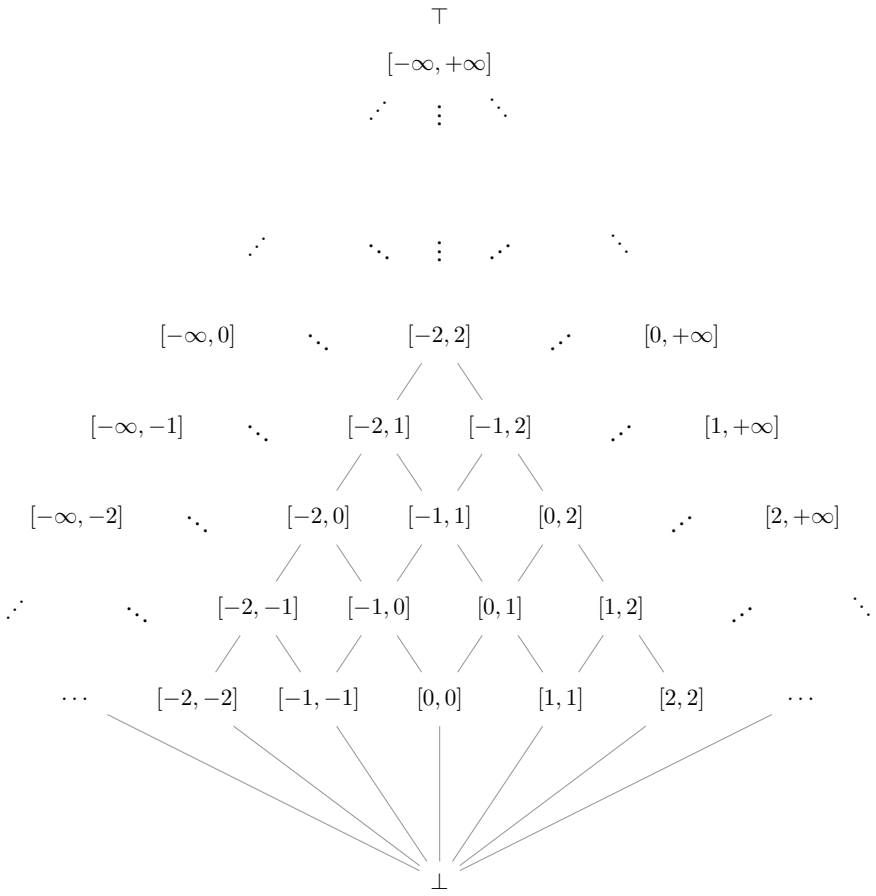


Figure 2.3: Lattice of intervals with an infinite height.

2.1.3 Definition of Program Semantics

The operational semantics of a program characterizes the reachable set of program states. Let A denote the set of possible addresses of a machine and let $P \subseteq A$ denote a set of addresses at which instructions can be executed, the program code. Furthermore, let $S : A \to V$ denote the state of a program, including stack, heap, global variables and code where V is some unit of memory, for instance, bytes. Given a pair of program counter and state $\langle p, s\rangle \in P \times S$, executing the n-byte instruction i stored at $s(p), s(p+1), \ldots s(p+n-1)$ leads to a new state of the program, defined by $\mathit{next}(p, s) = \langle p', s'\rangle$. Let a trace be a sequence $\pi = \langle p_1, s_1\rangle \cdot \langle p_2, s_2\rangle \cdots \in \Pi$ where $p_i \in P, s_i \in S$, $\mathit{next}(p_i, s_i) = \langle p_{i+1}, s_{i+1}\rangle$ and $\Pi \subseteq (P \times S)^*$ the set of all execution traces.

A program is not well-defined if it reaches a state $\langle p, s\rangle$ that does not represent a valid execution state of the processor. An invalid processor state is for example an unknown instruction $s(p), s(p+1), \ldots$ due to a misaligned jump target. Even if the instruction is valid, executing it may raise exceptions such as division-by-zero or a segmentation fault that are symptoms of a faulty program. Let $\mathit{safe} : P \times S \to \{\mathit{true}, \mathit{false}\}$ denote a predicate

that holds if calculating the next step from $\langle p, s\rangle$ is well-defined and raises no unwanted exceptions. We call a program correct if *safe*$(\langle p_i, s_i\rangle) = \mathit{true}$ for all $i \geq 0$ and for all traces $\cdots \langle p_i, s_i\rangle \cdots \in \Pi$ of a program. Note that functional faults such as incorrect calculations not defined in the specification of the program are therefore not considered as errors in our model.

The verification of a program requires that *safe* is shown to hold on all states in all executions of the program. Since there are an infinite number of traces with some of them passing through an infinite number of states, the task of statically verifying a program must be reduced to a computable problem which we do in two steps: The first step relies on the observation that the semantics of *safe* relies on the instruction being executed and, thus, is fixed for each program address $p \in P$. Furthermore, assuming that the code is not self-modifying the program addresses are finite. Thus, we can reduce the problem of evaluating an infinite number of *safe*-tests to evaluating one test for each program location $p \in P$. To this end, we can define the static or collecting semantics $\mathcal{CS} \subseteq P \times \wp(S)$ of all traces Π as follows:

$$\mathcal{CS} = \{\langle p, \bar{s}\rangle \mid p \in P \wedge \bar{s} = \{s \in S \mid \langle p, s\rangle \in \pi, \pi \in \Pi\}\}$$

The collecting semantics $\langle p, \bar{s}\rangle \in \mathcal{CS}$ associates each program point p with all states $\bar{s}$ with which it is executed. Thus we reduced the problem of computing the set of traces of a program (trace semantics) to the problem of computing all states for each program point (collecting semantics). As the second step by lifting *safe* : $P \times S \to \{\mathit{true}, \mathit{false}\}$ to operate on sets of states *Safe* : $P \times \wp(S) \to \{\mathit{true}, \mathit{false}\}$ we can now equivalently state that a program is correct iff *Safe*$(\langle p, \bar{s}\rangle)$ for all $\langle p, \bar{s}\rangle \in \mathcal{CS}$. In order to compute $\mathcal{CS}$ we analogously lift the transition relation *next* : $P \times S \to P \times S$ to operate on sets of states as follows:

$$\begin{array}{rcl} \mathit{Next} & : & P \times \wp(S) \to \wp(P \times \wp(S)) \\ \mathit{Next}(p, \bar{s}) & = & \{\langle p', \bigcup_i \{s'_i\}\rangle \mid \langle p', s'_i\rangle = \mathit{next}(p, s) \wedge s \in \bar{s}\} \end{array}$$

Given a starting point $\langle p_1, \{s_1\}\rangle$ and the transition relation *Next* we can now build the set $\mathcal{CS}$ by iteratively adding elements to it until a fixpoint is found. Let lfp_m denote the least fixpoint starting in m, then the collecting semantics can equivalently be defined as follows:

$$\mathcal{CS} = \mathit{lfp}_{\{\langle p_1, \{s_1\}\rangle\}}\left(\lambda m \,.\, \left\{ \langle p', \bigcup_i \bar{s}'_i\rangle \;\middle|\; \begin{array}{l} \langle p', \bar{s}'_i\rangle \in \mathit{Next}(p, \bar{s}) \\ \wedge \langle p, \bar{s}\rangle \in m \end{array} \right\}\right)$$

Although the collecting semantics only operates on a finite set of program points $p \in P$, it is still not computable since the states $\bar{s}$ associated with each p might be infinite. A static analysis therefore approximates a potentially infinite set of states with a single abstract state $s^\sharp \in S^\sharp$ and uses $s^\sharp_1 \sqcup s^\sharp_2$ to over-approximate the union of two states $s^\sharp_1, s^\sharp_2$. The abstract semantics can now be defined analogously to the collecting semantics:

$$\mathcal{A} = lfp_{m_1}\left(\lambda m\,.\,\left\{\langle p', \bigsqcup_i \bar{s}'_i\rangle \;\middle|\; \begin{array}{l}\langle p', \bar{s}'_i\rangle \in \mathit{Next}^\sharp(p,s)\\ \wedge \langle p, s\rangle \in m\end{array}\right\}\right)$$

Here, $m_1 = \langle p_1, s^\sharp\rangle \in P \times S^\sharp$ must be chosen such that $s_1 \in \gamma(s^\sharp)$ where $\gamma : S^\sharp \to \wp(S)$ states how an abstract state relates to a set of concrete states. The abstract transformer $\mathit{Next}^\sharp$ is allowed to over-approximate *Next*. This is expressed by requiring that for all $p \in P$ and abstract states $s^\sharp \in S^\sharp$, it must hold that $\bigcup_i s'_i \subseteq \gamma(\bigsqcup_i s'^\sharp_i)$ where s'_i and $s'^\sharp_i$ are defined as $\langle p', s'_i\rangle \in \mathit{Next}(p, \gamma(s^\sharp_i))$ and $\langle p', s'^\sharp_i\rangle \in \mathit{Next}^\sharp(p, s^\sharp)$ for all $p' \in P$. The set of successor points and states in the abstract semantics for a given $\langle p, s^\sharp\rangle$ always over-approximates the concrete semantics. A static analysis will choose an efficient function for $\mathit{Next}^\sharp$, thereby ensuring that $\mathcal{A}$ is computable in finite time.

Note that the collecting semantics and thus the abstract semantics can generally be extended to code executed by virtual machines or self-modifying code by lifting single program addresses to abstract addresses [Kin12]. Instructions are not anymore associated with a subset of fixed program addresses but are part of the abstract state $s^\sharp$. By introducing a virtual program counter p_v and tracking a set of instructions $\bar{i} = s^\sharp(\langle p, p_v\rangle)$ for each pair of program counter and virtual program counter we can over-approximate the executed instructions. However, the abstract transformers need to be lifted to take the virtual program counter into account. Additionally, due to possibly infinitely many valuations of the virtual program counter p_v to ensure termination a special widening is required [Kin12].

In the next sections we will detail how programs are represented and how the fixpoint of the abstract program semantics is computed. After which, we introduce our novel intermediate language RREIL that is used to describe the semantics of low level code and which our analysis takes as input. Finally, we will conclude this chapter by illustrating the various abstract domains that are used in our analysis.
Note that in the rest of this thesis, we use s instead of $s^\sharp$ to denote the abstract state as our analysis operates only on abstract states. Whenever the concrete state is required in the notation it is explicitly mentioned.

2.2 Preliminaries

2.2.1 Programs as Control Flow Graphs

This section details the program analysis problem we address. Our analysis operates on the control flow graph (CFG) of a program. The CFG is represented by a set of vertices labeled $v_1, v_2, \ldots$ and a set of directed edges representing the transfer functions. The transfer functions are either assignments $v_i \xrightarrow{Assign} v_j$ or assumptions/tests $v_i \xrightarrow{Pred} v_j$ where $Assign$ and $Pred$ are given by the grammar in Fig. 2.5. Here, the variables are denoted by x_i and the constants are in bold $\mathbf{c_i}$. Additionally, programs may use assertions of the form $v_i \xrightarrow{Pred} v_j$, e.g. `assert(x != 0);` that correspond to edges $v_i \xrightarrow{x=0} v_e$ to a designated error node v_e. Figure 2.4 shows an example program and the corresponding CFG. Here,

```
1 int x = f();
2 assert (x != 0);
3 while (x < 100) {
4   x = x + 1;
5 }
6 ...
```

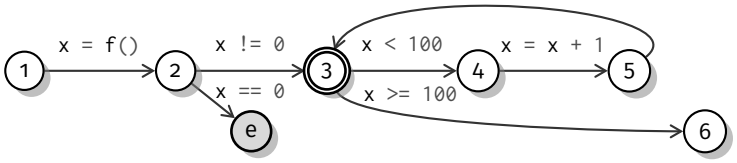


Figure 2.4: An example program and its control flow graph (CFG). Circled nodes are widening points. The node labeled with "e" is a designated error node.

$$
\begin{array}{llll}
\mathit{Pred} & ::= & \mathit{Test} & \\
\mathit{Test} & ::= & \mathit{Lin} \bowtie \mathit{Lin} & \\
\mathit{Assign} & ::= & x = \mathit{Expr} & \\
\mathit{Expr} & ::= & \mathit{Lin} \mid \mathit{NonLin} \mid \mathit{Test} & \\
\mathit{Lin} & ::= & \mathbf{c_0} + \mathbf{c_1} x_1 + \ldots + \mathbf{c_n} x_n & \\
\mathit{NonLin} & ::= & \mathit{Lin} \boxtimes \mathit{Lin} & \\
\bowtie & ::= & \leq | \not\leq | < | \not< | = | \neq & \\
\boxtimes & ::= & \times \mid / \mid \% \mid >> | << | \hat{} \mid \& \mid \mid &
\end{array}
$$

Figure 2.5: The grammar decorating a control flow graph (CFG).

nodes v_i are labeled with i and an empty edge $v_5 \longrightarrow v_3$ was added so that labels match line numbers. Note that the above defines an intraprocedural CFG where call edges, e.g. `x = f();` just connect to the next vertex in the CFG. An extension to an interprocedural CFG and different approaches to analyzing procedures is discussed in Sect. 2.7.

Note that throughout this document we use pseudo C code for our examples instead of programs that adhere to the given grammar and insert "..." wherever the code is irrelevant for the demonstration. In particular, an ellipsis as branch condition **`if`**`(...)` means that both branches, i.e. the **`then`** and **`else`** branches, are taken non-deterministically. An ellipsis used as condition in loops, e.g. **`while`**`(...)`, expresses that the loop will be executed an arbitrary number of times.
We use this pseudo C notation for presentational reasons as machine code would be more verbose and the C language is familiar to most readers. Nonetheless, the analyses operate on the machine code that is produced by compiling and disassembling the code shown in the examples. As a consequence of using the C language for code examples we also annotate the edges of the example CFGs using the same syntax instead of using our grammar defined above (e.g. `<=`, `!=` instead of $\leq$, $\neq$). This permits an easier matching of CFG edges with the source code of the examples. When discussing the analysis results, however, we use the corresponding notation used in our intermediate languages, that is closer to mathematical notation.

2.2.2 Fixpoint Analysis on the CFG

An analysis associates each vertex v_i with an abstract state $d_i \in \mathcal{D}$ where $\mathcal{D}$ is the universe of a lattice $\langle \mathcal{D}, \sqsubseteq_\mathcal{D}, \sqcup_\mathcal{D}, \sqcap_\mathcal{D}, \top_\mathcal{D}, \bot_\mathcal{D} \rangle$ with a greatest $\top_\mathcal{D}$ and a lowest element $\bot_\mathcal{D}$. The least upper bound of two lattice elements, the join operator $\sqcup_\mathcal{D}$ is defined as follows:

Definition 2.2 (Join) Given a domain $\mathcal{D}$ and a partial order $\sqsubseteq_\mathcal{D}$, define $\sqcup_\mathcal{D} : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ such that it is a sound over-approximation:

$$\begin{aligned} \forall x, y \in \mathcal{D} &: x \sqsubseteq_\mathcal{D} (x \sqcup_\mathcal{D} y) \\ \forall x, y \in \mathcal{D} &: y \sqsubseteq_\mathcal{D} (x \sqcup_\mathcal{D} y) \end{aligned}$$

The join performs an over-approximation (the union) of two states. The greatest lower bound of two lattice elements, the meet $\sqcap_\mathcal{D}$ is defined analogously as the dual to the join $\sqcup_\mathcal{D}$ operator. Meet describes the elements that are common in two states (the intersection). Finally, the greatest element (top) $\top_\mathcal{D}$ is defined as the join over all elements $\top_\mathcal{D} = \bigsqcup_\mathcal{D} \mathcal{D}$ and the least element (bottom) $\bot_\mathcal{D}$ defined as the join over the empty set $\bot_\mathcal{D} = \bigsqcup_\mathcal{D} \emptyset$. This definition gives rise to the following identities:

$$\begin{aligned} \forall x \in \mathcal{D} &: x \sqcup_\mathcal{D} \bot_\mathcal{D} = x \\ \forall x \in \mathcal{D} &: x \sqcup_\mathcal{D} \top_\mathcal{D} = \top_\mathcal{D} \end{aligned}$$

Initially, at the start of an analysis, the states at each vertex v_i are $d_0 = \top_\mathcal{D}$ and $d_i = \bot_\mathcal{D}$ for $i \neq 0$. That is, all nodes are unreachable except for the entry node d_0 for which we assume any possible state. The solution to the program analysis problem is characterized by a set of constraints on the abstract states $s_j \sqsupseteq_\mathcal{D} [\![F_i^j]\!]^\mathcal{D}(s_i)$, each constraint corresponding to an edge $v_i \xrightarrow{F_i^j} v_j$. The semantics or abstract transformer of an assignment edge $v_i \xrightarrow{x=e} v_j$ in $\mathcal{D}$ is given by $F_i^j = [\![v_i : x = e]\!]^\mathcal{D} : \mathcal{D} \to \mathcal{D}$; likewise for test edges. Where control flow paths merge, e.g. due to **if**-branches and loops, the join of all the incoming states is used as constraint for the merge point. Let v_i and v_j be predecessors of v_m, then the state s_m is computed using the constraint $s_m \sqsupseteq_\mathcal{D} [\![F_i^m]\!]^\mathcal{D}(s_i) \sqcup_\mathcal{D} [\![F_j^m]\!]^\mathcal{D}(s_j)$. This ensures a safe over-approximation of all paths that coincide at a program point, capturing all possible values at this point.

The solution of the analysis, that is, the solution of the constraint system, can be inferred using, e.g. chaotic iteration [Bou93] which randomly picks indices i, j for which the constraint is not satisfied and, for the edge from v_i to v_j updates s_j. The update is performed by joining the current value for s_j with the new computed value contributed from v_i, i.e. $s_j := s_j \sqcup_\mathcal{D} [\![F_i^j]\!]^\mathcal{D}(s_i)$ so as to ensure $s_j \sqsupseteq_\mathcal{D} [\![F_i^j]\!]^\mathcal{D}(s_i)$, for non-monotone transformers F_i^j. Consequently, the state at any program point s_j can only grow as the right-hand side of the equation is extensive [HH12]. The updates are performed until all constraints on the states are satisfied, that is, all updates do not contribute any new information. This fixpoint is the solution of the analysis.

2.2.3 Acceleration and Termination using Widening

If the lattice height is finite (e.g. the constants lattice in Fig. 2.2) then repeatedly applying the abstract transformers will eventually terminate and result in a fixpoint. In general, however, most interesting lattices $\mathcal{D}$ have infinite ascending chains. Especially inferring numeric information about program variables usually requires the use of abstract domains such as convex polyhedra [CH78] or intervals [CC76; Har77] that have infinite increasing chains, e.g. $[0,0] \sqsubseteq [0,1] \sqsubseteq [0,2] \sqsubseteq [0,3] \ldots \sqsubseteq [0,+\infty]$ (see Fig.2.3). Consider an interval analysis the examples in Fig. 2.6. Analyzing loop a) will terminate but very slow as it requires 1000000 iterations to reach a fixpoint whereas analyzing loop b) does not terminate.

a)

```
1 int i = 0;
2 while (i <= 1000000) {
3   ...
4   i++;
5 }
```

b)

```
1 int i = 0;
2 while (true) {
3   ...
4   i++;
5 }
```

Figure 2.6: Examples for loops with a) slow termination and b) non-termination if only join $\sqcup$ is used for state updates.

In these cases, termination of the fixpoint computation can be guaranteed if at least one widening operator ∇ is inserted into each cycle of the graph, that is, replacing the state update with $s_j := s_j \nabla_{\mathcal{D}} [\![F_i^j]\!]^{\mathcal{D}}(s_i)$. The widening operator must obey the following definition [CC76; CC77; CC92b]:

Definition 2.3 (Widening) Given a domain $\mathcal{D}$, define $\nabla_{\mathcal{D}} : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ such that:

$$\begin{aligned} \forall x, y \in \mathcal{D} &: x \sqsubseteq_{\mathcal{D}} (x \nabla_{\mathcal{D}} y) \\ \forall x, y \in \mathcal{D} &: y \sqsubseteq_{\mathcal{D}} (x \nabla_{\mathcal{D}} y) \end{aligned}$$

and for all increasing chains $x_0 \sqsubseteq_{\mathcal{D}} x_1 \sqsubseteq_{\mathcal{D}} \ldots$ the increasing chain $y_0 = x_0, \ldots y_{i+1} = y_i \nabla_{\mathcal{D}} x_{i+1}$ is not strictly increasing.

Comparing this to Def. 2.2, the difference between join and widening is that widening ensures termination by allowing only finite increasing chains in lattices of possibly infinite height. The idea of a widening operator is to extrapolate the change in the abstract state between consecutive iterations at a node in the graph. Consider the development of the state at the loop head (line 3) of the program in Fig. 2.7. Note that here we use an informal value analysis (using affine equalities and intervals) and will give concrete definitions for the join and widening on e.g. the interval domain later in Sect. 2.5.4. The first iteration shows the state s_3 with $x = y = 1$ due to the assignments in lines 1 and 2. Continuing this iteration the values are incremented inside the loop resulting in state s_3' with $x = 2$ and $y = 3$. Performing a join of this new state and the old state $s_3'' = s_3 \sqcup s_3'$, we can

approximate the resulting state s_3'' by a line $2x - 1 = y$ with start and endpoint $x \in [1, 2]$. This shows a powerful feature of the join operator. The join of two states infers a new invariant that holds in both states $2x - 1 = y$ and is not syntactically stated in the program. We could continue the analysis joining states until a fixpoint is found but to accelerate the fixpoint computation and ensure termination widening is applied in the third iteration $s_3'''' = s_3'' \nabla s_3'''$. Now widening extrapolates the state by removing the upper bound on x and y resulting in $x \in [1, +\infty]$ and $y \in [1, +\infty]$. Any further iteration results in a state s_3^* that is smaller or equal to this extrapolated state $s_3^* \sqsubseteq s_3''''$ and, thus, a fixpoint is found.

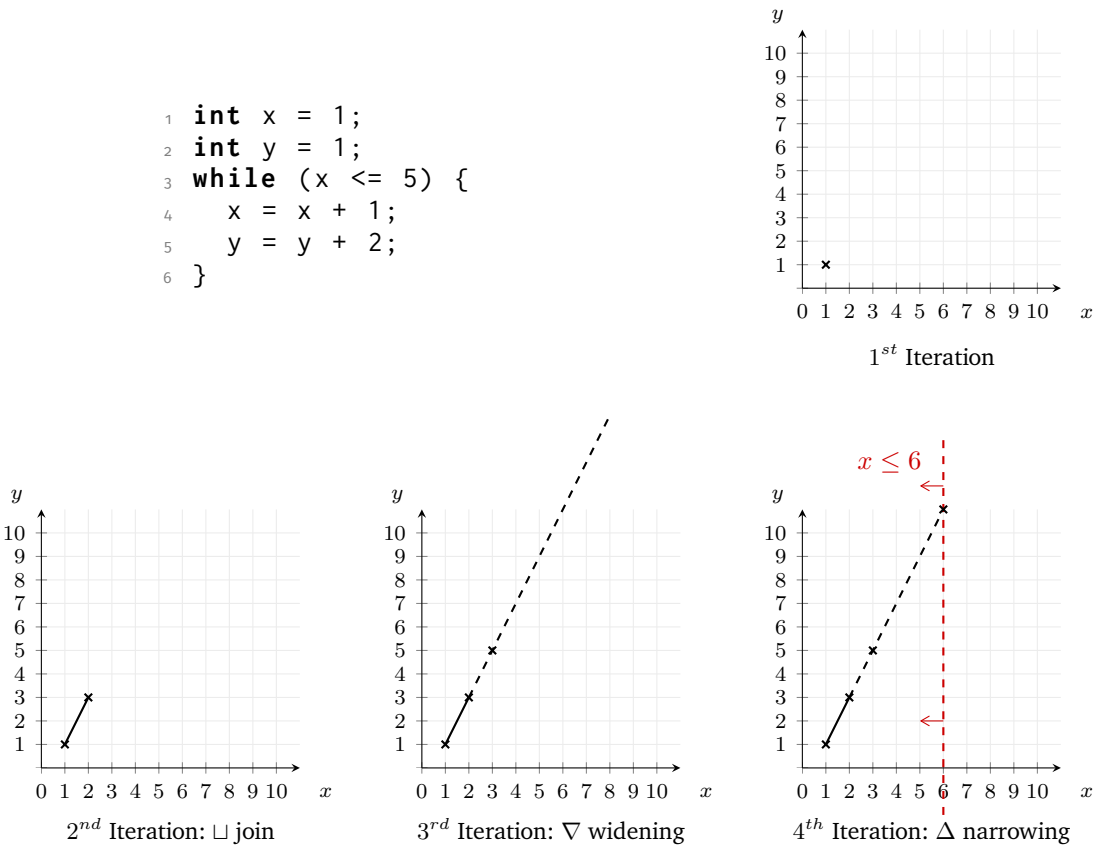


Figure 2.7: Development of the state at the **while**-loop head during the analysis.

2.2.4 Recovering from the Precision Loss of Widening

Evidently, the fixpoint inferred by widening vastly over-approximates the concrete states in line 3 of the program. To recover the precision loss of the extrapolation of widening a technique called narrowing is used. A narrowing operator Δ obeys the following definition [CC76; CC77; CC92b]:

Definition 2.4 (Narrowing) Given a domain $\mathcal{D}$, define $\Delta_{\mathcal{D}} : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ such that:

$$\forall x, y \in \mathcal{D} \quad : \quad y \sqsupseteq_{\mathcal{D}} x \Rightarrow (y \sqsupseteq_{\mathcal{D}} (x \Delta_{\mathcal{D}} y) \sqsupseteq_{\mathcal{D}} x)$$

and for all decreasing chains $x_0 \sqsupseteq_{\mathcal{D}} x_1 \sqsupseteq_{\mathcal{D}} \ldots$ the decreasing chain $y_0 = x_0, \ldots y_{i+1} = y_i \Delta_{\mathcal{D}} x_{i+1}$ is not strictly decreasing.

After inferring a fixpoint with widening we can replace widening by narrowing $s_j := s_j \Delta_{\mathcal{D}} [\![F_i^j]\!]^{\mathcal{D}}(s_i)$ and start a descending sequence from that fixpoint that eventually stabilizes in a more precise fixpoint. In Fig. 2.7 narrowing is used in the fourth iteration. As the state inside the loop is guarded by the test `x <= 5` narrowing can improve the previous state $x \in [1, +\infty]$ inferred by widening. This is achieved by propagating the widened state $x \in [1, +\infty]$ in another iteration through the loop, i.e. applying the guard $[\![x \leq 5]\!]\{x \in [1, +\infty]\} = x \in [1, 5]$ and incrementing the result in line 4 to $x \in [2, 6]$. Our analysis inferred in the first iteration that $2x - 1 = y$ holds and that this relation holds for all loop iterations, thus we can refine the values for y with narrowing, too. Further iterations cannot improve this result, thus we reached a fixpoint.

Although the precise (least) fixpoint can be inferred in this simple example using widening and narrowing this is not always the case. The precision loss introduced by the extrapolation of widening often cannot be recovered by narrowing. Hence, for widening techniques it is important to improve the extrapolation to curtail the precision loss. Widening, unlike join, is not monotone, commutative nor associative [Coro8]. Thus the precision of widening depends on the set of widening points [Bou93], the evaluation order of the constraints [AS13] and how it is implemented for a given domain. Note that these crucial factors to the precision of widening are even more delicate in machine code analysis as the CFG is not known up-front but constructed during the analysis and may depend on the precision of the analysis.

One goal of this thesis is to introduce improvements to widening and to the over-approximation of the join operator thereby maintaining the necessary precision for machine code analysis. A more detailed discussion about the limitations of narrowing and improvements to widening is given in Chap. 3. Improvements to the precision of numeric domains are discussed in Chap. 4 and Chap. 5.

2.3 Fixpoint Algorithm

To compute the fixpoint of the constraint system that corresponds to a program we use the worklist algorithm, shown in Fig. 2.8. The algorithm starts with an entry point p_0 to the program and an initial state s_0 and applies the abstract transformers (line 7) of the instructions in the program. The evaluation of an instruction returns a new abstract state $s_n \in \mathcal{D}$ and the location of the next program point $p_n \in P$ to consider for evaluation, that is, it computes $[\![F_c^n]\!]^{\mathcal{D}}(s_c)$. Due to branches, the evaluation may return more than one such

```
1  setState(p_0, s_0)
2  insert(p_0, W)
3  while (|W| ≠ 0) do
4    p_c ← extract(W)
5    insn ← disassemble(p_c)
6    s_c ← getState(p_c)
7    T ← evaluate(insn, s_c)
8    foreach (⟨p_n, s_n⟩ in T) do
9      addTransition(p_c, p_n, CFG)
10     s_o ← getState(p_n)
11     if (s_n ⋢_D s_o) then
12       s_r ← s_o ⊔_D s_n
13       setState(p_n, s_r)
14       insert(p_n, W)
15     fi
16   done
17 done
```

Figure 2.8: The fixpoint algorithm.

successor point, hence, a set T containing tuples of states and program points is returned. For each such tuple we retrieve in line 10 the abstract state of previous iterations (if there exists one, otherwise it is $\bot_{\mathcal{D}}$) and compare it to the current abstract state in line 11. If the current evaluation resulted in a greater state we update the tracked state for p_n by joining it with the old state at this point. Furthermore, as the program point p_n was updated we need to propagate this update to any program points that are reachable from p_n, that is, we need to insert p_n into the worklist (line 14). The algorithm thus iteratively considers all program points reachable from p_0 and updates the states for each such program point by applying the abstract transformers. The iterative computation is performed until the constraint system which is given by the program is satisfied and a fixpoint is found. In particular, a fixpoint is reached whenever evaluating the abstract transformer for any program point does not contribute any new information. In this case no new program points are added to the worklist W and the algorithm terminates. As described in Sect. 2.2.3, to enforce termination the join $\sqcup_{\mathcal{D}}$ in line 12 must be replaced by widening $\nabla_{\mathcal{D}}$ for programs containing loops. How we recognize loops and where we use widening instead of join will be detailed later in Chap. 3. Note that building up a CFG in line 9 is a side-effect of the reachability analysis.

After discussing the fixpoint algorithm and how abstract states are computed for each program point we will describe how the `disassemble()` function translates machine code to instructions with semantics. Moreover, we will show how an instruction is translated step by step into an abstract transformer for an abstract domain, thus describing how `evaluate()` transforms states and infers the next program points to be evaluated.

2.4 Intermediate Language (RREIL)

One challenge in analyzing machine code is the inference of precise numeric information to determine possible pointer offsets. It turns out that many intermediate representations in the literature express the semantics of native instructions at the bit-level [CS98; DP09] which often requires expensive SAT solving techniques to infer numeric range information [BK10; KK08a]. We propose to re-use static analysis techniques such as numeric abstract domains (intervals [Har77], linear equations [Kar76] and inequalities [CH78]) that have been successfully applied to high-level languages [Bla+03a].

Employing numeric domains for binary analysis is challenging as assembler instructions operate on bit vectors whose numeric value depends on whether the vector is interpreted as signed or unsigned integer. In particular, certain instructions such as **add** or **shl** have the same bit-level semantics regardless of whether their arguments are signed or unsigned and thus no information is available on the signedness of their arguments. In contrast, a relational test $a \leq b$ translates into different assembler instruction sequences, depending on the signedness of a, b. We present an analysis framework that maps native assembler into a small intermediate language (IL) called RREIL that is carefully crafted to allow for an easy recovery of numeric information, especially from relational tests and conversions between differently sized integers. RREIL is then translated into an IL on bit-vectors which is, in turn, translated into an IL on variables over $\mathbb{Z}$. An abstract domain that implements an analysis thus implements an interface in form of one of these ILs. For example, an interval domain implements the IL over $\mathbb{Z}$ while a bit-level analysis based on SAT-solving would implement the IL over bit vectors. We argue that this hierarchical approach allows for both, efficiency and precision.

2.4.1 Designing an Intermediate Language for Relational Analysis

Mobile devices and other specialized architectures like PLCs (programmable logic controllers) are increasingly used in security critical contexts. Automatic analysis of these diverse systems requires tools to abstract from the instruction set of each individual architecture by using an intermediate language. Given this multitude of devices simulating the semantic effect of each instruction of each platform becomes infeasible. In fact, even when targeting a single platform such as Intel x86, the semantic effects of over 900 instructions would have to be simulated. However, each x86 instruction usually assembles several distinct effects. Not unlike the decomposition into micro-ops within the processor, where each instruction is translated into a small intermediate language with simple semantics. This is the idea of the next sections, that motivate the design of a new intermediate language called RREIL.

2.4.2 Translation of Comparisons

One goal in the design of such intermediate languages has been to minimize the number of instructions of the IL required [CS98; DP09]. A small IL promises a simple analysis, however, small ILs often require clever bit-shifting and bit-masking operations to model a

native instruction. Consider the following translation of the x86 instructions that implement a comparison followed by a conditional jump (jump if less):

```
0 cmp eax, ebx
1 jl  some_target
```

The translation into REIL (Reverse Engineering Intermediate Language) [DP09] which was the starting point for our RREIL, is as follows (the numbers following an operator, e.g. :32 show the bitsize of the operator):

```
0  and  t0:32, 2147483648:32, eax:32
1  and  t1:32, 2147483648:32, ebx:32
2  sub  t2:64, ebx:32,        eax:32
3  and  t3:32, 2147483648:32, t2:64
4  bsh  SF:8,  -31:32,        t3:32
5  xor  t4:32, t1:32,         t0:32
6  xor  t5:32, t3:32,         t0:32
7  and  t6:32, t5:32,         t4:32
8  bsh  OF:8,  -31:32,        t6:32
9  and  t7:64, 4294967296:64, t2:64
10 bsh  CF:8,  -32:32,        t7:64
11 and  t8:32, 4294967295:64, t2:64
12 bisz ZF:8,  t8:32
13 xor  t9:8,  OF:8,          SF:8
14 jcc  t9:8,  some_target:32
```

This translation converts the compare instruction into a subtraction **sub** that stores its intermediate result in the temporary register t_2. Various bit operations then calculate the sign flag $SF = t_2[31]$, overflow flag $OF = (eax[31] \oplus ebx[31]) \wedge t_2[31]$, carry flag $CF = t_2[32]$, and zero flag $ZF = \neg(t_2[0] \wedge \ldots \wedge t_2[31])$. Here, **bsh** is a bit shift and **bisz** corresponds to the not operator ! of C. The intention of **jl** is to test if $eax < ebx$ where eax and ebx hold signed integers. In the above translation the conditional jump **jcc** is testing if $SF \oplus OF = 1$. This translation is at odds with the need of common static analyses that infer numeric information on program variables. These analyses require explicit relational tests to infer precise bounds on program values that are required to find, for instance, buffer overflows. The bit-level semantics render the extraction of these relational tests difficult: recovering the semantics $eax < ebx$ of the tests requires the analysis to recover the arithmetic semantics implemented by **cmp** by tracing the bit-level calculation of the flags. The implicit assumption of using bit-level semantics in intermediate languages seems to be that SAT solving is to be used to reason about the values of variables. However, using SAT solving to perform a reachability analysis (in contrast to inferring single traces) faces scalability problems as soon as non-trivial loops are analyzed. Even when combining SAT solving with abstraction to numeric ranges [BK10], the resulting analyses only perform well on embedded code where registers are 8 bit wide. Thus, our aim is to obtain a scalable analysis by applying the arithmetic semantics implemented by **cmp** directly on some numeric abstract domain. One way forward would be to pattern match the Intel or

REIL instruction sequences that implement **cmp** r1,r2; **jl**, **cmp** r1,r2; **jg**, etc. However, this approach is architecture-specific as each instruction set is different. Furthermore, it fails if an optimizing compiler inserts instructions between the test and the jump or if malicious code is considered.

We therefore propose a new variant of REIL [DP09] called RREIL (Relational Reverse Engineering Intermediate Language) in which flag calculations are translated into arithmetic instructions if possible. For instance, the introductory example x86 code:

```
0 cmp eax, ebx
1 jl  some_target
```

translates into RREIL as follows:

```
0 sub    t0:32,      eax:32,      ebx:32
1 cmpltu CF:1,       eax:32,      ebx:32
2 cmpleu CForZF:1,   eax:32,      ebx:32
3 cmplts SFxorOF:1,  eax:32,      ebx:32
4 cmples SFxorOForZF:1, eax:32,   ebx:32
5 cmpeq  ZF:1,       eax:32,      ebx:32
6 cmplts SF:1,       t0:32,       0:32
7 xor    OF:1,       SFxorOF:1,   SF:1
8 brc    SFxorOF:1,  some_target:32
```

The comparisons in lines 1 to 6 test for, $<, \leq$ of unsigned values, $<, \leq$ for signed values, equality and whether the result is negative, respectively. The translation creates what we call virtual flags, that is, flags that are not present in the processor but which express relational information that can be conveyed to numeric domains. For instance, $SFxorOF$ is set iff $eax < ebx$ when eax and ebx are interpreted as signed integers. The benefit of this approach is that each arithmetic test is associated with one single flag and, once the conditional branch is reached, the test matching the flag can be applied to the numeric abstract domain. Observe that the overflow flag OF carries no meaningful arithmetic information but is still calculated to also allow bit-level analyses such as SAT solving.

The unsatisfactory translation of relational tests into REIL is only one reason for our new intermediate language RREIL. Two other differences are that in RREIL non-Boolean arguments to an instruction, e.g. **sub**, **cmpltu**, etc., must have the same size (no implicit conversions) and may be accessed at an offset, e.g. eax:24/8 accesses only the upper 24 bits of eax. This implies that our analysis cannot directly analyze REIL code. Thus, RREIL is not an extension of REIL; rather, it is a new IL whose features are carefully chosen to allow for a precise numeric analysis of the resulting RREIL code, as detailed in the following sections.

2.4.3 Fields in Registers

A thorough understanding of an executable requires the recovery of memory layout, that is, the location and sizes of variables on the stack, heap and global memory. Indeed, many

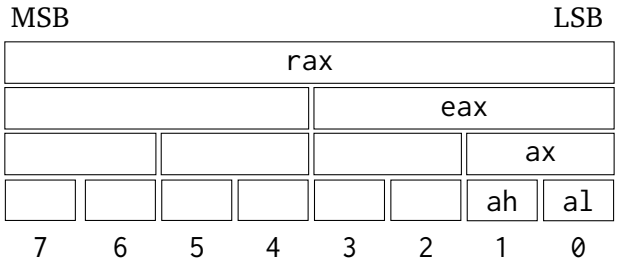


Figure 2.9: The possible fields of register rax.

analyses incorporate some sort of memory layout inference [BR04]. The idea is to represent sequences of bytes with a single numeric variable in some abstract domain. We extend this layout inference to registers, thereby enabling the access to registers with different sizes and offsets which is common on the x86-32 and x86-64 platforms. One particular oddity of the x86-64 architecture is the implicit zero-extension when writing the lower 32 bits of a register. For instance, the instruction **dec** eax has the effect of setting the upper 32 bits of rax to zero and the lower 32 bits to $eax - 1$. The translation into RREIL of this instruction (without the calculation of flags) is as follows:

```
0 sub rax:32/0, rax:32/0, 1:32
1 mov rax:32/32, 0:32
```

Here, the notation rax:sz/o is an access to sz bits of rax starting at bit offset o. Thus in the above example rax:32/0 corresponds to the register eax, as can be seen in Fig. 2.9. Note that we usually omit the offset if $o = 0$, writing rax:32 for eax. In principle, our analysis can infer fields at any bit offset and of any size. However, the translations of common register instructions will lead to byte aligned fields as shown in Fig. 2.9. Inferring these fields by tracking accesses has been described in the context of analyzing C programs [Mino6a; Simo8b]. For the sake of an example, suppose that the analysis has inferred the interval $[-2, 2]$ for eax. The analysis of the two instructions above will infer the interval $[-3, 1]$ for the lower half of rax while the interval $[0, 0]$ is inferred for the upper half of rax. In contrast, we could instead use a single numeric variable to model the content of rax:

```
0 sub           t0:32, rax:32, 1:32
1 zero-extend   rax:64, t0:32
```

Here, the instruction **zero-extend** calculates the effect of zero-extending the 32-bit value t_0 to rax. However, the best interval approximation for rax in the above example would be $[0, 2^{32} - 4]$, that is, nearly the full 32-bit unsigned range. Other, more precise, numeric domains such as polyhedra [CH78] will incur a similar precision loss. Thus, modeling registers as a set of fields leads to an improved precision when employing numeric abstract domains.

A challenge arises in the analysis when converting between differently sized fields as it becomes difficult to retain precise numeric information. For instance, consider a function

f returning **unsigned int**. The code:

```
1 unsigned int f() {
2   ...
3 }
4
5
6 unsigned long n = f();
7 n = n - 3;
```

translates into the following Intel x86-64 instructions (where the compiler has chosen to store n in the register eax):

```
0 call f
1 mov  eax, eax
2 sub  rax, 3
```

Here, the second statement implicitly sets the upper part of rax to zero which gives rise to the following RREIL code:

```
0 call f
1 mov  rax:32/32, 0:32
2 mov  rax:32,    rax:32
3 sub  rax:64,    3:64
```

When evaluating the **sub** instruction, the analysis finds two 32-bit fields that constitute the 64-bits that are to be read. Depending on the values of the two fields, a new 64-bit field can be synthesized that holds a precise numeric value. For instance, given an upper field whose value is tracked by u and a lower field tracked by l with both values lying in $[0, 2^{32} - 1]$, a new 64-bit field c can be created with $c = 2^{32}u + l$. In the example, since $u = 0$, this reduces to $c = l$. More propagation rules are developed in [Simo8b, Chap. 5].

Although SAT solving can infer range information, these approaches are less precise than our field-based approach [Eld+11, Ex. 4] nor do they scale [KS10]. Scalability was one aspect in the design of RREIL as detailed next.

2.4.4 Making Side-Effects Explicit

The translation to RREIL code makes every side-effect of native instructions visible similar to other intermediate languages [DP09; Son+08]. An instruction has side-effects if it modifies registers that are not explicitly mentioned as instruction arguments. Hence, one machine instruction translates to several RREIL instructions that express the side-effects. In order to map between RREIL and the native instructions, RREIL addresses are encoded using the special notation $x.y$ where x denotes the underlying native instruction's address and y the yth RREIL instruction used to translate the native instruction. Using these so called sub-instruction offsets allows for defining loops that are required for the translation of x86 instructions using the **repz** and **repnz** prefixes. To demonstrate implicit side-effects, consider the translation of the instruction **call** 11 on x86 to RREIL:

```
06.00: sub     esp:32, esp:32, 4:32
06.01: store   esp:32, 7:32
06.02: br      11:32
```

First, the stack pointer is decremented after which the address of the native instruction immediately following the **call** is saved on top of the stack, then control is transferred to the target function. Analogously, the translation of the x86 **ret** instruction results in this RREIL code:

```
19.00: load    t0:32,  esp:32
19.01: add     esp:32, esp:32, 4:32
19.02: br      t0:32
```

The return address is loaded from the stack, which requires incrementing the stack pointer and then a jump to the return address is performed. Note that semantically the instructions for **call** and **return** in RREIL are simple jumps.

2.4.5 Reducing the Size of RREIL Programs

A major disadvantage in the analysis of binaries is the output size of the translation into the IL. As illustrated in Sect. 2.4.2, a simple comparison translates into no fewer than 8 RREIL instructions. Moreover, the translation of arithmetic instructions, e.g. **add**, **sub**, creates similarly many assignments to flags, even though the flags are often not tested thereafter. Many ILs that express the semantics at the bit-level create even more instructions; consider, for instance, the corresponding REIL translation of the test in Sect. 2.4.2 yielding 15 instructions. Fortunately, a simple liveness analysis followed by a dead code removal can often eliminate the majority of instructions as their calculated result is never used. For instance, consider again the introductory example consisting of 8 RREIL instructions:

```
0 sub     t0:32,      eax:32,      ebx:32
1 cmpltu  CF:1,       eax:32,      ebx:32
2 cmpleu  CForZF:1,   eax:32,      ebx:32
3 cmplts  SFxorOF:1,  eax:32,      ebx:32
4 cmples  SFxorOForZF:1, eax:32,   ebx:32
5 cmpeq   ZF:1,       eax:32,      ebx:32
6 cmplts  SF:1,       t0:32,       0:32
7 xor     OF:1,       SFxorOF:1,   SF:1
8 brc     SFxorOF:1,  some_target:32
```

Applying dead variables removal only 2 RREIL instructions remain:

```
0 cmplts SFxorOF:1, eax:32, ebx:32
1 brc    SFxorOF:1, some_target:32
```

Moreover, due to the use of fields in the RREIL language, liveness analysis can be performed for each field (rather than for the whole register), thereby also removing most of the RREIL instructions generated for the x86-64 instruction set that model the implicit zero extension of 32-bit arithmetic instructions. Since liveness is a global analysis, full dead code removal can only be done once the complete CFG of a function is known. However, a partial removal is still possible by assuming that all variables are live after an unresolved branch. Further optimizations could reduce the size even more, ultimately enabling analyses on the binary that are already possible at the C level [Bla+03a].

2.4.6 A Formal Definition of RREIL

We conclude the discussion of translating assembler to RREIL by a formal definition of the RREIL syntax. A precise definition is instructive not only for comparisons with other languages, but also in illustrating how RREIL instructions are broken down into simpler statements that relate to memory regions, fields of memory regions and their numeric content. This simplification is shown in detail in Sect. 2.5.
We commence by presenting the RREIL instructions that concern the copying, the read and write access of a memory cell, a conditional branch and an unconditional branch.
The latter carries a hint from the translator about the type of the jump, either a function call, return instruction or a plain jump. Note that the hint is only syntactic, that is, hints carry no semantic meaning. The translator provides them only if the architecture has special instructions for procedure calls. Interprocedural analyses may benefit from discriminating between these jumps as described in Sect. 2.7. For example, a first approximation of procedure boundaries can be achieved by distinguishing between jumps and calls/returns. As we will show in Sect. 2.5.1, discovering procedure bounds can be performed semantically in an abstract domain.

In all statements, information travels from right to left, following the Intel convention.

$$
\begin{aligned}
\textit{rreil} ::= & \ \textbf{mov}\ \textit{lval}, \textit{rval} \\
| & \ \textbf{load}\ \textit{lval}, \textit{lval} \\
| & \ \textbf{store}\ \textit{lval}, \textit{rval} \\
| & \ \textbf{brc}\ \textit{condition}, \textit{rval} \\
| & \ \textbf{br}\ \textit{rval}, \textit{branchtype} \\
| & \ \textit{rreil-stmt} \\
\textit{rreil-stmt} ::= & \ \textit{binop}\ \textit{lval}, \textit{rval}, \textit{rval} \\
| & \ \textit{cmpop}\ \textit{flag}, \textit{rval}, \textit{rval} \\
| & \ \textit{primop}\ (\textit{lval})+, (\textit{rval})+ \\
| & \ \textbf{sign-extend}\ \textit{lval}, \textit{rval} \\
| & \ \textbf{zero-extend}\ \textit{lval}, \textit{rval}
\end{aligned}
$$

The right-hand sides of each instruction use the definition of a writable location *lval* and an *rval*, i.e. a constant expression or interval where $\mathbb{Z}_\infty = \mathbb{Z} \cup \{-\infty, \infty\}$. We use the term memory region/variable for $id \in \mathcal{X}_M$ which encompasses the set of registers and temporary variables that are generated during translations. Each memory region carries the size of the access which must be the same across an instruction. One of two exceptions to this rule are *flags* that are always one bit wide. The second are explicit conversion instructions, shown in the definition of *rreil-stmt*. Memory regions may carry an optional bit offset written */o*.

$$
\begin{aligned}
\textit{lval} &::= \textit{var} \\
\textit{rval} &::= \textit{var} \mid \underline{c} : \textit{sz} \mid [\underline{l}, \underline{u}] : \textit{sz} \\
\textit{var} &::= \underline{id} : \textit{sz}(/o)? \\
\textit{condition} &::= \textit{flag} \mid \underline{c} \\
\textit{flag} &::= \underline{id} : 1 \\
\textit{sz} &::= \underline{n} \\
o &::= \underline{n} \\
\textit{branchtype} &::= \textit{jump} \mid \textit{call} \mid \textit{return}
\end{aligned}
\qquad
\begin{aligned}
c &\in \mathbb{Z} \\
l, u &\in \mathbb{Z}_\infty \\
id &\in \mathcal{X}_M \\
c &\in \mathbb{N} \\
id &\in \mathcal{X}_M \\
n &\in \mathbb{N} \\
n &\in \mathbb{N}
\end{aligned}
$$

These definitions are also used in the rules for arithmetic and comparison operations whose mnemonics are defined as follows:

$$
\begin{aligned}
\mathit{binop} ::= & \ \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \\
& \mid \texttt{div} \mid \texttt{divs} \mid \texttt{shl} \mid \texttt{shr} \mid \texttt{shrs} \\
& \mid \texttt{mod} \mid \texttt{mods} \mid \texttt{and} \mid \texttt{or} \mid \texttt{xor} \\
\mathit{cmpop} ::= & \ \texttt{cmpeq} \mid \texttt{cmpleu} \mid \texttt{cmples} \\
& \mid \texttt{cmpltu} \mid \texttt{cmplts} \\
\mathit{primop} ::= & \ \underline{\mathit{name}}
\end{aligned}
$$

The binary instructions must be applied to arguments of equal size. Only the arguments of the conversion instructions: **sign-extend**, **zero-extend** feature different sizes. Binary operators express arithmetic operations for signed (carrying the s suffix) and unsigned values. Some operations are agnostic to the signedness, e.g. **add**, **sub**, **mul**. The comparison statements test for equality, for "less or equal" on unsigned and signed values and for "less than" on unsigned and signed values, respectively. Both arguments have the same size. Greater-than comparisons are translated by swapping their arguments.
Primitive operations are special functions definable in RREIL (see Sect. 7.2.3). They have a name as identifier and a list of outgoing *lval* and incoming *rval* operands. By using primitives one can forward the implementation of complex semantics to the abstract domains. For example, processor specific cryptographic instructions, e.g. AES, are easier approximated if implemented as a primitive in a numeric domain than translating them to a long block of RREIL statements.
While there is no intention to include such cryptographic primitives in RREIL future work should integrate concurrency primitives and the handling of floating point computations.

2.4.7 Conclusion

While the 24 RREIL instructions make for a concise language, the previous REIL language was even simpler, featuring only 17 instructions. As illustrated in Sect. 2.5, the RREIL language is translated several times during the analysis, each time turning into a simpler language that is interpreted by abstract domains. Since most end-user implemented domains will use one of these simpler languages, we believe that our design is in fact easier on third-party developers than the original REIL instruction set.

2.5 Hierarchy of Abstract Domains

The fixpoint algorithm discussed in Sect. 2.3 is parameterizable by different abstract domains $\mathcal{D}$. However, in our analyzer $\mathcal{D}$ is not only one domain but a set of abstract domains that each track some partial information about the program state. We combine different domains in order to improve the precision of the analysis as will be discussed later in Sect. 2.6.

Cofibered Domains The key idea of our approach is to implement abstract domains as cofibered domains [Ven96], an approach sometimes called "functor domains" [Bla+03a]. Here, each domain $\mathcal{D}$ has a child domain $\mathcal{C}$ that it controls. The combined domain is written as $\mathcal{D} \rhd \mathcal{C}$ and a cofibered state as a tuple $\langle d, c \rangle \in \mathcal{D} \rhd \mathcal{C}$. Note that the combined domain is itself a domain, thus can be again combined with another cofibered domain $\mathcal{E}$, yielding $\mathcal{E} \rhd \mathcal{D} \rhd \mathcal{C}$ and the state $\langle e, \langle d, c \rangle \rangle$. Consequently, cofibered domains are nested in a straightforward compositional way. Only the leaf domain, here $\mathcal{C}$ has no child and is not cofibered.
With cofibered domains, a transfer function is applied on the parent $\mathcal{D}$ with the child $\mathcal{C}$ being opaque. The parent domain is responsible for synthesizing transfer functions for its child. The benefit is that a transfer function $tr^{\mathcal{D}}$ of domain $\mathcal{D}$ on a state $s \in \mathcal{D} \rhd \mathcal{C}$ may execute any number of transfer functions $tr_1^{\mathcal{C}}, \ldots, tr_n^{\mathcal{C}}$ on its child $\mathcal{C}$ (even zero) before returning the result as a new state $s' = [\![tr]\!]^{\mathcal{D}} s$. This enables domains to perform reductions on the child during each transfer function $tr^{\mathcal{D}}$ (see Sect. 2.6).
Although we represent a cofibered domain $\mathcal{D}$ as a tuple $\langle d, c \rangle$ of a "local" state $d \in \mathcal{D}$ and a child state $c \in \mathcal{C}$, the child is actually an opaque object in the implementation. Specifically, let $L_A = \langle A, \sqsubseteq_A, \sqcup_A, \sqcap_A, tr_1^A, \ldots, tr_n^A \rangle$ be an abstract interpretation, that is a domain A equipped with abstract transformers. Then a cofibered domain is a functor $\mathcal{F} : L_A \to L_{\mathcal{D}}$ that, given abstract interpretations L_A and $L_{\mathcal{D}}$, returns a new abstract interpretation that combines both analyses. In particular a transfer function on the parent $tr^{\mathcal{D}} : \mathcal{D} \to \mathcal{D}$ is mapped to a set of transfer functions $tr^{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$ on the child [Ven96]. A resulting benefit is that the functor endows a cofibered domain with the ability to modify the variable support set of its child, as will be discussed next.

Support Set With few exceptions all domains have a so-called support set of memory $\mathcal{X}_M$ and/or numeric variables $\mathcal{X}_V$ for which they track some information. In literature [Bla+03a] the support set is sometimes called the environment Σ. As a result of the cofibered design, the support set does not have to be the same for all domains. In particular, this means that domains may track only a subset of all existing variables. Another benefit of the cofibered design is that a domain $\mathcal{D}$ may modify the support set of its child $\mathcal{C}$. For instance, a domain $\mathcal{D}$ tracking pointers as symbolic variables will introduce these pointer variables in the child $\mathcal{C}$ (a relational domain) and exploits the relational information that $\mathcal{C}$ infers for these variables, e.g. that two variables are equal. This allows to separate concerns, simplifies the design of abstract domains and encourages the reuse of existing domains.

Organization of Hierarchy Our set of abstract domains is organized as a stack of cofibered domains, where each domain is specialized on tracking certain properties of the program. Similar domains are grouped together and use a common interface to communicate. Between the four groups or tiers we use well-defined intermediate languages with RREIL being the language of the topmost domain. Towards the bottom of the domain hierarchy, the grammar of the intermediate languages becomes simpler. This is achieved by letting higher domains consume complex expressions and translate them into simpler ones, e.g. remove pointer expressions. As a result of this modular structure and the cofibered design

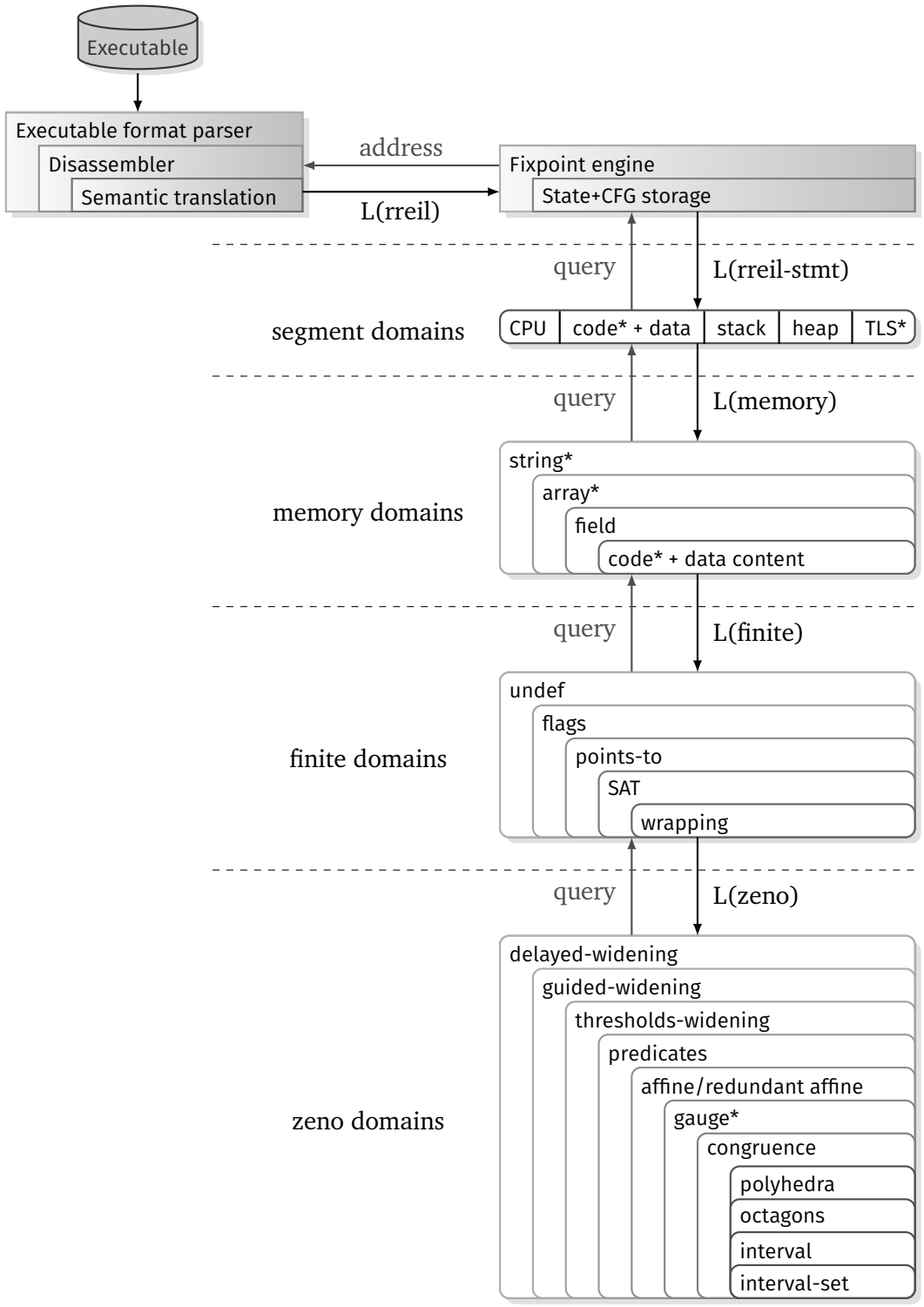


Figure 2.10: The analyzer structure and the hierarchy of the abstract domains. Domains with an asterisk (*) are unfinished or future work and thus not in use.

we are able to easily add domains to the hierarchy to specialize the analysis or remove domains to trade precision for scalability.
For the organization of the domains stack we follow the model used in Astrée [Cou+06] and add two additional tiers to suit the need for the analysis of executables. The original two tiers are the *memory* domains and the *numeric* domains that we call *zeno* domains as shown in Fig. 2.10. The additional two tiers are the *segment* domains and the *finite* domains. The former models the memory segments of a machine executable program and the latter are domains that deal with variables of fixed bit-sizes.

2.5.1 Segment Domains

Since RREIL allows a calculated address in memory accesses and indirect jumps, we create memory regions $m \in \mathcal{X}_M$ and associate a consecutive range of addresses with m that form a semantic entity such as variables on the stack, heap allocated memory, the code segment or the constants and data segments. An RREIL operation on pointers is translated into an operation on memory region(s) which is the task of the segment and memory domains. Some of the domains are also responsible for summarizing memory regions (such as heap regions or stack frames in case of recursive calls). However, the main task of the segment domains is to resolve a pointer access, that is, to determine to which memory regions the pointer points to and whether it is within the bounds of that memory region.

The interface to the segment domains is a subset of *L(rreil)* named *L(rreil-stmt)*. It contains all the instructions in RREIL without the control flow instructions as these are handled by the fixpoint engine and CFG storage in Fig. 2.10. The fixpoint engine resolves jump targets during the reachability analysis. In particular, given an unconditional branch instruction "**br** *rval*, *branchtype*", first all feasible targets for *rval* are queried from the domains stack. Each branch target t_i is associated with an abstract state, where $\mathit{rval} = t_i$ holds. These states are propagated each to the respective inferred target. For the conditional branch instruction "**brc** *condition*, *rval*" we first evaluate the condition and its negation (i.e. $\mathit{flag} = 1$ and $\mathit{flag} = 0$) before propagating the respective states to the jump targets of *rval* and the address following the branch. The latter is the fall-through case for $\mathit{flag} = 0$. The remaining RREIL instructions in *L(rreil)* are directly applied as operations on the domains.

The topmost set of abstract domains manages the organization of memory regions. The segment domain is a sum of abstract domains which together model the complete memory of a program after it has been loaded for execution [Lev99]. As an example, Fig. 2.11 shows the organization of the program memory under Linux on the Intel x86-32 platform. Our abstract domains model each of the shown segments except for the kernel memory.
A segment contains a set of memory regions with similar semantics. Each segment is disjunct from the other segments. Hence, a memory access on the set of segment domains is performed only in one of the segments, the one able to resolve the access. Thus, segment abstract domains themselves are not cofibered domains but the sum of all segments is a cofibered abstract domain. We now present each segment domain in turn.

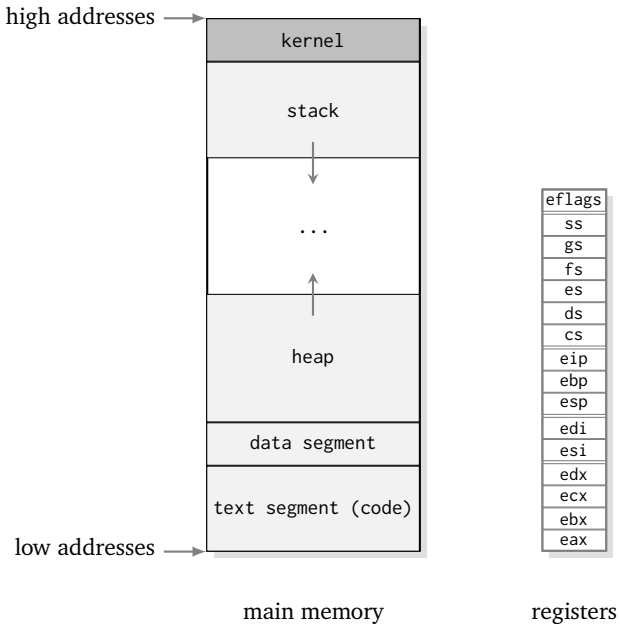


Figure 2.11: The memory layout of a program on Linux x86-32.

CPU The CPU abstract domain tracks and resolves the registers of a processor $\mathcal{X}_R \subseteq \mathcal{X}_M$. It translates from variable ids in the operands of an instruction to the subset $\mathcal{X}_R$ of CPU memory regions. Because the RREIL translation introduces temporary registers, the support set of the CPU domain will contain more variables than there are physical registers in a platform. Note that accesses to fields in a register must always happen at constant offsets. After all, current architectures do not allow for computed offsets to bits in registers.

code* and data Currently this domain only models accesses to global data segments but not to the code segments of a binary. The latter is future work and would allow us to handle self-modifying code as mention earlier (see Sect. 2.1.3). The domain is initialized at analysis start with the address range of data segments in the executable and is thereafter able to resolve pointers into these segments. We model two types of accesses to the data segment: accesses at absolute addresses for global variables and accesses at offsets to symbolic addresses, i.e. address variables: $a \in \mathcal{X}_V$. The latter type is work in progress and is intended for relocatable data and code.

stack This abstract domain models the stack and its subdivision into stack frames. In particular we do not track the stack as one large region but rather track single stack frames and pointers to the predecessor frame(s), that is, the frame of each caller (see Fig. 2.12). The *stack* domain is able to perform this separation by observing modifications to the stack pointer on assignments [Lak+11] and observing call and return instructions. The latter allows us to infer procedure boundaries during the analysis, associate each stack frame with a procedure and to reconstruct an over-approximation of the call graph. By

using pointers to predecessor frames we can model the stack either as a linked list of stack frames or as a graph with multiple predecessor frames. This flexibility allows us to use different approaches to interprocedural analysis as described in Sect. 2.7.

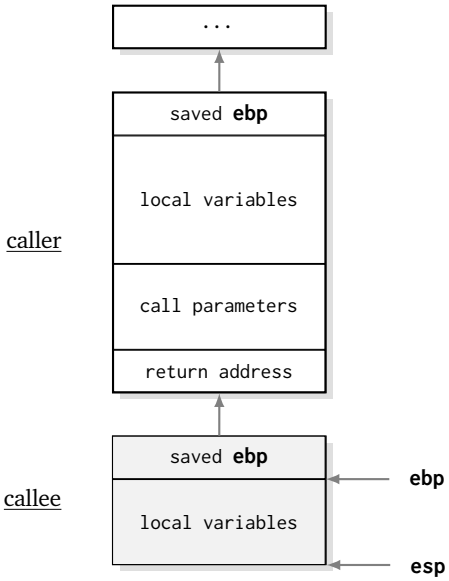


Figure 2.12: The stack layout of a program on Linux x86-32.

However, the main purpose of the stack domain is to check the bounds of read and write accesses to the stack. We resolve accesses to the frames of callers by following pointers to predecessor frames. These accesses do not issue a warning. However, non-constant accesses that cross boundaries between stack frames issue a warning. This allows us to differentiate between accesses to function parameters going to the previous stack frame (shown as "call parameters" in Fig. 2.12) and buffer overflows that might overwrite the return address. Future work should distinguish between static local variables and dynamically allocated variables (e.g. through `alloca()` in C) and warn if the boundary between these two regions is violated by a non-constant access. Moreover, by analyzing stack accesses and using debug information for the bounds of local variables we could also warn whenever accesses to a local array may overwrite other local variables. This feature is useful as exploiting an out-of-bounds vulnerability does not necessarily require overwriting the return address on the stack. If function pointers are stored on the stack, the overflow might corrupt these pointers to deviate the control flow. In the current implementation we only warn if the return address is overwritten.

heap The heap domain resolves pointers to memory regions that have been dynamically allocated using `malloc()`. It tracks a set of regions and performs the summarization of heap allocated structures using similar methods as in shape analysis [SS13]. In order to be able to track dynamically allocated memory the domain handles the RREIL primitives for `malloc()` and `free()` that need to be provided by the translation front-end (see Sect. 7.2.3).

TLS* This abstract domain implements *Thread-Local-Storage* semantics [Lev99], i.e. the addressing of different memory regions for a variable depending on which thread accesses the variable. It does this by maintaining a designated address variable and resolving memory accesses depending on the value of this variable. This is analogous to the way *Thread-Local-Storage* is implemented in current operating systems. Our implementation is unfinished, though, hence the domain is not in use. We also lack concurrency primitives in the RREIL language to support threads.

After discussing which domains are responsible for resolving pointers into a collection of regions we will discuss how accesses to each region are resolved. As each region itself may be accessed at different offsets, we subdivide regions into fields, which are consecutive sequences of bits.

2.5.2 Memory Domains

Let $\mathcal{X}_V$ be the set of variables, that map to numeric values in the current state. The purpose of the memory domains is to associate ranges of bits of $m \in \mathcal{X}_M$ with an abstract variable $x \in \mathcal{X}_V$. For instance, the register $rax \in \mathcal{X}_M$ may have a variable $x_l \in \mathcal{X}_V$ associated with the bits $0 - 31$, thereby representing the value of eax and a variable $x_h \in \mathcal{X}_V$ that represents bits $32 - 63$.
While registers can only be accessed at constant offsets, the interface *L(memory)* also allows accesses at calculated offsets involving a variable x:

$$
\begin{aligned}
\textit{memory} &::= \textit{mem-stmt} \mid \textit{mem-test} \mid \textit{mem-sup} \\
&\mid \textit{lval} = \textit{rval} \\
&\mid \textit{lval} = \textit{region} \xrightarrow{sz} \textit{offset} \\
&\mid \textit{region} \xrightarrow{sz} \textit{offset} = \textit{rval} \\
\textit{mem-stmt} &::= \textit{rreil-stmt} \\
\textit{mem-test} &::= \textit{flag} = \underline{0} \mid \textit{flag} \neq \underline{0} \mid \textit{fin-test} \\
\textit{mem-sup} &::= \texttt{intro}\ \textit{region} \mid \texttt{drop}\ \textit{region} \\
\textit{region} &::= \underline{m} && m \in \mathcal{X}_M \\
\textit{offset} &::= \underline{x} \mid \textit{sz} && x \in \mathcal{X}_V \\
\textit{sz} &::= \underline{n} && n \in \mathbb{N}
\end{aligned}
$$

As the language is in large parts similar to the RREIL grammar, we reuse some of the definitions from Sect. 2.4.6. The main operations here are the same as the *rreil-stmt* operations in RREIL, that is, arithmetic and comparison operations. The next production shows tests on flags which are used in implementing the semantics of conditional branches. Additionally, we allow tests for generic numeric assumptions for which the grammar productions given by *fin-test* are described later in Sect. 2.5.3. Next are the operations

manipulating the support set of the domain that allow to introduce or remove regions using **intro** and **drop**, respectively. The last three grammar productions of *memory* show the operation to copy memory regions and operations accessing fields in memory regions. Note that these are the translations of the **mov**, **load** and **store** instructions from *rreil*. The translation is performed by the *segment* domains during which pointers are resolved to memory regions with the help of the *points-to* domain (see Sect. 2.5.3). The resulting instructions access fields in memory by specifying the region, an offset and the size of the access. Creating and resolving fields for accesses is then the task of the *field* domain. We thus describe this domain first before discussing more specialized domains in this tier.

field In the above definition of *L(memory)*, the C-like pointer dereference $m \xrightarrow{sz} o$ accesses $m \in \mathcal{X}_M$ at an offset $o \in \mathcal{X}_V$ with a size of sz-bit. The *memory* domain resolves such accesses to fields in m representing a string of bits starting in o with size *sz*. The domain maps these fields to numeric variables $x \in \mathcal{X}_V$ that are tracked in child domains. Hence, the main purpose of the *field* domain is to translate accesses to consecutive bits in a region to accesses to a numeric variable. Note that the offset o must be constant for the *field* domain to be able to resolve it to a single variable. Reads to non-constant offsets, that is, an interval $[l, u]$, with $l \neq u$, are approximated as reading an unknown value $\top$. On writes to non-constant offsets, however, the *field* domain will remove all fields overlapping with bits $[l, u + sz - 1]$ with sz being the access size. Removing these fields also removes any numeric variables from child domains associated with these fields.
The *field* domain is also responsible for making two states that are joined compatible, that is, it introduces variables in both states so that their support sets match. For example, if domain $\mathcal{D}_1$ does not track a variable x that exists in the other domain $\mathcal{D}_2$, it is introduced in domain $\mathcal{D}_1$ as a new field with the same size and offset and the value of x mapped to $\top$.

string* Although this domain is not yet implemented it is intended to abstract string operations in programs as described in [SK02]. In particular the domain tracks the position of the first NUL-terminator in a string using a symbolic value (a variable $t \in \mathcal{X}_V$). Read and write accesses at offset a to the string are then separated in operations that lie in front, and after this NUL position. In combination with relational domains, such as octagons or polyhedra, it allows to reason about out-of-bounds memory accesses involving strings.
Note that this domain may actually use the *array* domain described next to track summaries for a string.

array* Whenever a write access has an offset o that denotes a non-constant value, that is, an interval, the *field* domain will remove all fields overlapping with that offset. Other domains such as the *array* domain are able to summarize accesses [Gop+04; CCL11] to a range of offsets into a single array cell. Figure 2.10 shows the *array* domain as a parent domain of the *field* domain which means that it can execute any operation on the *field* domain while evaluating $m \xrightarrow{sz} o$.

In order to illustrate the interaction between *array* and *field*, consider the following loop in Fig. 2.13.

```
1 struct {
2   short l, h;
3 } m[100];
4
5 for (short i = 0; i < 100; i++) {
6   m[i].l = 0;
7   m[i].h = i;
8 }
```

Figure 2.13: Iterating over an array of structs.

We represent the array `m[100]` as a memory region m that summarizes all array elements into one. The loop is first analyzed with $i = 0$ so that the *field* domain will create two fields as a response to $m \xrightarrow{sz} 0 = 0$ and $m \xrightarrow{sz} 16 = i$. The second iteration of the fixpoint computation will execute line 5 with $m \xrightarrow{sz} o = 0$ where $o \in \{0, 32\}$, that is, $o = 32i, i \in [0, 1]$. The *array* domain will observe the non-constant offset and intercept the assignment before the *field* domain removes all information on the inferred fields. Instead, it copies all fields at bits $0 - 31$ to a new summary memory region m' using an assignment operation on the *field* domain. The write operations on the ith element are then translated to an operation on m' at offset $o - 32i$ which is constant, thereby re-using the capabilities of the *field* domain, to handle overlapping fields. The access in line 6 is summarized analogously. The ability to re-use existing domains is a major benefit in this hierarchical arrangement.

code* and data content Currently this domain is not a separate domain but its semantics are part of the *field* domain. The domain tracks accesses to the content of the data segments in a binary, that is, the domain forwards read accesses to data from the binary (code segments or data segments). For read-only segments this functionality would suffice. However, write accesses may modify parts of the data thus the domain maintains a mapping from fields to a boolean marking a field as clobbered. The value of a clobbered field thus has to be retrieved from the data tracked in the *field* domain. This design allows us to have a sparse representation of all the data in the binary in that no fields need to be created upfront to hold the data in the executable.

2.5.3 Finite Domains

The memory domains associate an abstract variable $x \in \mathcal{X}_V$ with each field inferred by the *field* domain. The underlying numeric domains map x to possible numeric values. Each abstract variable represents a finite number of bits and the operations emitted by the memory region always access all bits in a variable. The grammar of the interface *L(finite)* between memory domains and the *finite* domains therefore does away with offsets and, in particular, accesses at offsets. To this end, it makes use of a production *rreil-stmt*$^{\mathcal{X}_V}$

that correspond to productions *rreil-stmt* (see Sect. 2.4.6) where the variables $\underline{id} \in \mathcal{X}_M$ are substituted with numeric variables $\underline{x} \in \mathcal{X}_V$ in each rule. The grammar in *L(finite)* adds relational tests consisting of two linear expressions.

$$
\begin{aligned}
\mathit{finite} &::= \mathit{rreil\text{-}stmt}^{\mathcal{X}_V} \mid \mathit{fin\text{-}test} \mid \mathit{fin\text{-}sup} \\
&\mid \textbf{sign-extend}\ \mathit{var}, \mathit{var} \\
&\mid \textbf{convert}\ \mathit{var}, \mathit{var} \\
\mathit{fin\text{-}sup} &::= \textbf{intro}\ sz\ \mathit{var} \mid \textbf{drop}\ \mathit{var} \\
\mathit{fin\text{-}test} &::= \mathit{lin} : \mathit{sz}\ \mathit{relop}\ \mathit{lin} : \mathit{sz} \\
\mathit{relop} &::= < \mid \leq \mid = \mid \neq \\
\mathit{lin} &::= \sum_i \underline{a_i}\underline{x_i} + \underline{c} \qquad\qquad x_i \in \mathcal{X}_V,\ a_i, c \in \mathbb{Z} \\
\mathit{var} &::= \underline{x} : \mathit{sz} \qquad\qquad x \in \mathcal{X}_V \\
\mathit{sz} &::= \underline{n} \qquad\qquad n \in \mathbb{N}
\end{aligned}
$$

Note that each variable $x \in \mathcal{X}_V$ in the *finite* domain has a fixed bit size sz. This size is specified when creating a variable using **intro**. Analogous to the requirement in RREIL that every operation must use the same size for all arguments, each operation in *L(finite)* may only use variables of equal size. Exceptions are the result of comparisons, the **sign-extend** operation and the new operation **convert**. The latter is used to strip off most significant bits or to zero-extend a variable.

We will now continue by describing each domain in the *finite* tier before we detail how the *wrapping* domain translates from variables of finite bit-sizes to variables in $\mathbb{Z}$.

undef This abstract domain improves the precision of numeric analyses in case of a changing set of active variables during the analysis. For example, in case of dynamic memory allocation or recursive procedures a memory region $m \in \mathcal{X}_M$ might not exist in all execution traces of the program. Moreover, the numeric variables $x_i \in \mathcal{X}_V$ that are associated with m are only present in some of the abstract states inferred by the analysis. Joining these states with states where the variables are not existent leads to precision loss. The *undef* domain uses flags f_{x_i} to denote if a variable is defined or not. Relational child domains of *undef* may then infer relations between the flags f_{x_i} and the valuations of x_i, thereby improving the precision of the numeric analysis in case of undefined variables. A more detailed description of the use cases and the implementation of the domain is given in Chap. 5.

flags One challenge in the analysis of binaries is that conditionals in high-level programs are translated into a test setting certain flags and a conditional jump that tests a flag. One way to link test and jump is to perform forward substitution of the flag assignments into the conditional jump [KVZ09; KSS13a]. This approach fails when the arguments of the

test are modified before the conditional jump, a situation that may arise as a result of compiler optimizations. One fix is to apply any such modification to the relational test that is being propagated forward. Our analyzer pursues this approach in a less ad-hoc way by tracking all observed tests in a symbolic abstract domain dubbed *flags* in Fig. 2.10. In order to illustrate this process, consider the following RREIL statements, showing a comparison followed by a modification to the compared register and a jump using the result of the comparison:

```
0 cmpltu CF:1, eax:32, 100:32
1 add    eax:32, eax:32, 1:32
2 brc    CF:1, loop:32
```

The purpose of the *flags* domain is to track assignments to flags symbolically, thus, it stores $CF \equiv eax < 100$ after the first statement. The second statement increments a variable that is mentioned in the symbolic expression, thereby rendering this expression invalid. Rather than removing all information, any linear assignment and transformation is simply applied to the symbolic information, leading to $CF \equiv eax < 101$ in the example. Once the conditional jump leads to test $CF = 1$ on the memory domain, the predicate domain also executes the test $eax < 101$ on its child domain, thereby making the effect of the indirect test explicit. Analogously, to find out if the else-branch is feasible, that is, if the negated flag test $CF = 0$ is true, our domain applies the negated comparison $101 \leq eax$ on the child.

points-to This domain is an implementation of a flow-sensitive *points-to* domain as discussed in [Simo8a, Sect. 4.1]. The domain observes assignments of pointers to variables and tracks a set of all possible pointers that a variables may contain. Pointers are modeled using address variables $a \in \mathcal{X}_A \subseteq \mathcal{X}_V$ that denote symbolic addresses. These variables are initialized with a large numeric range, e.g. $[0, +\infty]$, thus effectively disallowing address arithmetic involving symbolic addresses. The *points-to* domain resolves accesses involving offsets to address variables by returning a set of tuples: $\{\langle r_i, o_i \rangle\} \in \wp(\mathcal{X}_V \times \mathbb{N})$. This set approximates a pointer dereference as offsets into memory regions.

Whenever the domain observes an assignment of the form $x = a + o$ with $x \in \mathcal{X}_V$ and a being an address, it stores the information that $x = fa$ where $f = 1$ is a new Boolean variable stored in the child. The pointer offset o is a numeric value that is stored in the child, e.g. the *interval* domain as $x = [o, o]$. In general, the *points-to* domain tracks linear combinations of addresses and flags $f_i \in \mathcal{X}_F$ of the form $x = \sum_i f_i a_i$ for each variable that may contain pointers. Hence, let $s_x = \sum_i f_i$ with $0 \leq s_x \leq 1$ express that x points either to one single address a_i or to none. The `NULL` pointer is modeled by $f_i = 0$ for all flags f_i associated with a variable. Testing if an access is valid thus reduces to testing if $s_x = 1$.

The benefit of the approach using flags can be seen when joining two states. Assuming that in one state $x = f_1 a_1$ and $f_1 = 1$ and in the other state $x = f_2 a_2$ with $f_2 = 1$, the join of the states is $x = f_1 a_1 + f_2 a_2$. With $s_x = f_1 + f_2 = 1$ this exactly captures the semantic that x points either to a_1 or a_2 but not both nor to none. When dereferencing x the *points-to* domain is queried and the flags f_i are returned. The child can now be partitioned so that

in each state $f_i = 1$ and $f_j = 0$ for $j \neq i$. This expresses that only one address is a valid pointer for x in the state.

SAT The *SAT* domain is designed to observe boolean variables (flags) and track formulas between them (in CNF form). On joining two states, the domain computes a new boolean formula that has the sum as the disjunction of the two formulas. Furthermore, the domain manipulates tracked formulas according to the transfer function. The main use case is to infer boolean formulas between the flags $f_i \in \mathcal{X}_F$ of the *points-to* domain, thus precisely describing the inferred points-to sets. Instead of using numeric variables to track the sum of the flags, one can precisely express the value of the flags using a boolean formula.
The domain uses MiniSat [EMS07; EMA10] as the back-end for satisfiability checking.

wrapping In order to infer strong relational invariants, such as inequality relations between variables, we translate operations over $\mathbb{Z}_{2^w}$ to abstract domains over $\mathbb{Z}$ on which many expressive numeric domains have been defined. However, an integer represented by a string of w bits can be interpreted as a signed (in the range $[-2^{w-1}, 2^{w-1} - 1]$) or unsigned (in the range $[0, 2^w - 1]$) value in $\mathbb{Z}$. Since some operations such as **add** and **shl** carry no signedness information, it is not possible to simply check for overflow after each assignment. Thus, we interpret each value $v \in \mathbb{Z}$ of $x \in \mathcal{X}_V$ in the numeric domain as $v \mod 2^w$ in the finite domain, assuming that x was introduced as having w bits. In this interpretation, no checks for overflow are required for linear numeric transformations [SK07]. Linear transformations, in turn, are exactly those assembler instructions that carry no sign information. As an example, consider an unsigned 4-bit variable $x \in \mathcal{X}_V$ for which the numeric domain tracks an interval $[13, 15]$ corresponding to the finite interpretations $1101_b, 1110_b, 1111_b$. A left-shift by one bit is a multiplication by two and leads to the interval $[26, 30]$. Interpreting these values $\mod 2^4$ yields the bit patterns $1010_b, 1011_b, 1100_b, 1101_b, 1110_b$ which is a sound approximation of the possible set of bit-patterns. Note that the interval approximation introduced spurious values: 27 and 29, which correspond to the second and fourth bit pattern.

The *wrapping* domain proceeds as follows: The arguments of tests (e.g. $x < 10$) and other operations whose semantics depend on sign information (e.g. **shr** and **shrs**) are checked if their values lie in the range of the corresponding signed or unsigned w-bit integer. If not, they need to be adjusted which might involve further approximation [SK07]. These translations are performed by the *wrapping* domain by executing required operations on the *zeno* child domains. To illustrate this, consider Fig. 2.14 which shows the value range of one variable, namely x, in a one-dimensional plot assuming that x is of size 32 bits and unsigned. As the values of x do not lie inside the unsigned range $[0, 2^{32} - 1]$, the *wrapping* domain will adjust the values of x by issuing the following operations on the child domain state $c \in \mathcal{C}$: $[\![x \leq 2^{32} - 1]\!]^{\mathcal{C}} c \sqcup_{\mathcal{C}} [\![0 \leq x]\!]^{\mathcal{C}} [\![x = x - 2^{32}]\!]^{\mathcal{C}} c$. These operations shift values of x inside the unsigned 32 bits range "quadrant" (thus implementing the modulo operation) and join them with the values that already lie inside this range. This method is performed for all quadrants $[k2^{32}, (k+1)2^{32} - 1]$ with $k \in \mathbb{Z}$ that overlap with values of x. Moreover, this method can be generalized for a state space in n dimensions,

i.e. for n variables [SK07].
Note that the diagram in Fig. 2.14 shows the precise value range of x after wrapping, which is non-convex. Most numeric domains, however, use a convex approximation of the state space, thus joining the two partial ranges will result in the numeric approximation $x \in [0, 2^{32} - 1]$, incurring a loss of precision.

Figure 2.14: How wrapping adjusts value ranges.

We chose to implement wrapping as a separate cofibered domain in order to combine it with other numeric domains. The straight forward and most common approach is to incorporate wrapping as an operation in the numeric domain, e.g. affine equalities [MS05], congruences [Byg10] or intervals [Min12]. This approach yields more efficient implementations but also complicates the implementation and requires modifications to each domain that is not oblivious to wrapping. Especially in the setting of our modular domain hierarchy that allows to try out new numeric domains, e.g. by using off-the-shelf libraries, it would require to enhance existing domains to implement wrapping. Thus a benefit of our approach to implement wrapping as an abstract domain is that abstract domains in *zeno* do not need to know about the bit-sizes of variables.
In summary, our approach might be less precise as it cannot fully exploit domain specific knowledge for each abstract domain but is general and adds wrapping to already existing domains. Moreover, the implementation of numeric domains is much easier when they operate over $\mathbb{Z}$.

2.5.4 Zeno Domains

The domains discussed next are called *Zeno* domains because the grammar they operate on does not track sizes for variables and the domains operate on variables ranging over $\mathbb{Z}$. The interface to the domains features standard operations on variables and is given by the language *L(zeno)*, defined as follows:

$$
\begin{aligned}
\textit{zeno} &::= \textit{zeno-stmt} \mid \textit{zeno-test} \mid \textit{zeno-sup} \\
\textit{zeno-stmt} &::= \textit{lval} = \textit{lin binop lin} \\
&\mid \textit{lval} = \textit{lin} \,/\, d && d \in \mathbb{N} \\
&\mid \textit{lval} = [l, u] && l, u \in \mathbb{Z}_\infty \\
\textit{zeno-test} &::= \textit{lin relop } \underline{0} \\
\textit{zeno-sup} &::= \textbf{intro}\ \textit{lval} \mid \textbf{drop}\ \textit{lval} \\
\textit{binop} &::= \times \mid \div \mid \% \mid >> \mid << \mid \& \mid \hat{} \mid \mid \\
\textit{relop} &::= \leq \mid = \mid \neq \\
\textit{lval} &::= \underline{x} && x \in \mathcal{X}_V \\
\textit{lin} &::= \sum_i \underline{a_i}\underline{x_i} + \underline{c} && x_i \in \mathcal{X}_V,\ a_i, c \in \mathbb{Z}
\end{aligned}
$$

The language is comprised of statements, tests and of instructions to modify the set of mapped variables, i.e. **intro** and **drop**. A statement sets a variable to the result of a binary operation, a linear operation with divisor d or an interval. No bit-level binary operators are allowed as they cannot reasonably be expressed using numeric domains that model a convex state space. An interval approximation is used when translating bit expressions to arithmetic in numeric domains, e.g. **and** $x, y \equiv 0$ if $x = [0, 0]$ or $y = [0, 0]$. Common bit-level identities such as **xor** $x, x \equiv 0$, are identified in the *wrapping* domain and simplified. Tests in *zeno* consist of a linear expression and a relational operator that compares the expression to 0. These tests are translations of tests in *L(finite)*; the *wrapping* domain translates finite tests "l_1 *relop* l_2" by wrapping each of the linear expressions l_1, l_2 to the corresponding bit-size and then applying "$l_1 - l_2$ *relop* 0" on the child. As *zeno* domains operate on integers, the operator $<$ is expressed by $\leq$ and a subtraction of 1 from the linear expression. Our linear expressions consist of variables and integer coefficients plus a constant. As we deal with arithmetic in $\mathbb{Z}$ all integers are of arbitrary precision. The *wrapping* domain translates linear expressions from *L(finite)* to *L(zeno)* by wrapping the whole expression to the corresponding bit-size. If the interval approximation of the linear expression lies in the corresponding range, wrapping is a no-op. Otherwise, the expression l is assigned to a temporary variable t by applying "$t = l/1$" on the child. This temporary variable is wrapped using the operations described in the previous paragraph 2.5.3. Due to the assignment, a relational child domain approximates the operations on the wrapped variable t rather than on the whole linear expression l.

The simplicity of the numeric interface, mainly consisting of assignments and tests, aligns well with existing implementations of numeric domains in the literature. Furthermore, the benefit of working on arbitrary size integers is that transfer functions can be implemented resembling their mathematical definition.

We will describe next a set of symbolic domains, that operate on predicates over variables.

These domains are not numeric domains in the classic sense of tracking values for variables, however, the simplicity of *L(zeno)* lends itself to implement these domains at this level. After that, we will discuss the convex numeric abstractions known from the literature that are implemented in our analyzer as abstract domains.

widening strategies We implemented various widening strategies from the literature as zeno abstract domains. This allowed us to try out new widening heuristics without complex changes to the fixpoint engine. The strategies we implemented are delayed widening, inference of widening points, widening with thresholds and guided widening. The benefit of implementing widening as an abstract domain is that we are able to add or remove widening heuristics to trade scalability for precision. The implementation demonstrates that cofibered domains are powerful enough to implement even very complex widening strategies.These domains and further strategies are described in detail in Chap. 3.

predicate This abstract domain started as an extension of the *flags* domain. The *predicate* domain generalizes the association of tests with flag variables to implications between predicates. Additionally, the domain adds an entailment mechanism between predicates, thereby implementing a lightweight predicate abstraction and reduction mechanism. Moreover, by using only the existing numeric transformers on the child, the domain is usable as a modular addition to existing numeric domains.
The *predicate* domain tracks implications over predicates, e.g. $x < 5 \rightarrow y < 10$ to refine the numeric state of its child by applying the consequence of an implication whenever it can infer that the premise holds. Hence, the domain adds relational information to non-relational child domains. Furthermore, the *predicate* domain also supplements numeric domains with the ability to express non-convex invariants. This powerful feature requires observing the precision loss that occurs in convex numeric domains, e.g. during joins. From the precision loss, the domain synthesizes predicates to counteract the approximation common in numeric domains without costly replication of numeric states. The *predicate* domain thus serves as a lightweight disjunctive domain. A more detailed description of the features and usage of the domain is given in Chap. 4.

Numeric Domains

Tracking the values for the variables in a program is the task of numeric domains. A numeric domain can be seen as a concise representation of a set of values. Simple, non-relational domains map single variables to values. An example of such a simple domain is the *interval* domain. More sophisticated domains relate the values of some or all variables, so that a test on one variable will restrict the value of other variables. In our framework we implemented a set of numeric abstract domains from the literature and used existing domain libraries where possible. In the following we will describe in more details the numeric abstract domains that our framework uses and where their strengths and weaknesses lie.

affine The *affine* domain tracks equality relations of the form $\sum_i a_i x_i = c$ between variables as originally described in [Kar76]. Here, $a_i \in \mathbb{Z}$ are coefficients, $c \in \mathbb{Z}$ the constant

term and $x_i \in \mathcal{X}_V$ are numeric variables. In general, the domain has the complexity of $\mathcal{O}(n^3)$ as it stores equalities as a matrix over the variables $\mathcal{X}_V$. Our implementation is a variation that tracks equalities in the normal form of an upper-triangular matrix using some fixed total ordering on the variables. When calculating the join of two domains with n rows, the normal form makes it possible to extract only the m equalities that differ, which is possible in $O(m)$ rather than $O(|\mathcal{X}_V|)$ [Bla+03a]. Since binary programs exhibit many equality relations (e.g. between registers and fields on the stack, see Fig. 2.16) our implementation improves the performance of the analysis.

The *affine* domain can also be seen as a more advanced constant propagation domain [WZ91]. The domain propagates constants and equalities between variables and additionally abstracts constants by inferring equality relations between variables during joins. Consider the example in Fig. 2.15 a) which is motivated by the Sendmail code presented in Sect. 9. After line 2 the *affine* domain tracks the constant values $pos = 5$ and $len = 10$. At line 6 we know that $pos = 1$ and $len = 9$. At the join point in line 8, the values are combined by computing the the affine hull of both states, here, the equality relation: $pos + 35 = 4len$. In the Sendmail code analysis described in Sect. 9, a similar equality relation is used to prove that a buffer overflow cannot occur.
Besides tracking constants, the *affine* domain further unfolds its usefulness when combined with other numeric domains, such as the *interval* domain described in Sect. 2.5.4. In Fig. 2.15 b) we have an equality relation between the variables x and y after the assignment in line 1 without knowing the values. Using the *interval* domain the test in line 3 results in $x \in [-\infty, 9]$. Moreover the *affine* domain tracks $x = y$ so that we are able to infer $y \in [-\infty, 9]$, which would not be possible with the *interval* domain alone.
Now lets consider the last example in Fig. 2.15 c) which combines the two previously mentioned features of the *affine* domain. By performing the affine closure on the equalities during the join, the *affine* domain infers the loop invariant $j = k + 3i + 2$ that also holds after the loop in line 7. At line 8 we infer new bounds for the variables i and k. With the equality tracked in the *affine* domain we are able to infer that $j \in [-\infty, 37]$ also holds. Combining the *affine* domain with the *interval* domain, however, is not always straightforward. Though the *affine* domain is useful to improve the values tracked in the *interval* domain, it does not always lead to an improved precision. We now discuss the benefits and disadvantages that are introduced by the *affine* domain.

We provide two slightly different implementations of the domain. The regular *affine* domain implementation removes redundant information about variables from its child domains, thus minimizing the information that needs to be stored in the child. For example, if an equality $x = y + z + 5$ is stored in the *affine* domain then the variable x is fully determined and thus does not have to be stored in the child domain, leading to a smaller support set for the underlying domains. This factoring trick also simplifies many linear operations. For instance, the assignment $x = x + 1$ is evaluated by inlining existing equalities into the right-hand side, yielding $x = y + z + 6$, and replacing the previous definition of x. Since x is only known in the *affine* domain, no operation on the child domain is necessary. Besides having fewer operations on the child domain the simplifications performed in the *affine* domain also improve the precision when the child domains are not

a)

```
pos = 5;
len = 10;
...
if (...) {
  pos = 1;
  len = len - 1;
}
...
```

b)

```
x = y;
...
if (x < 10) {
  ...
}
```

c)

```
i = 2;
j = k + 5;
while (...) {
  i = i + 1;
  j = j + 3;
}
...
if (i < 10 && k < 5)
  ...
```

Figure 2.15: Example usages of affine equalities.

closed under certain arithmetic operations or lose precision on some arithmetic operations [Mino6b]. Consider for example the assignment $y = 2z - z$ on the domain $\mathcal{A} \triangleright \mathcal{I}$, that is, using the *interval* domain as child domain of the *affine* domain. The value of z in the *interval* domain is $z \in [0, 10]$. Performing the assignment on the intervals would result in $y \in [0, 20] -_{\mathcal{I}} [0, 10] = [-10, 20]$ which introduces imprecision. Simplifying the assignment in the *affine* domain results in $y = z$ which is more precise when evaluated on the intervals. In fact the value of y would be removed from the *interval* domain since an equality $y = z$ is stored in the affine domain.

redundant affine Unfortunately, not propagating redundant information to the child domain might also incur a loss of precision. An observation that has motivated our second implementation, the so-called *redundant affine* domain that performs all the linear optimizations mentioned above but does not remove fully determined variables from the child. A simple example will illustrate why not storing values for fully determined variables causes a loss of precision. Consider again the *affine* domain as it was presented above with the *interval* child domain storing $x \in [1, 2]$ and $y \in [0, 10]$. Executing the assignment $z = x + y$ on the child will result in the addition of $z \in [1, 12]$ to the tracked intervals. Additionally, the *affine* domain now tracks the above assignment as a new equality in the form $x = -y + z$ due to the internal variable order. In this example we use the lexical ordering of variables, i.e. x comes before y and z. In a second step, because variable x is now fully determined by the other variables on the right-hand side of the equation, it can be removed from the child domain which will then only store $y \in [0, 10]$ and $z \in [1, 12]$. Whenever the value for x needs to be retrieved, the *affine* domain will inline the known equalities and evaluate the linear expression in the *interval* domain thus resulting in $x \in -[0, 10] +_{\mathcal{I}} [1, 12] = [-9, 12]$. This value for x is less precise than the original value $[1, 2]$ that was tracked for x before the assignment. Although it seems that the variable ordering in the *affine* domain causes the precision loss, it is not possible to find a perfect ordering that does not introduce a precision loss for a given set of assignments. The problem is that the affine domain performs linear operations on the child to normalize its internal state. The *interval* domain, however, uses

interval arithmetic to implement these linear operations, which are inexact.

To solve this issue we introduce a modified implementation of the domain that never removes variables from its child. In the above example this *redundant affine* domain would still store $x \in [1, 2]$ in the *interval* domain after evaluating the assignment. Additionally, when querying the value for x the *redundant affine* domain does not inline known equalities, i.e. $x = -y + z$ to produce the result but will query the value from its child, and return the precise value $x \in [1, 2]$. However, in order to keep the benefits of arithmetic simplifications, the *redundant affine* domain uses equalities for improving the *interval* domain. We implement this reduction by deriving and applying additional tests based on inlining equalities into assignments and test operations. Consider applying the test $x < 2$ to the state $s = \langle\{x = y - 1, y = z\}, \{x \in [0, 2], y \in [1, 3], z \in [1, 3]\}\rangle$. By inlining the equalities tracked in the state of the *affine* domain, the test not only restricts variable x to $[0, 1]$ but also y to $[1, 2]$ and z to $[1, 2]$. In detail, we inline the equality $x = y - 1$ resulting in the test $y < 3$ and further inline the equality $y = z$ resulting in the test $z < 3$. Applying these tests on the *interval* domain yields the more precise result: $[\![x < 2]\!]^{\mathcal{A}} s = \langle\{x = y - 1, y = z\}, \{x \in [0, 1], y \in [1, 2], z \in [1, 2]\}\rangle$. Note that this causes more tests to be applied on the child as compared to the regular *affine* domain implementation. The obvious reduction is to apply all equalities that share variables with a test or an assignment as new tests on the child domain. However, we want to have something cheaper than applying $|\mathcal{X}_V|$ tests. The idea to evaluate only the inlined assignment may unfortunately introduce imprecision as illustrated next with an example.
Consider again the domain $\mathcal{A} \triangleright \mathcal{I}$ using the *redundant affine* domain and the state $s_{xor} = \langle\{x = y + z\}, \{x \in [1, 1], y \in [0, 1], z \in [0, 1]\}\rangle$ which expresses that y and z are either 0 or 1 but cannot have the same value–the *exclusive-or* of the variables. Now consider the *redundant affine* domain seeing the assignment $a = x - 1$. Inlining the known equalities and executing the resulting assignment $a = y + z + 1$ on the child would result in the value $a \in [-1, 1]$ whereas the non-inlined assignment results in the value $a \in [0, 0]$. Because inlining can lead to a blowup of the amount of variables in an assignment it may be less precise than not inlining assignments.

Consequently, a solution is to apply either the original or the inlined assignment depending on which results in a better precision. We use a simple heuristic to decide which assignment is more precise when executed on the child. The heuristic is derived from the fact that each term in interval arithmetic can introduce a precision loss, so we choose the assignment containing fewer variables.

Usage of Affine Equalities in Machine Code

Tracking affine equalities is a must for the analysis of binary code as register spillage, i.e. moving register values to the stack and later back to registers, introduces many equality relations. Optimizing compilers can help to reduce the amount of such equalities but we need to be able to also analyze non-optimized code. Specifically it is important to

propagate values of variables that have relations to other variables. Consider the simplified code example in Fig. 2.16 a) that shows a pattern that occurs often in the context of register spilling.

a)

```
1 r1 = stackvar;
2 if (r1 < 5) {
3   r1 = ...;
4   r2 = stackvar;
5   ... // computations using r2
6 }
```

b)

```
1 r1 = 0;
2 r2 = &array;
3 do {
4   ... = *r2;
5   r2 = r2 + 4;
6   r1 = r1 + 1;
7 } while (r1 <= 16);
```

Figure 2.16: Common compiler code patterns: a) register spilling to the stack and b) usage of temporary registers for computations.

The pseudo-code illustrates reading a stack-allocated variable $stackvar$ into the register $r1$. After entering the **if**-branch we know that $r1 < 5$ but without affine equalities we would not be able to refine the value tracked for $stackvar$. Thus when $stackvar$ is loaded to $r2$ and its value is subsequently used in line 5, we cannot infer the invariant $r2 < 5$ even though $r1 = r2 = stackvar$ holds.
Now consider the code in Fig. 2.16 b) that shows the necessity of affine equalities when temporary registers are used. Here, $r1$ is the loop counter and $r2$ is used to index into an array. As the array uses 4 bytes per entry the compiler will use a temporary register $r2$ and increase the value of that register in each iteration (line 5) along with the loop counter (line 6). However, the loop condition in line 7 tests the loop counter $r1$ and not the array index variable $r2$. Thus, after inferring the value range for the loop counter $r1 \in [0, 16]$ it is necessary to propagate this range information to $r2$ otherwise we cannot show that the array access is within bounds. Here, the affine relation $4r1 = r2 - \&array$ that holds inside the loop body can be used to refine the value of $r2$ and prove that the array access is bounded.

In summary, without an *affine* domain the precision of analyses on binary code is insufficient. Our experiments immediately confirmed this. Combining the *affine* domain with other numeric domains can still result in precision loss if the child domains are incomplete under linear operations. To overcome these deficits we implemented the *redundant affine* domain. Nevertheless, this implementation loses some of the speed benefits of the original as it results in a multiple application of tests to keep the child domain reduced. We also observed that not storing the child variables that are fully determined increases the analysis speed and improves memory consumption especially when using other relational domains or costly domains as child. In conclusion, which of the two *affine* domain implementations to use remains a trade-off between precision and performance.

gauge* The *gauge* domain as defined in [Ven12] approximates loop invariants for variables by associating them with the values of loop counters. It tracks for each variable, $x \in \mathcal{X}_V$ an upper and lower bound as affine inequalities of the form $\pm x \leq c + \sum_i a_i \lambda_i$ where $c, a_i \in \mathbb{Z}$ and $\lambda_i \in \mathcal{X}_C$. The set $\mathcal{X}_C \subseteq \mathcal{X}_V$ represents the set of the loop counters in the program. Geometrically, the domain approximates the values of a variable using a wedge or cone (see Fig. 2.23) that limit the range of values in terms of loop counters. Note that the domain does not infer loop bounds for the counters but needs to be combined with other numeric domains, like the *interval* domain, for that.
As an example, consider the code in Fig. 2.17 which iterates over a message buffer using a pointer. The loop iterations depend on the type of messages in the buffer. The message type is given by a tag that is read from the current datum in the buffer in line 3. Analyzing the code with the *gauge* domain yields the state $s_g = \{\lambda \leq i \leq \lambda, 16\lambda \leq p \leq 32\lambda\}$ with loop invariants for the variables i and p. Here, the variable i is itself the loop counter thus equal to λ. To infer the loop bound $i < n$ we need to combine the *gauge* domain with a relational domain, e.g. *octagons*, as the loop bound is symbolic and thus the *interval* domain cannot express this invariant.

```
1  p = &msg;
2  for (i = 0; i < n; i++) {
3    if (*p == ...) {
4      ...
5      p += 16;
6    } else {
7      ...
8      p += 32;
9    }
10 }
```

Figure 2.17: Iterating over the contents of a message buffer depending on the message.

As demonstrated the purpose of the *gauge* domain is to infer conditional loop increments. Combined with another numeric domain such as the *interval* domain, the *gauge* domain is a scalable replacement for more expensive relational domains used in loop analyses, e.g. polyhedra. Its complexity is comparably low, namely: $\mathcal{O}(k|\mathcal{X}_V|)$ as all operations depend on the number of nested loop counters $k \leq |\mathcal{X}_C|$. The domain achieves this low complexity by assuming that the λ_i variables can only contain positive values.
We did not finish the implementation of the domain yet. In particular one issue we had is the automatic discovery of a good set of loop counters λ_i as this is not trivial at the machine code level. On the source code level, loop counters can be extracted syntactically from the loop constructs whereas we need to infer variables for which the values change by a constant increment.

congruences This domain implements the inference and tracking of non-relational or arithmetic congruences as suggested in [Gra89]. Congruence information is of the form

$x \equiv b \pmod{a}$ with $a, b \in \mathbb{Z}$ and $x \in \mathcal{X}_V$. It is sometimes also written as $a\mathbb{Z} + b$ because it describes a set of numbers $\{an + b | n \in \mathbb{Z}\}$ that are multiples of a plus an offset b. Congruences form a grid in the numbers space (see Fig. 2.23). Congruence information is necessary for example to ascertain that an array access is aligned to the element boundary [BR04]. One application is the precise tracking of indices into jump tables [Mih09]. The original intention of the domain in [Gra89] was to help compilers perform automatic vectorization of computations. As the value space described by congruences is unbounded the domain is mostly used in a reduced product with other domains, such as the *interval* domain. This particular combination is sometimes called "strided intervals" [RBL06].

Consider for example the code in Fig. 2.18 a) which accesses and array at certain indices. Using congruences we can improve the inferred bounds for variable `i` inside the loop to $i \in [0, 96]$ instead of the less precise result $i \in [0, 99]$ that would be inferred by using an interval analysis alone. Additionally, congruences allow us to express that only each 4^{th} element of the array is accessed. The code in Fig. 2.18 b) shows how compilers usually translate **`switch`**-tables to machine code. An index into the switch jump table `i` is computed from an input value and then used to index into an array of jump addresses. Here, congruence information helps to extract only valid jump addresses from the jump table even when the value for `i` is not constant as it restricts the possible indices to multiples of 32.

a)

```
1 int a[] = ...
2 int i = 0;
3 while (i < 100) {
4   ... = a[i];
5   i = i + 4;
6 }
```

b)

```
1 unsigned i = ...
2 if (i < 10) {
3   i = i * 32;
4   goto *(table + i);
5 }
```

Figure 2.18: Precision improvements using congruence information in interval analysis.

Apart from their usage in combination with intervals, congruences can sometimes prove invariants that convex domains cannot. Consider a similar example as in [Cou01], where the analysis inferred the congruences: $x \equiv 2 \pmod{3}$ and $y \equiv 3 \pmod{6}$ at a certain program point, i.e. $x \in \{\ldots, -1, 2, 5, 8, 11, 14, \ldots\}$ and $y \in \{\ldots, -3, 3, 9, 15, 21, 27, \ldots\}$. Using this information we can prove that the division $1/(x - y)$ can never fail, that is, the divisor being equal to 0 because there are no numbers to satisfy $x = y$ given the congruences for x and y. A convex approximation as the polyhedra domain would infer, is not able to exclude the line of points given by $x = y$ thus is less precise in this case.
Another common application of congruence information is the analysis of computations that align a pointer. Pointers are expected to be aligned to a certain memory boundary. For example GCC on x86-32 architecture aligns the stack pointer `esp` on a 16 bytes boundary by either using a combination of shift instructions: `mov eax, esp; shr eax, 2; sal eax, 2;`

or by using a bitmask: `and esp, 0xfffffff0`. Knowing the multiplicity of a variable–its congruence–we can perform these operations without losing precision.

Our *congruence* domain is an implementation of the non-relational (or arithmetic) congruence analysis introduced in [Gra89]. Inferring relational (or linear) congruence analysis as described in [Gra91; Gra97] requires a more complicated implementation which also negatively impacts scalability. The simple congruences have a complexity of $\mathcal{O}(n)$ (with n the number of tracked variables) whereas the linear congruences are $\mathcal{O}(n^4)$ with some potential for optimizations [Bag+07]. Moreover, for linear relations between variables we use the *affine* domain and combine it with congruences. Alas, linear congruences are more powerful than the product of *affine* and *congruences* (see Sect. 2.6). Hence, an implementation thereof remains interesting future work. One possibility is to reuse already existing implementations such as the linear congruences in the *Parma Polyedra Library* [Bag+07]. However, that would require modifications to the domain for it to be usable as a cofibered domain.

Leaf Domains

At the bottom of our domain hierarchy we have a number of so-called leaf domains that do not have a child domain. These domains are either non-relational and simple in that they map variables to values or they are from external libraries that were not designed for the cofibered construction.

Apron

We use the library of abstract domains from the *Apron* [JM09] project to augment our existing numeric domains stack. The library provides implementations for a number of numeric domains and a unified high-level API to the domains. We implemented an adapter for this high-level domain interface allowing us to use any of their abstract domain implementations in our analyzer. Notice however that Apron is able to use real or floating point numbers in their domains, whereas we are limited to integers due to the compatibility with our other numeric domains. We use the library as a black box through its API. Thus, in our domain hierarchy it is a leaf domain that does not have a child domain. In any case, the Apron domains are not designed for such a cofibered construction.

To improve the precision when combining Apron numeric domains with our domain hierarchy, we implemented the *synthesized channel* for Apron which propagates information about the domain changes upwards after a transfer function (see Sect. 2.6.5). However, as the channel is implemented only in code wrapping the Apron domains the performance is not optimal as it does not have access to the set of changed variables during transfer functions or lattice operations. On the other hand, having the implementation in the adapter enables us to handle all Apron domains with one implementation. There is no need to reimplement a *synthesized channel* for each Apron numeric domain.

The numeric domains in Apron also implement a different semantic for division by zero than our domains. Apron returns 0 for division by the constant 0 and when the divisor is not a constant but contains the 0, e.g. $[-5, 10]$, the result is $\top$. For the sake of consistency

we implemented a workaround in the adapter for the Apron domains to give it the same semantics as our *interval* numeric domain.

The following two domains are part of Apron and have been evaluated in our framework. Hence, we will describe their benefits and use cases.

polyhedra This domain tracks a set of inequalities of the form $\sum_i a_i x_i \leq c$ with $a_i, c \in \mathbb{Z}$ and $x_i \in \mathcal{X}_V$, which represent convex polyhedra, as first described in [CH78]. Currently, *polyhedra* is the most general and expressive relational numeric domain. However, the expressive power comes with a high runtime cost. The domain is exponential in the number of variables. This makes it a bad fit for binary analysis where we have to deal with many more variables than in a source code analysis.

Nevertheless, the power of polyhedra to reason about symbolic values is necessary whenever one abstracts over program inputs or performs modular analyses where parameters are unknown. For example in Fig. 2.19 a) an analysis using the *polyhedra* domain infers, among others, the loop invariants in line 4: $0 \leq index < arrayLength$ which are necessary to prove that the loop does not access the array outside of its bounds. As we do not have a numeric value for the variable `arrayLength` it is necessary to symbolically reason about its relations to other program variables. The analysis of the loop in Fig. 2.19 b) infers the loop invariants: $0 \leq j \leq i \leq j + 3 \wedge j \leq 4$ for line 4. Although the initial values for all variables are known in this case, the approximation in simpler numeric domains such as intervals is too imprecise to infer this loop invariant.

a)

```
1  int l = 0;
2  int h = arrayLength - 1;
3  int index, value;
4  while (l <= h) {
5    index = (l + h) / 2;
6    value = array[index];
7    if (value == searched)
8      return index;
9    else if (value < searched)
10     l = index + 1;
11   else
12     h = index - 1;
13 }
```

b)

```
1  int i = 0;
2  while (i < 4) {
3    int j = 0;
4    while (j < 4) {
5      i = i + 1;
6      j = j + 1;
7    }
8    i = i - j + 1;
9  }
```

Figure 2.19: Example for invariants requiring polyhedra: a) is the binary search algorithm (example from [FL09]) and b) is a loop example from [HH12]

There exist specialized implementations of polyhedra, e.g. TVPI [SKH03] or octagons that improve the scalability by limiting the expressiveness. However, we use the general polyhedra implementation from the *Apron* library within our analyzer. Although Apron

is able to interface with the *Parma Polyhedra Library* [Bag+02] they provide an own polyhedra domain, called NewPolka which we use. However, the team around the *Parma Polyhedra Library* (PPL) show in [BHZ08] that the PPL is the most efficient in a benchmark that compares the commonly available polyhedra libraries. It is easily possible to switch the backend of Apron to use PPL instead of NewPolka in our implementation and we performed some experiments using PPL. However, as we did not perform a thorough comparison between the two and our need for polyhedra was limited the fact that NewPolka is shipped with Apron makes it the default choice.

Where the *polyhedra* domain becomes prohibitively expensive–which happens fast when tracking many variables–we rely on another relational domain that comes with Apron, which is described next.

octagons A much faster but also less expressive, a so-called *weakly relational* domain, is the *octagons* domain [Min01; Min06c]. It is able to relate at most two variables, expressing invariants of the form $\pm x_i \pm x_j \leq c$ with $x_i, x_j \in \mathcal{X}_V$, $i \neq j$ and $c \in \mathbb{Z}$. In 2 dimensions the state space is a subspace of polyhedra with at most 8 vertices, giving the name to the domain. However, octagons allow only unitary coefficients -1, 0, 1 for the variables. A more expressive domain are *octahedrons* [CC04] which relates more than two variables but is also more expensive.
Although the *octagons* domain provides symbolic reasoning like polyhedra with a much better performance, it is still very expensive for programs containing more than one hundred variables. The algorithmic complexity of the domain is $\mathcal{O}(n^3)$ (with n the number of tracked variables). One approach to improve scalability is to partition the variables in so-called packs [Bla+03b]. We could implement this ideas by using a disjunctive domain that tracks several separate child domains. However, finding a good packing requires up-front knowledge about the program structure, such as which variables are used in the same expression. In binary analysis, these program features are discovered only during the analysis, thus making an up-front partitioning difficult. Similar reasons motivated in [VB04] the development of a dynamic variables partitioning strategy, which should be considered in future work.
Our current improvement is to use the *affine* domain on top of *octagons* to factor out variables that are fully determined by affine equalities. In practice we were able to halve the number of variables that need to be tracked by the *octagons* domain. Note that using the *redundant affine* domain with *octagons* would not have this benefits. Additionally, the *octagons* domain has infinite ascending chains and stores redundant information, leading to non-termination if a strong closure is performed during widening. We were able to reproduce this behavior using the *redundant affine* domain on top of the *octagons* domain as the former performs a closure-like reduction. Thus, the *redundant affine* domain should not be used in combination with *octagons*.
Octagons provide a good replacement for polyhedra. For example the invariants in Fig. 2.19 are expressable with octagons, too. However, due to the restriction of being only able to track relations between two variables and the unitary coefficients, octagons are very limited.

Consider the example in Fig. 2.20 a) which is from [LJG11]. The loop invariant $i + 2j = 20$ in line 3 is not representable by octagons as the variable coefficient of j is not unitary. The code in Fig. 2.20 b) has the loop invariant: $i + j + k = 20$ which is not representable by octagons because it involves 3 variables. Note that the *octagons* domain is less expressive than the *affine* domain as the latter can track equalities over arbitrary many variables and is able to infer the loop invariants in both examples.

a)

```
1 int i = 0;
2 int j = 10;
3 while (i <= j) {
4   i = i + 2;
5   j = j - 1;
6 }
```

b)

```
1 int i = 0;
2 int j = 10;
3 int k = 10;
4 while (i <= j) {
5   i = i + 2;
6   j = j - 1;
7   k = k - 1;
8 }
```

Figure 2.20: Example for loop invariants that cannot be expressed by the octagons domain.

Apron also provides an experimental implementation of the *zonotopes* abstract domain [GGP09] which is more precise than octagons at a slightly higher runtime. However, we did not yet include it in our framework as the necessary Java wrappers in the Apron library are not present due to its experimental character.
Besides the mentioned relational domains, Apron also provides an implementation of the non-relational *interval* domain that is called Box. Comparing it to our own implementation it was 2-3 times slower. We did not investigate why, but guess that the overhead of native library calls weighs in, whereas our Java implementation can be optimized better by the Java virtual machine.

intervals Our implementation of the classic *interval* domain [CC76] tracks non-relational inequalities of the form $\pm x \leq c$. That is, it tracks an interval $x \in [l, u]$ for every variable in its support set $x \in \mathcal{X}_V$ where $l \in \mathbb{Z} \cup \{-\infty\}$ and $u \in \mathbb{Z} \cup \{+\infty\}$. An interval analysis or range analysis [Har77] is the basic value analysis performed to infer a superset of the possible values for each program variable. There are other basic value analyses, such as constant propagation [WZ91] and its extension to sets of constants, so called value-sets. The first loses all information as soon as there is more than one value per variable to track. The second requires a widening to $\top$ (the set of all values) as soon as the set of values does not stabilize. In practice, an even more draconian k-limit widening [BHV11] is used where the set of values may contain no more than k elements. In contrast, interval analysis is a very lightweight and scalable analysis as it only tracks two constants per variable and the transfer functions are cheap. Compared to constant propagation or value-set analysis it has the benefit of gracefully losing precision through its convex approximation instead of losing all information joining values as the constants domain. Especially for our use case

where we want to find out-of-bounds memory accesses, inferring the range of values for a variable is sufficient. Next, we will describe briefly our implementation of the *interval* domain.

$$
\begin{array}{lcl}
\alpha(c) & = & [l,u] \wedge l = min(c) \wedge u = max(c) \\
\gamma([l,u]) & = & \{c \in \mathbb{Z} \mid l \leq c \leq u\} \\
 & & \\
\top_{\mathcal{I}} & = & [-\infty, +\infty] \\
\bot_{\mathcal{I}} & = & [l,u] \text{ for some } l > u \\
[l_1,u_1] \sqsubseteq_{\mathcal{I}} [l_2,u_2] & = & l_1 \geq l_2 \wedge u_1 \leq u_2 \\
[l_1,u_1] \sqcap_{\mathcal{I}} [l_2,u_2] & = & [max(l_1,l_2), min(u_1,u_2)] \\
[l_1,u_1] \sqcup_{\mathcal{I}} [l_2,u_2] & = & [min(l_1,l_2), max(u_1,u_2)] \\
[l_1,u_1] \nabla_{\mathcal{I}} [l_2,u_2] & = & [\text{if } l_2 < l_1 \text{ then } -\infty \text{ else } l_1, \text{if } u_2 > u_1 \text{ then } +\infty \text{ else } u_1] \\
[l_1,u_1] \Delta_{\mathcal{I}} [l_2,u_2] & = & \text{let } l = \text{if } l_1 = -\infty \text{ then } l_2 \text{ else } min(l_1,l_2) \\
 & & \text{and } u = \text{if } u_1 = +\infty \text{ then } u_2 \text{ else } max(u_1,u_2) \text{ in } [l,u] \\
 & & \\
[\![x = e]\!]^{\mathcal{I}}\, i & = & i\,[x \mapsto i(e)] \\
[\![x \leq e]\!]^{\mathcal{I}}\, i & = & \text{let } [l_x,u_x] = i(x) \text{ and } [l_e,u_e] = i(e) \text{ in } i\,[x \mapsto [l_x, min(u_x,u_e)]] \\
[\![x \geq e]\!]^{\mathcal{I}}\, i & = & \text{let } [l_x,u_x] = i(x) \text{ and } [l_e,u_e] = i(e) \text{ in } i\,[x \mapsto [max(l_x,l_e), u_x]] \\
[\![x \neq e]\!]^{\mathcal{I}}\, i & = & [\![x \leq e-1]\!]^{\mathcal{I}} i \sqcup_{\mathcal{I}} [\![x \geq e+1]\!]^{\mathcal{I}} i \\
[\![x == e]\!]^{\mathcal{I}}\, i & = & \text{let } r = i(x) \sqcap_{\mathcal{I}} i(e) \text{ in } i\,[x \mapsto r]
\end{array}
$$

Figure 2.21: Lattice and transfer functions for the interval domain.

$$
\begin{array}{lcl}
[l_1,u_1] +_{\mathcal{I}} [l_2,u_2] & = & [l_1+l_2, u_1+u_2] \\
[l_1,u_1] -_{\mathcal{I}} [l_2,u_2] & = & [l_1-u_2, u_1-l_2] \\
[l_1,u_1] \times_{\mathcal{I}} [l_2,u_2] & = & [min(l_1l_2,l_1u_2,u_1l_2,u_1u_2), max(l_1l_2,l_1u_2,u_1l_2,u_1u_2)] \\
[l_1,u_1] /_{\mathcal{I}} [l_2,u_2] & = & [min(l_1/l_2,l_1/u_2,u_1/l_2,u_1/u_2), max(l_1/l_2,l_1/u_2,u_1/l_2,u_1/u_2)] \\
[l_1,u_1] \&_{\mathcal{I}} [l_2,u_2] & = & [minAnd(l_1,u_1,l_2,u_2), maxAnd(l_1,u_1,l_2,u_2)] \\
[l_1,u_1] |_{\mathcal{I}} [l_2,u_2] & = & [minOr(l_1,u_1,l_2,u_2), maxOr(l_1,u_1,l_2,u_2)] \\
[l_1,u_1] \hat{}_{\mathcal{I}} [l_2,u_2] & = & [minXor(l_1,u_1,l_2,u_2), maxXor(l_1,u_1,l_2,u_2)] \\
[l_1,u_1] <<_{\mathcal{I}} [l_2,u_2] & = & \text{let } l = min(l_1 << l_2, l_1 << u_2, u_1 << l_2, u_1 << u_2) \\
 & & \text{and } u = max(l_1 << l_2, l_1 << u_2, u_1 << l_2, u_1 << u_2) \text{ in } [l,u] \\
[l_1,u_1] >>_{\mathcal{I}} [l_2,u_2] & = & [l_1,u_1] /_{\mathcal{I}} \left[2^{l_2}, 2^{u_2}\right]
\end{array}
$$

Figure 2.22: Interval arithmetics. Division is only defined for divisors: $[l,u] \not\ni 0$. The $minXor()$ functions and similar are from [Waro3].

Figure 2.21 shows the lattice of intervals and the corresponding transfer functions. Commonly, $\bot$ is defined as the empty interval and $\top$ the complete range of numbers. The shown operations naturally lift to environments e except that $\exists x.i(x) = \bot \Rightarrow e = \bot$. The subset test, meet, join, widening and narrowing are the standard definitions from [CC76; CC77].

We use $i[x \mapsto \dots]$ to denote an update of the value for variable x in the domain and $i(x)$ to retrieve the value and in abuse of notation we use $i(e)$ for the value of expression e evaluated in the domain. Hence, the assignment simply overwrites the current value of a variable and tests restrict the current value. As an abstract domain over-approximates the concrete values, the sound semantic for tests is to conservatively restrict a value, e.g. for $x \in [0, 10]$ and $x \leq [5, 8]$ the result is $x \in [0, 8]$. The *interval-set* domain to be discussed next is able to handle such disequalities as well as our novel *predicates* domain mentioned earlier.

Figure 2.22 shows the basic interval arithmetics used during the transfer functions to evaluate expressions e. Bit-wise operations, i.e. *and*, *or*, *xor*, *shift right*, *shift left* ($\&, |, \hat{}, >>, <<$) are implemented using interval arithmetic operations from [War03]. Note that the operands of the shifts, that is the number of bits to shift, must not contain negative values and the division is only defined for divisors not containing zero. To overcome the latter restriction we use a more general division semantics, that maps divisions by zero to zero, that is, if an interval contains the zero we perform the division ignoring the zero and re-add it to the result. This is implemented by splitting up the divisor into two intervals without the zero and joining the results of each division, e.g. $[5, 10]/_{\mathcal{I}}[-2, 2] = ([5, 10]/_{ign0}[-2, -1]) \sqcup_{\mathcal{I}} ([5, 10]/_{ign0}[1, 2]) \sqcup_{\mathcal{I}} [0, 0] = [-10, 10]$. This especially means that dividing by zero is always zero: $[l, u]/_{\mathcal{I}}[0, 0] = [0, 0]$. We implemented this behavior as some architectures like the *ARM* architecture allow to return 0 as the result of a division by zero by setting a processor flag [ARM10]. As a consequence, the concrete zero-handling semantics for the division has to be implemented in the translators for each platform by adding code that checks for 0 and e.g. throws an exception. Note that *PPL* implements a similar semantic for their *interval* domain implementation [BHZ08] whereas *Apron* returns zero for the division by the constant zero value and $\top$ for division by range values containing zero. The latter behavior originates from the use of rationals as coefficients in Apron. When factoring out the zero from a value, e.g. $[-a, a]$ it will take the closest possible rational r to zero for the divisors: $[-c, -r]$ and $[r, c]$. Division by such a small number leads to very large values, both negative and positive, and results in an approximation by $\top$.

The second characteristic of our interval division is that we provide three integer rounding semantics. One rounds towards 0 which is the semantics of the `div`, `idiv` instructions on the x86 platform and the `udiv`, `sdiv` instructions on ARM. The second mode rounds towards $-\infty$ (truncates the result) which is the semantics of the C language [ISO05, Ch. 6.5.5] for division and of right shifts and the `sar` instruction on x86. The third rounding semantics we provide is to round towards a smaller interval, i.e. if $r \in \mathbb{R}$ is the result of a division, the resulting interval is $[\lceil r \rceil, \lfloor r \rfloor]$. We use this semantics to implement tests and reduction operations. For instance, the congruence domain may divide an interval by a constant in which case a rounding semantic is needed that results in the largest integral interval within the solution.

interval-sets The *interval-set* domain is an extension of the *value-set* domain that uses intervals as values. It can also be seen as a powerset domain of intervals or the finite disjunctive completion [CC79a] for intervals. Consequently they are called DisIntervals

in [FL11] but simply Boxes in [GC10a]. An interesting feature of the domain is that it can represent non-convex spaces, in particular, perform thus some disequality tests precisely. However, as it is a non-relational abstract domain, symbolic disequalities can not be tracked, e.g. $x \neq y$ where neither x nor y is constant.
Generally, for each variable in its support set $x \in \mathcal{X}_V$ the domain tracks a disjunction of inequalities $\bigvee_i (l_i \leq x \wedge x \leq u_i)$, that represents a set of non-overlapping, ordered intervals, written as $x \in \{[l_1, u_1], \ldots, [l_n, u_n]\}$ for $i \in \{1, \ldots, n\}$. The interpretation of these intervals when understood as a union of their concretization is defined analogously to the *interval* abstract domain. Lattice operations and transfer functions on an interval-set are lifted from the interval transfer functions by point-wise application on each interval with a normalization step to maintain the invariant of disjunct, ordered intervals, that is $u_i < l_{i+1}$. This normalization step and the fact that most binary operations require building the cartesian product of the involved sets makes the *interval-set* much slower compared to the plain *interval* domain. While most binary operations on intervals are $\mathcal{O}(1)$ they become $\mathcal{O}(n^2)$ for interval-sets with n being the number of intervals in a set. It is possible to reduce this costs by keeping the intervals in an ordered tree structure. In fact the implementation in [GC10a] uses an extension of binary decision diagrams (BDDs) so-called linear decision diagrams (LDDs) to speed up the transfer functions. They combine interval constraints using boolean formulas which also makes their domain more expressive as it captures some relational information. This, however, requires a much more complex implementation especially regarding the widening. Our implementation does not use any of these techniques yet, thus requiring a linear traversation of the data structure during the transfer functions. Nevertheless, the domain is a good compromise between the precision of value-sets and the scalability of intervals. It gracefully loses precision by going from tracking concrete values to tracking intervals during widening.
The implementation of widening involves some extra logic, though. In the *interval* domain widening is defined as an extrapolation of unstable bounds to $-\infty$ or $+\infty$. With interval-sets widening extrapolates bounds up to the next interval (or down to), i.e. it "fills" the gap between neighboring intervals. For the first and last interval in the set, which are the bounds of the convex hull, widening extrapolates to infinity like the classic interval widening. We will illustrate the behavior of widening with some examples. Unstable inner bounds lead to the contraction of neighbor intervals: $\{[2,3],[9,10]\} \nabla_{\mathcal{IS}} \{[2,4],[9,10]\} = \{[2,10]\}$ whereas unstable outer bounds introduce infinity values, e.g. $\{[2,3],[9,10]\} \nabla_{\mathcal{IS}} \{[2,3],[9,11]\} = \{[2,3],[10,+\infty]\}$. The widened chain of interval-sets will eventually stabilize as the iterates converge towards one single interval representing the convex hull of the iterates. Termination is ensured as the outer bounds of the interval-set are widened just like normal intervals. However, given this definition, widening might still lead to slow termination if the new values are not adjacent to any of the interval in the set. For example given the interval-set $\{[0,0],[100,100]\}$ it is possible to add all the even numbers to this value with widening not removing any bounds: $\{[0,0],[2,2],[4,4]...[100,100]\}$. To avoid this value enumeration problem we do not allow the right arguments of widening – the new value – to contain more intervals than the left argument. If this is the case we join as many intervals as necessary to uphold this invariant.

Using this heuristic we do not require a fixed k-limiting for the size of an interval-set and still ensure fast convergence during widening.

Rather than lifting the interval domain $\mathcal{X}_V \to \mathcal{I}$ to interval sets $\mathcal{X}_V \to \wp(\mathcal{I})$, one could perform a more expressive, relational lifting by considering $\wp(\mathcal{X}_V \to \mathcal{I})$. In Fig. 2.23 for example the *interval-set* domain tracks spurious values–the box on the lower right–thus violating the error space. Here, the powerset of intervals would be able to represent the state space without these spurious values by tracking two separate states $s_1 = \{x \in \{[2,5],[7,8]\}, y \in \{[5,8]\}\}$ and $s_2 = \{x \in \{[2,5]\}, y \in \{[2,3],[5,8]\}\}$. However, the powerset construction has the overhead of duplicating some tracked values in this case. A solution without duplicating states is provided by our *predicate* domain (see Chap. 4), which uses implications to separate states. In the above example tracking $5 < x \to 5 \leq y$ is enough to exclude the erroneous state.
Another issue shown in [BHZ04] is that widening has to be taken special care of in powerset constructions as a pointwise lifting of widening may not terminate. Devising a widening for a disjunctive completion requires knowledge about the underlying domain as shown above with the *interval-set* domain. A similar route is taken in the implementation of the Boxes domain in [GC10a] that additionally also adds relational information between the variables. As a consequence, they are more precise than the powerset construction used in *PPL* [BHZ08].

We plan to implement a wrapper for the *Parma Polyhedra Library* (PPL) in the future to benchmark their powerset construction of intervals and compare it to our combinations of the *predicate* and *interval-set* domain. Furthermore, it would be interesting to see how their implementations of polyhedra and octagons compare to the ones provided by *Apron*. PPL also implements simple congruences and linear congruences under the name *grid abstractions* and provides reduced products thereof with polyhedra.

After showing the various features and shortcomings of the domains in our hierarchy we will next discuss how domains are combined in order to improve the analysis precision.

2.6 Combining Abstract Domains

The diagrams in Fig. 2.23 illustrate how each numeric domain in our *zeno* domains tier approximates a given state space of concrete values. We use 2-dimensional diagrams (using two variables x and y) to illustrate relational and non-relational approximations. The diagram at the top left depicts some concrete value-pairs that may occur at runtime in a program and how these values are expressed using constraints. Furthermore, the red area on the lower right represents program values for x and y that cause a program error. Here, the concrete state space does not intersect with the erroneous area, thus the program is safe. However, depending on the precision of the abstraction, some numeric domains cannot exclude this area from the approximated state. If the numeric domain infers that the intersection of the approximated value space and the erroneous space is not empty an analysis will emit a warning that the program might be incorrect.

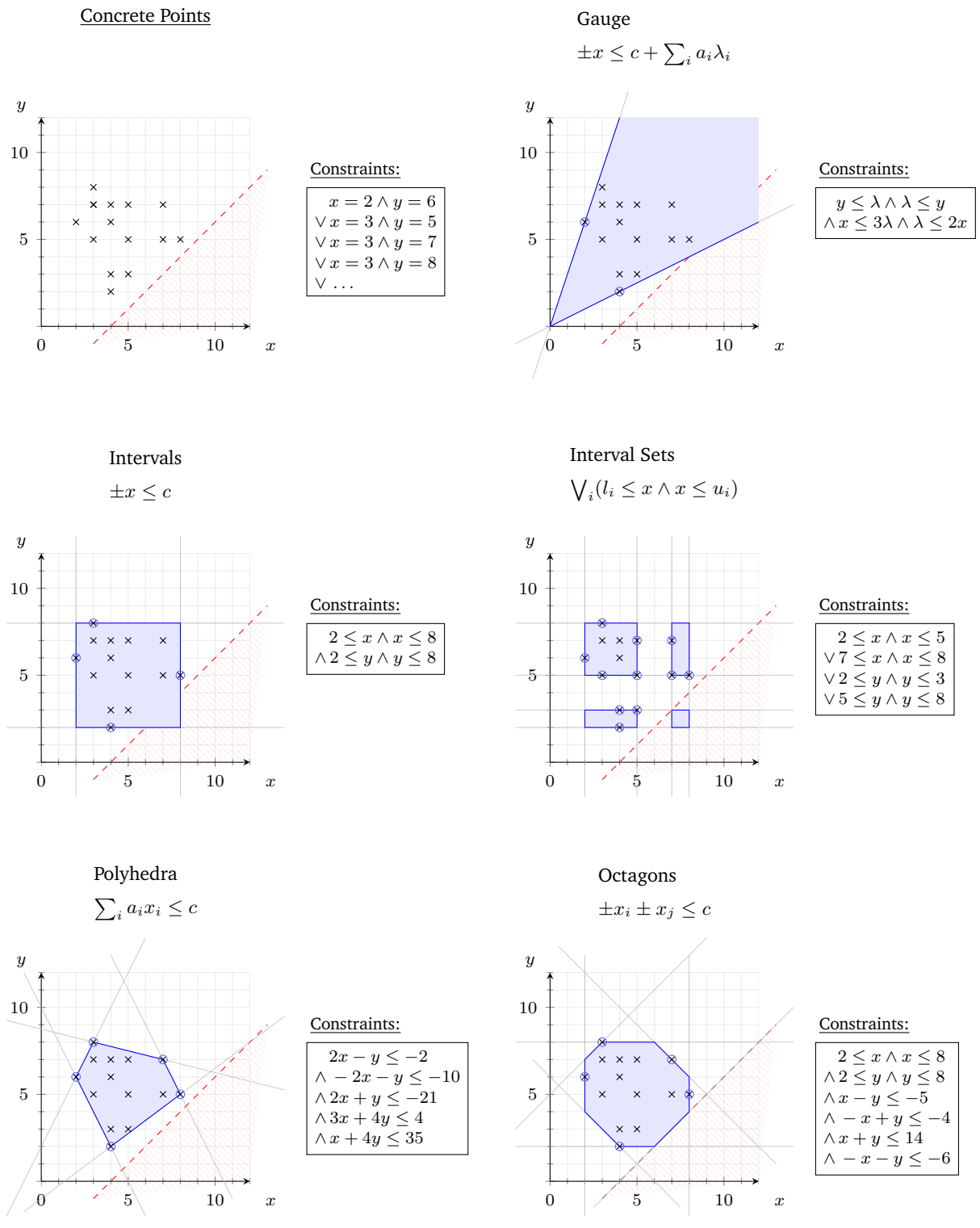


Figure 2.23: State space for two variables x and y and how various numeric abstract domains approximate this state space. The red area shall not be violated.

Out of all numeric domains displayed in Fig. 2.23 only the *polyhedra* abstraction is able to express that the program is correct. The *gauge* domain uses a wedge to approximate the space which is too coarse to prove that no intersection with the erroneous state exists. Moreover, the depicted state assumes that some loop counter λ exists that can define the shown wedge. Intervals in two dimensions form a box which in this case is too imprecise. As discussed in the previous section, *interval-sets* are more precise but due to being non-relational they only subdivide the box of intervals into smaller boxes that are not precise enough for this example. The last domain, the *octagons* domain is relational but not as precise as *polyhedra* in this case the octagon intersects with the erroneous space.

Not surprising, the *polyhedra* approximation is the most precise in this example. However, the *polyhedra* domain is very costly. Thus, depending on the program property that needs to be proved, it is often desirable to use cheaper abstractions and combine them to improve the precision. For instance, if we intersect the state spaces of the *gauge* domain and the *interval sets* domain, it is possible to exclude the erroneous state. We will next consider another example of combining cheap domains to express relatively complex spaces.

2.6.1 Cartesian Products

In Fig. 2.23 we omitted the *affine* and *congruence* domain as both cannot express a useful approximation to the presented concrete points. Both domains would be maximally imprecise here, i.e. $\top$. With respect to the *affine* domain, the domain can only track a linear relation $ax + by = c$ between x and y. However, the presented concrete points do not fit any linear relation. Similarly, the points exhibit no interesting congruence except the trivial congruence $x, y \equiv 0 \pmod 1$. Therefore, we use in Fig. 2.24 a different example for the concrete state space that results in an interesting approximation when using the *affine* and *congruence* domains.

As can be seen in Fig. 2.24, none of the domains (*intervals*, *affine*, *congruences*) alone is able to exclude the erroneous state in the approximation. However, using the cartesian product of abstract domains [CCM11; CCF13], that is, intersecting approximations from different domains we can improve the abstraction. Although the product of *affine* and *congruences* is not precise enough for verifying the example, adding *intervals* is sufficient to exclude the erroneous area from the abstraction. Note that the product of all three domains approximates the given concrete state space well enough and that it only contains one spurious value.

2.6.2 Reduced Products

The cartesian product is the most basic [CCF13] implementation of domain products where a set of domains are used to analyze a program with no exchange of information between them. The appeal of this approach is that the analysis can be run separately for each domain. However, a better analysis result can be obtained if, at each analysis step, the existing abstractions refine each other by exchanging information, a process called reduction. This idea is called a “reduced product” [CC79a; CCM11].

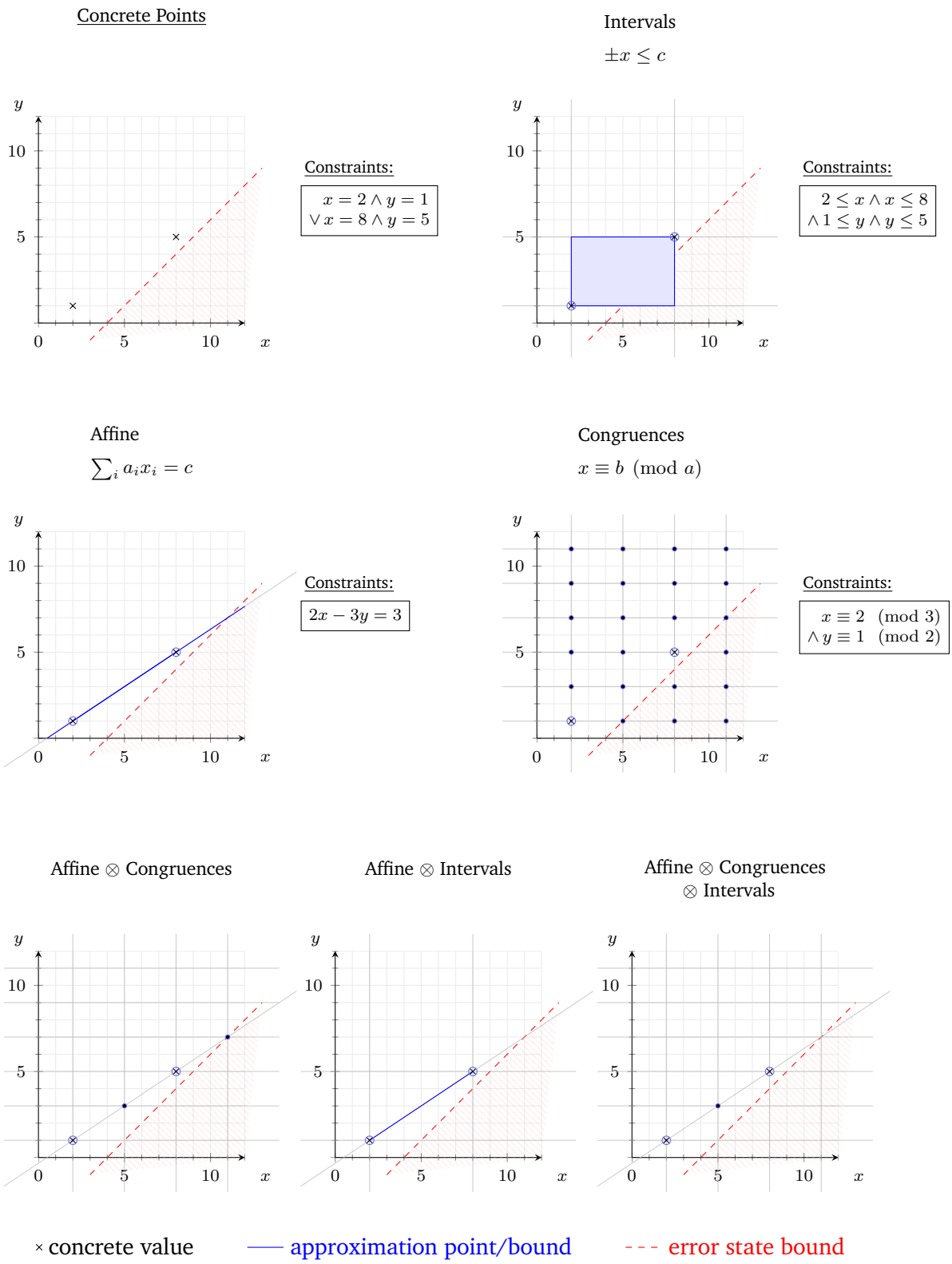


Figure 2.24: Simpler state space than in Fig. 2.23 along with the approximation by numeric domains and the product of these domains.

In theory, the reduced product requires that all possible reductions are performed all the time. In practice, however, this amounts to implementing a monolithic abstract domain that is the composition of the individual domains. Not only is it a complex task to implement such a domain, but extending the implementation whenever further domains are added requires extending the logic of the whole domain.

2.6.3 Partially Reduced Products

Hence, in practice, abstract domains are implemented and optimized to track only a small subset of program properties, e.g. congruence information or equality relations. These domains are then later combined in a so-called "partially reduced product" where each domain may interact with other domains to perform reductions after each transfer function. The goal is to build a complex abstraction by composing simple domains and using some reduction. Consider for example the *congruence* domain combined with the *intervals* domain. The first tracks the information $x \equiv 0 \pmod 4$ and the latter tracks $x \in [0, 20]$. Applying the transfer function $x < 15$ on both domains results in $x \in [0, 14]$ with the congruence domain unchanged. Now reduction may refine *intervals* using the information tracked in *congruences*, thus yielding the more precise state $x \in [0, 12]$. Note that this involves communicating the reduction from the *congruence* domain and an interpretation by the *interval* domain.
A partially reduced product is thus a cartesian product that does not perform a complete reduction between the domains but only some reductions. This allows for a modular implementation and avoids the complexity of applying reduction at every step. Nevertheless, the reduction process in a partially reduced product can itself be very complex. It may even involve a fixpoint computation, that is, refinements from domain $\mathcal{D}_1$ to domain $\mathcal{D}_2$ may cause refinements in the opposite direction and so forth. Worse, the reduction process might not even terminate with domains of infinite height. One solution to these problems is to simplify the reduction process by enforcing a domain hierarchy and allowing reduction operations only in one direction or between certain domains only [Ber+10; FL11].

One general way of implementing the partially reduced product is the iterative pairwise reduction [Gra92; CCM11] between domains and the use of well-defined interfaces for reduction. These facilitate the modularity as domains can be added or removed without modifications to the reduction process. Implementations thereof use so-called query channels to communicate information between domains. Each domain actively queries the channel after transfer functions and implements the reduction of its own state, e.g. the *intervals* domain queries congruence information and interprets it to refine itself. The so-called "open product" from [CLV00] extends the idea of communication between domains to happen at any time during transfer functions on a push principle instead of queries.

All of these techniques and further ideas have been implemented in the Astrée [Cou+07] static analyzer. Our domain refinement mechanism implements the ideas mentioned in this section. Some, such as modularity, iterative pairwise refinement, reduction in only one direction, are a natural consequence of cofibered domains whereas other ideas had to be retrofitted, e.g. the refinement between child and parent domains.

2.6.4 Reduction in Cofibered Domains

Cofibered domains implement a more restricted refinement mechanism compared to the partially reduced product. Here, transfer functions are used to perform the reduction from parent domains to child domains. A parent synthesizes the transfer functions to apply on the child depending on its own state, thus performing reduction. The benefit is that no separate refinement operations or a domain communication channel needs to be implemented. Each domain only needs to implement reduction semantics for its child, which in our modular construction means implementing a well-defined interface. In this setting reductions are between a pair of domains, are propagated iteratively and might even involve a fixpoint computation. Note that these ideas follow the reduction framework proposed by Granger [Gra92]. However, reductions are generally only one way.
As an example, consider the reduction between the *redundant affine* domain and its child the *interval* domain as described in Sect. 2.5.4. Here, each test on the *redundant affine* domain might lead to the application of further tests to the child domain. If the domain tracks $x = y$ and sees the test $x < 5$ it will also apply the test $y < 5$ on the child thereby performing a reduction.

A second characteristic of cofibered domains is that transfer functions can be modified, thereby performing reduction of the child. For instance, as shown in Sect. 2.5.4 the *affine* domain can simplify linear assignments, e.g. $y = 2x - x$ to $y = x$ before applying them on the *interval* child, thus improving the precision of the result. Executing the original assignment in the *interval* domain is less precise if the value for x is an interval. As partially reduced product implementations execute a transfer function on all domains in parallel and only then perform the reduction, the cofibered approach can be cheaper.
Another example is the *conguence* domain, which scales linear assignments or tests by inlining the tracked congruences, thereby keeping the child reduced. Consider the congruence domain tracking $x \equiv 0 \pmod 4$ like in the example in the previous section. However, as the child is reduced, the *interval* domain tracks $x \in [0, 5]$ instead of $x \in [0, 20]$ as before. Now, applying the test $x < 15$ on the *congruence* domain will result in the scaled test $x < 4$ being applied to the *interval* domain, which itself results in $x \in [0, 3]$. Here, no further reduction in intervals is necessary. The querying of the intervals of x on the *congruence* domain will return the value $4[0, 3] = [0, 12]$. By using the possibility to modify transfer functions we remove the need for later reduction steps.
The translation of transfer functions also has the benefit of short-circuit evaluation, that is, if a parent domain infers that a test is infeasible then it does not need to apply the test to the child.

2.6.5 Domain Reduction using Channels

As argued in the previous section, due to the cofibered design the reduction from parent to child domains is implemented by applying transfer functions on the child. Reduction in one direction, namely from the top to the bottom of the domains stack can thus be implemented naturally. However, it is desirable to perform reduction in the opposite direction, too, i.e. from a child to its parent. To facilitate this backward reduction between domains we use

two channels to communicate the necessary information.

Query Channel

This channel exists to query the values of variables or linear expressions. It is used by the fixpoint engine to ask for all possible jump targets when evaluating a computed jump. However, a domain may use this channel to request information from child domains and use the information to reduce its own state. Nevertheless, the channel is currently not used for reductions as requesting range information can usually be performed by testing range constraints, e.g. $x < 5$ on the child domain and observing if the result is $\bot_{\mathcal{C}}$. The latter is more precise as the query performs a convex approximation to suit the relatively simple data-type used in the channel. Note that it would be possible to extend the channel and enrich the information that can be queried. However, using transfer functions, i.e. tests, will always be more precise as domains implement semantics for tests, which are not easily expressible in a query.

Synthesized Channel

This so-called $synthChannel$ is used when applying transfer functions on the child. During a transfer function, a child domain synthesizes information that describe new facts caused by the current transfer function. A parent domain may use this information to perform new refinements to its own state and/or apply new transfer functions to the child. It is even possible to perform a fixpoint computation to do reduction as in [Gra92]. Hence, care must be taken to ensure termination of the domain-local fixpoint computation [Ber+10]. In our setting, most domains do not reduce the child more than once, thus propagation of refinements is only in one direction. However, domains that refine the child again, e.g. the *predicate* domain, make sure that the fixpoint computation terminates in finitely many steps. Currently, we propagate three types of information through the $synthChannel$:

- variables that became constant: $x = c$
- new equalities between variables: $x = 2y$
- new inferred predicates and implications: $x < 5 \rightarrow y < 10$

We will now give some examples for reduction using this information.

The *affine* domain tracks equalities between variables and removes fully determined variables from its child. That is, if it knows that $x = y$ only y needs to be tracked in the child. Hence, when applying a transfer function on the child domain the *affine* domain extracts from the $synthChannel$ the new equalities and then removes some variables from its child. As the *affine* domain also tracks constants, that is, equalities of the form $x = c$ it also removes variables from its child that became constant. This reduction of the affine state improves precision because the domain itself infers new equalities on joins. For example consider two states $s_1, s_2 \in \mathcal{A} \rhd \mathcal{I}$, that is, the *affine* domain having the *interval* domain as child. Now let $s_1 = \langle \{x = 3\}, \{y \in [5, 10]\} \rangle$ and $s_2 = \langle \{x = 5\}, \{y \in [3, 10]\} \rangle$. Applying

tests to each state yields two new states $s'_1 = [\![y \leq 5]\!]^{\mathcal{A}} s_1$ and $s'_2 = [\![y \leq 3]\!]^{\mathcal{A}} s_2$. The *affine* domain simply forwards the test to the child as it does not implement semantics for inequality tests. Assuming that the *affine* domain reduces its state using the $synthChannel$, the resulting states after applying the transfer functions are: $s'_1 = \langle \{x = 3, y = 5\}, \emptyset \rangle$ and $s'_2 = \langle \{x = 5, y = 3\}, \emptyset \rangle$. The join of $s'_1 \sqcup_{\mathcal{A}} s'_2$ then infers a new affine equality in the resulting state: $s'_1 \sqcup_{\mathcal{A}} s'_2 = \langle \{x = y\}, \{y \in [3, 5]\} \rangle$. This equality would not be inferred without the reduction, i.e. joining the unreduced states: $s''_1 = \langle \{x = 3\}, \{y \in [5, 5]\} \rangle$ and $s''_2 = \langle \{x = 5\}, \{y \in [3, 3]\} \rangle$ results in $s''_1 \sqcup_{\mathcal{A}} s''_2 = \langle \emptyset, \{x \in [3, 5], y \in [3, 5]\} \rangle$.

The *congruence* domain benefits from reduction in a similar way to the *affine* domain. It infers new congruence information for constant variables during a join. For example joining $x = 3$ and $x = 6$ infers the congruence $x \equiv 0 \pmod 3$. Hence, the domain reduces its state whenever variables have become constant in the child.

The *predicate* domain uses the $synthChannel$ to extract predicates that describe the precision loss during the join of convex child domains (see Chap. 4). Additionally if a variable became constant due to a test, the *predicate* domain uses this new fact to deduce the truth valuation of predicates mentioning the variable. For example the domain may track the implication $f = 0 \to x \leq 0$ with the *interval* domain tracking $\{f \in [0, 1], x \in [-5, 5]\}$. After applying the test $f < 1$ to the domain state the *predicate* domain can infer due to the information propagated in the $synthChannel$ that the premise of the implication is true. As a consequence, it applies the predicate $x \leq 0$ to its child thereby performing a further reduction.

The *undef* domain similarly uses the fact that a flag variable becomes constant to infer if the values guarded by the flag are defined or not (see Chap. 5). It then repartitions its state but does not apply further transfer functions to the child. Still, the reduction of the *undef* domain modified the concretization of the child domain. For instance, if $x = \top$ because the variable might be undefined, then after applying the test on a flag we might know that $x \in [0, 10]$, that is, we know that x is defined.

As the $synthChannel$ notifies only about the parts of the domain state that have changed due to a transfer function, it allows a parent domain to only query a small subset of all known variables. A corresponding reduction in a partially reduced product would have to push the set of changed variables (since this set may differ from the set of variables in the transfer function) and let other domains perform queries on any domain for these variables. While the implementation in the cofibered setup is less flexible as it only allows querying its child, it still allows for very powerful reductions.

2.6.6 Reduced Cardinal Power

A lesser known domain combination method that facilitates reduction is the "reduced cardinal power" [CC79a; CCF13] or its generalization the "reduced relative power" [GR99]. The construction allows expressing relational information between two domains, that is, values in one domain depend on values in another domain. In particular, it allows to track disjunctive information over abstract values as the relation acts as an implication. The construction takes a domain $\mathcal{D}_b$ as base and another domain $\mathcal{D}_e$ as exponent, resulting in

the power $\mathcal{D}_b^{\mathcal{D}_e}$. Abstract elements $e \in \mathcal{D}_e$ imply states $b \in \mathcal{D}_b$, thereby implementing a case analysis: if e then b.
Take for instance our *flags*, *predicate* and *undef* domain, all of which track implications between boolean expressions and constraints over variables. This allows the domains to split the state space in the child depending on their own state. Hence, these domains can be seen as an instance of the "reduced cardinal power". Where *flags* and *undef* use a boolean lattice: $\{\bot_{\mathcal{B}}, false, true, \top_{\mathcal{B}}\}$ as exponent, the *predicate* domain is more complex. The latter domain uses the same lattice $\mathcal{P}$, where $\mathcal{P}$ is a set of predicates, for the base and the exponent, a so-called auto-dependent reduced cardinal power [GR99]. Our implementation of the reduced cardinal power does not track separate states $b \in \mathcal{D}_b$ per exponent $e \in \mathcal{D}_e$ but uses one child state that is refined based on the state e. The base domain $\mathcal{D}_b$ is thereby itself a partial reduced product of a domain tracking predicates and an arbitrary numeric child domain that interprets predicates.

In conclusion, the flexibility of cofibered domains coupled with a reduction channel allows us to partially implement both types of reductions from [CC79a].
Next, we will detail how an analysis of programs containing procedures is achieved.

2.7 Interprocedural Analysis

Our analysis infers transitions to new program points during the fixpoint computation as detailed in Fig. 2.8 and stores these transitions using the CFG storage shown in Fig. 2.10. In the resulting CFG data structure, calls may be treated semantically like a normal jump and are thus tracked by adding a new CFG transition from a call-site to the callee (analogously for returns). Following this approach, an interprocedural CFG is simply a graph with the intraprocedural CFGs inlined. However, this approach does not allow to take into account the context of a function call during the analysis, i.e. the distinct call-sites and the values of the function parameters. It is desirable to not discard this information as it allows for more precise context-sensitive analyses and allows to reuse analysis results for a function body. Hence, we use the branch type hints described in Sect. 2.4.6 that are provided by the instruction translator to facilitate the implementation of more specialized semantics for call and return instructions.

2.7.1 Call-String Approach

A context-sensitive interprocedural analysis approach that we implemented is the call-string approach as proposed in [SP81]. Here, the call-stack is modeled by the so-called call-string, that records each call transition as a tuple $\langle c_s, c_t \rangle \in \wp(C_s \times C_t)$ where $C_s \subseteq P$ is the set of addresses of call sites and $C_t \subseteq P$ are the addresses of function entries, or call targets. In contrast to [SP81] we also record the call targets as computed calls might call more than one function. The call-string is then just a list $\kappa \in (C_s \times C_t)^*$ of tuples $\langle c_s, c_t \rangle$ of call sites and call targets. We use $^*, ^+$ as in regular expressions to denote repetitions due to recursion. An analysis based on call-strings then maps a tuple $\langle \kappa, a \rangle$ with $a \in P$

being an instruction address, to abstract states $s \in \mathcal{D}$. This mapping $(C_s \times C_t)^* \times P \mapsto \mathcal{D}$ improves the precision of an analysis as it avoids the propagation of abstract states along interprocedurally infeasible paths.

```
1  void f(int param) {
2    g(param);
3  }
4
5  void g(int param) {
6    g(param - 1);
7  }
8
9  int main(void) {
10   f(1);
11   g(2);
12 }
```

Figure 2.25: Example for function calls.

For example consider the program in Fig. 2.25, the possible call-strings are all the prefixes of the following two strings: $\kappa_1^i = \langle 0, 9\rangle \cdot \langle 10, 1\rangle \cdot \langle 2, 5\rangle \cdot \langle 6, 5\rangle^i$ which represents the call-stack `[main; f; g`$^+$`]` and $\kappa_2^i = \langle 0, 9\rangle \cdot \langle 11, 5\rangle \cdot \langle 6, 5\rangle^i$ which represents the call-stack `[main; g`$^+$`]`. Note that entering a function appends a tuple and returning from a function removes the last tuple in the call-string, thus any prefix of κ_1 and κ_2 are valid call-strings. The size of the call-string set, is infinite as κ_1 and κ_2 are expandable to $\kappa_1', \kappa_1'', \ldots, \kappa_1^n$ by appending to the string the recursive call to function g. In order to deal with recursion the call-string length is limited by a constant k. Given a call-string longer than k we record only the suffix of length k, thereby merging contexts (abstract states) whose k^{th} parent differ.

The benefit of using a call-string is that we are able to distinguish different call paths and summarize states with the same call-string, that is, we can define a collecting semantics with the program points being $p \in (C_s \times C_t)^* \times P$ defined as a tuple of the call-string and the instruction address. Program points having the same call history will be mapped to the same abstract state. Moreover, propagation along infeasible paths is avoided, e.g. given the call-string $\langle 0, 9\rangle \cdot \langle 10, 1\rangle \cdot \langle 2, 5\rangle \cdot \langle 6, 5\rangle$ it is not feasible to return from function g to function `f` even though the interprocedural CFG contains a return edge from g to `f`. Recursion is handled by the call-string approach using summarization, that is, a series of k recursive function calls will result in a call-string suffix of k identical tuples so that the analysis will summarize the function. In our example the last k tuples of the call-string will contain k-times the tuple $\langle 6, 5\rangle$. Any further recursive call to `g()` will be merged with the states for this call-string.

The choice of the call-string length k thus defines a trade-off between precision and scalability as it indicates the maximal length of call paths that are kept separate in the analysis. However, in practice, finding a good choice for k is difficult as it depends on

the program and its runtime call-graph. In our implementation k may thus be given as a parameter to the analysis. A second problem is that we might want to not re-analyze some functions, e.g. function `g()` in the example, even though it was called on a different path but would want to use a summary instead. The call-string approach does not offer enough flexibility in this case. Finally, the main drawback of using a call-string is that, after exceeding k, a precision loss is incurred when we return as the state has to be propagated to all possible return sites whose $k-1$ suffix matches the current context.

2.7.2 Function Summaries

In order to overcome some of the drawbacks of the call-string approach and to summarize earlier we implemented a function summarization mechanism as part of the *stack* domain. As described in Sect.2.5.1 our *stack* domain does not track the stack as one large region but tracks separate stack frames per function and uses pointers to connect the frames, thereby expressing the caller-callee relationship. In the call-string approach the stack is modeled as a linked-list of stack frames whereas our summarization approach models the stack as a graph. Each stack frame may have multiple predecessor frames, given the set of call-sites of the function. Our summarization implementation uses only one stack frame per procedure, that is, a single summary for all calling contexts (analogous to a call-string of length 0). In general, one could allow an arbitrary number of summaries per function.

In the example in Fig. 2.25 we track three summaries in total, one per function, that is, we store one state per program point. The call-string approach with a conservatively small value for k, e.g. $k = 4$ would result in 13 different summaries for the whole stack at each program point in `g()`. Note that $k = 4$ is a small bound for call-strings as in practice non-recursive function calls may be hundreds of calls deep. Generally, given a call depth k and a program of n procedures, the upper bound for separately tracking non-recursive contexts is kn. With the assumption of $n \gg k$, the upper bound is $O(k^2)$. Hence, the call-string approach rapidly becomes too expensive.

In contrast, the summarization approach suffers from precision loss as we merge the states of all call sites that call a particular function, that is, we perform a context-insensitive analysis. In order to maintain precision we rely on relational domains to separate the state space for different call paths, thus achieving some context sensitivity. To this end, the *stack* domain builds and tracks an approximation of the call-graph and adds Boolean flags $f_{ij} \in \mathcal{X}_V$ to its child domain that states if a path ij in the call-graph is feasible. The child domain may then infer relations between the flags and the input parameters of a function call [Simo8a]. In the example in Fig. 2.25 the call to `g()` in the body of the function `main()` results in setting the flag $f_{mg} = 1$ whereas the flag $f_{fg} = 0$ expresses that the other call path reaching `g()` through `f()` is not feasible. By leveraging relational information in the child domains, e.g. the *affine* or the *predicate* domain, we can refine the set of input parameters that reaches `g()` on this path so that $param \in [2, 2]$.
As the idea behind tracking a graph of stack frames is to reuse methods from shape analysis for recursive data structures, we handle recursion by adding self-loop edges to the graph and computing a summary for recursive calls. However, the implementation of the

summarization approach does not yet handle tail recursion. A more detailed discussion how the flags connect callers and callees and how we leverage the *undef* domain to merge different stack layouts is presented in Chap. 5.

Note that we did not perform an experimental evaluation of both approaches as in [Mar99]. It would be interesting to compare the approaches using different domains in order to see how the precision vs. scalability trade-off turns out in practice. Especially, as both approaches are on opposite sides of the precision and scalability continuum. Future work should address the evaluation of both approaches and their combination, which may introduce a greater flexibility in interprocedural analysis.

2.8 Related Work

While many static analyses for executables exist, most of them do not raise above the abstraction level of our *L(finite)* interface which precludes the use of classic numeric domains that assume that variables range over $\mathbb{Z}$. The latter domains, however, are particularly attractive as they have been shown to scale up to the analysis of large-scale C programs comprising several 100kLOC [Bla+03a]. When analyzing C rather than binaries, the result of each assignment can be checked for overflow since the types of the high-level language indicate if signed or unsigned arithmetic is used. This distinction is lost for many assembler instructions. For instance, **add**, **sub** and **shl** on Intel x86 carry no signedness information. This difficulty seems to be reason enough for products such as CodeSurfer/x86 [Bal+05] to consider numeric domains ranging over $\mathbb{Z}_{2^w}$, that is, domains that perform operations over a modulus ring, in order to calculate the input/output behavior of a basic block [Eld+11]. Interestingly, it is possible to synthesize transfer functions for a complete basic block that very accurately model bit-level operations composed of **xor** and **shl** [KS10]. However, this technique requires expensive SAT solving and does not scale to 64-bit architectures [Eld+11]. Moreover, it is neither flexible since every new property has to be added to the Boolean formula passed to the SAT solver nor are the results easy to interpret. Note that SAT solving methods can still be integrated in our framework using the finite domain interface. Indeed, by only tracking variables defined by bit-level operations and using numeric domains to query and update other variables, much smaller Boolean formulas can be obtained as described in Sect. 2.5.3.

One recent approach to deal with wrapping in interval arithmetic [Nav+12] is to treat intervals as a superposition of signed and unsigned values until operations require the concretization of the sign. This is similar to our approach of implementing wrapping as a domain as we track intervals over $\mathbb{Z}$ and execute sign agnostic operations on this representation. The representation chosen by Navas et. al [Nav+12] is more precise on certain arithmetic operations but introduces problems such as non-monotonicity and a non associative join. The resulting domain is not a lattice anymore. In order to maintain precise, terminating and sound transfer functions requires special care, which, in turn complicates the fixpoint. Furthermore, their approach is limited to intervals whereas we aim for a modular implementation of wrapping using only transfer functions on child domains.
One drawback of our approach is that equality relations are lost when wrapping unbounded

values as the *wrapping* domain assigns the saturated range to such variables during wrapping. Future work should consider the ideas from [BLH12] to improve on this.

Another challenging aspect of analyzing low-level code, be it C or assembler, is the reconstruction of the memory layout. Our approach of using field inference [Mino6a] for registers seems novel and is conceptually simpler than approaches [CD11] based on program transformation.

Due to the conceptual simplicity, the call-string approach is widely used in interprocedural static analyses [SIG07; Mar99; KK08b] as well as in the analysis of executables [Bal07; Kin10; Lak+11; Boc09]. When dealing with recursion the call-string method is far from optimal as it re-analyzes a recursive function k times before a summarization is performed. To solve this issue Khedker et. al [KK08b] and later Lakhotia et. al [Lak+11] try to recognize recursive procedure calls by using regular expressions on the call-string. If a series of recursive calls is recognized the call-string is not augmented and a summarization is performed. Furthermore, Khedker et. al [KK08b] solve another shortcoming of call-string analysis, namely, the redundant re-analysis of a procedure due to a different call-string even if the input state parameters are the same. In such cases it would be advisable to re-use an already existing analysis result of the procedure. To this end they propose to use a tuple $\langle S^\sharp \times K \rangle$ consisting of the abstract state and the call-string as context for interprocedural analyses. However, this extension is only guaranteed to terminate for lattices with finite height.

Apinis et. al [ASV12] take the idea of using the state as a context further. They propose an interprocedural framework where only partial state information is used as context, which allows to fine-tune the analysis to e.g. treat global variables as context-insensitive whereas procedure parameters are analyzed with full context-sensitivity. Moreover, they show how the problem of non-termination with infinite lattices can be solved in their framework. Future work should consider including some of their ideas in our framework. However, in order to be able to use relational domains to compute partial state contexts, methods such as variable packing [Bla+03b; VB04] might be necessary.

The problems of applying widening in the context of binary analysis has attracted little attention so far. It has been observed that narrowing after widening may lead to a precision loss in domains whose transfer functions depend on the values of variables [SK06]. The reconstruction of the CFG can be thought of as such a domain: If the pointer offset into a table of jump goals is temporarily unrestricted after widening, the set of possible jump targets is meaningless and the analysis cannot continue.

In the next chapter we will detail how our novel approach to implement widening heuristics as abstract domains and, in particular, the use of widening thresholds significantly improves the precision of widening.

Part II

Precision Improvements through Novel Abstract Domains

3 | *Widening* as an Abstract Domain

3.1 Introduction

Adding numeric domains of infinite height to a static analysis requires that widening and/or narrowing is applied within each loop of the program to ensure termination [CC76; CC77; CC92a]. Commonly, this is implemented by modifying the fixpoint algorithm to perform upward and downward iterations while a pre-analysis determines necessary widening points. Firstly, downward iterations can be problematic since a widened state can induce a precision loss that cannot be reverted with the narrowed numeric state [HH12; SK06]. Secondly, determining a best set of widening points is not even possible for irreducible control flow graphs (CFGs) [Bou93]. Worse, these algorithms cannot be applied in the context of analyzing machine code, as the CFG is re-constructed on-the-fly while computing the fixpoint [Bal+05]. Moreover, narrowing alone is often not enough to obtain precise fixpoints which has been illustrated in many papers that present improved widenings/narrowings [GR06a; GR07; HH12; LJG11; SK06].

All of these approaches require disruptive changes to the fixpoint engine, for instance, tracking several abstract states [GR06a], temporarily disabling parts of the CFG [GR07], performing a pre-analysis with different semantics [HPR97; LJG11], widening with respect to "landmarks" [SK06] or referring to user-supplied thresholds [Bla+02; Kir+12; LL11].

This chapter shows that widening and its various refinements can be implemented without modifying an existing fixpoint engine, thereby making numeric domains available to analyses that are oblivious to the challenges of widening. Specifically, we implement the inference of widening points and the various widening heuristics as abstract domains that can be plugged into an analysis in a modular way. The key idea of our approach is to implement abstract domains as cofibered domains [Ven96], an approach sometimes called "functor domains" [Bla+03a]. This modular approach not only reduces the overall complexity of an analysis, it also facilitates the comparison and combination of various widening heuristics.

3.1.1 Rapid Convergence

Figure 3.1 presents a simple loop and the corresponding CFG. Consider inserting a widening operator into the equation of the no-op edge from v_6 to v_3, yielding $s_3 := s_3 \nabla_{\mathcal{D}} (s_3 \sqcup_{\mathcal{D}} F_6^3(s_6)) = s_3 \nabla_{\mathcal{D}} (s_3 \sqcup_{\mathcal{D}} s_6)$. Although because of widening termination is now guaranteed,

```
1 int x = 0;
2 int y = 0;
3 while (x <= 99){
4   x = x + 1;
5   y = y + 1;
6 }
7
```

step	line		intervals x	intervals y	affine	thresholds
1	2		$[0,0]$		$x=0$	
2	3		$[0,0]$	$[0,0]$	$x=0, y=0$	
3	4		$[0,0]$	$[0,0]$	$x=0, y=0$	$x \leq 99$
4	5		$[1,1]$	$[0,0]$	$x=1, y=0$	$x \leq 100$
5	6		$[1,1]$	$[1,1]$	$x=1, y=1$	$x \leq 100$
6	3	$\sqcup$	$[0,1]$	$[0,1]$	$x=y$	$x \leq 100$
6'	3'	∇	$[0,100]$	$[0,100]$	$x=y$	$x \leq 100$
7	4		$[0,99]$	$[0,99]$	$x=y$	$x \leq 99, x \leq 100$
8	5		$[1,100]$	$[0,99]$	$x=y+1$	$x \leq 100$
9	6		$[1,100]$	$[1,100]$	$x=y$	$x \leq 100$
10	3	$\sqsubseteq$	$[0,100]$	$[0,100]$	$x=y$	$x \leq 100$
11	7		$[100,100]$	$[100,100]$		$x \leq 100$

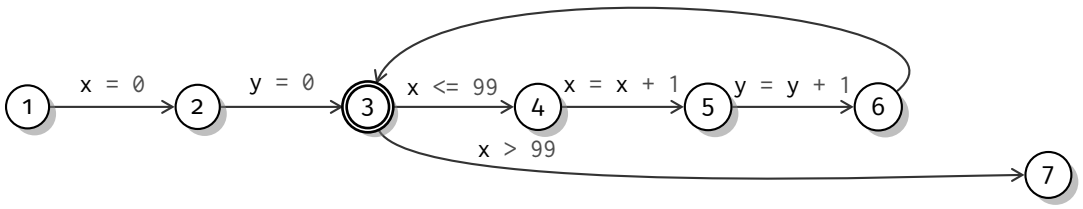


Figure 3.1: Rapid convergence during widening.

the result of, say, an interval analysis [CC77] is imprecise: $\{x \in [0,0], y \in [0,0]\} \nabla_{\mathcal{D}} \{x \in [0,1], y \in [0,1]\} = \{x \in [0,\infty], y \in [0,\infty]\}$. This stable post-fixpoint can, in principle, be made more precise by replacing the widening with a narrowing operator and re-running the fixpoint computation just for the loop body. However, this requires meddling with the fixpoint engine in order to identify the loop and its in- and outgoing edges and changing the way states are handled: for example our updates $s_j := s_j \sqcup_{\mathcal{D}} F_i^j(s_i)$ are *extensive* [HH12] ($s_j \sqsubseteq_{\mathcal{D}} s_j \sqcup_{\mathcal{D}} F_i^j(s_i)$) so that the states cannot shrink by evaluating the updates. We therefore avoid narrowing altogether to avoid changing the way states are stored. Instead, the next chapters present domains that implement more precise widenings. Before detailing these, we will show how the example of the simple loop is analyzed using our widening abstract domains. We illustrate this using a cofibered threshold domain $\mathcal{T}$ and a cofibered affine domain $\mathcal{A}$ and the interval domain to build the domain stack $\mathcal{T} \triangleright \mathcal{A} \triangleright \mathcal{I}$.

The table in Fig. 3.1 presents the analysis of the loop over $\mathcal{T} \triangleright \mathcal{A} \triangleright \mathcal{I}$ where the state of each domain is written in a separate column. The states of the interval and affine domain for steps 1 to 6 are straightforward. The threshold domain tracks all conditions in guards that are redundant. A test is said to be redundant if it does not modify the state if applied. Here in step 3 `x <= 99`. These so-called predicates are transformed by assignments, here yielding $x \leq 100$ after `x = x + 1;`. In step 6, the state at line 3: $x, y \in [1,1]$ corresponding to one loop iteration is joined with the previous state at line 3: $x, y \in [0,0]$, yielding the intervals $[0,1]$ for both, `x` and `y` together with the affine relation $x = y$ and the threshold $x \leq 100$ since it is still redundant in the joined state. The interim step 6' shows how the state obtained at step 2 is widened with respect to the state at step 6: the threshold domain applies widening on its child, yielding $x, y \in [0,\infty]$ for the interval domain while the

affine domain returns the join $x = y$. The affine domain always performs a join instead of widening since its lattice is of finite height [Kar76]. The threshold domain then refines this state by applying the test $x \leq 100$. The affine domain passes this test to its child, the interval domain, but also applies the tests $\sigma(x \leq 100)$ for any substitution $\sigma = [x/y]$ that can be derived from equalities over x. This refines the interval domain to $x, y \in [0, 100]$ as shown as step 6'. Steps 7 to 10 ascertain that this state is indeed a fixpoint of the loop, yielding the post-condition shown as step 11.

The example illustrates two consequences of this cofibered arrangement of domains: firstly, it is a modular way of combining several domains, thus keeping each domain simple; secondly, information can be propagated between domains by applying several operations on a child $\mathcal{C}$ for each operation on the parent $\mathcal{D}$.

One might argue that the modular design itself creates the need for propagation which is unnecessary when using a monolithic domain such as an off-the-shelf polyhedra package [Bag+05]. However, combining several simple domains allows for a more flexible trade-off between efficiency and precision by adjusting the interaction between domains [SMS11]. For instance, in this modular setup, the information in the affine domain is not intermingled with information on variable bounds, thereby allowing the affine domain (which has finite height) to compute a join while the interval domain performs widening.

3.1.2 Abstract Domains for Widening

The implementation of the various widening strategies builds on the ability to separate various concerns into individual domains. These domains are as follows:

Widening Points Domain: Rather than enhancing a fixpoint engine to identify widening points in loops, we propose a domain that turns a join operation into a widening when it observes that the state is propagated along a back-edge of the CFG. This simple technique for irreducible CFGs [Bou93] and CFGs that are constructed on-the-fly [Bal+05] works surprisingly well in practice.

Delay Domain: A domain which postpones widening is presented that ensures precise results for loops containing assignments of constants. These assignments often occur due to loop initialization code or state machines inside a loop.

Threshold Domain: We implement widening with thresholds [Bla+03a; HPR97] but infer the thresholds automatically. We present the basic domain that infers thresholds from tests. Unlike previous work [LJG11] that extracts thresholds from a pre-analysis using the domain of polyhedra [CH78], our domain is independent of any numeric domain.

Phased Domain: We provide an automatic way to separate the state space of loops into several phases, where phase boundaries are automatically inferred from tests within the loop, similar to guided static analysis [GR07]. This domain can be seen as an instance of a decision tree domain combinator [Cou+06].

3.2 Inferring Widening Points

For programs made up of well-nested loops, widening is only required at each loop head in the program [CC92b], which renders fixpoint computations relatively straightforward. However, widening, unlike join, is not monotone, commutative nor associative [Coro8] thus for better precision a minimal set of widening points is desirable. However, for programs with irreducible CFGs, it is generally necessary to place more than one widening point in each cycle [Bou93] and, hence, a widening heuristic must not lose precision when widening is applied several times within a loop. This, in turn, implies that a conservative heuristic, which suggests rather many widening points, suffices. We now present such a heuristic that is also applicable to the analysis of machine code, implemented as abstract domain $\mathcal{WP}$. The domain observes back-edges, that is, information flowing from higher to lower addresses. Once observed, the next join on $\mathcal{WP} \rhd \mathcal{C}$ translates to a widening on the child $\mathcal{C}$.

$$
\begin{array}{lcl}
[\![l : x = e]\!]^{\mathcal{WP}} \langle\langle l^w, f^w\rangle, c\rangle & = & \langle\langle l, f^w \vee (l < l^w)\rangle, [\![l : x = e]\!]^{\mathcal{C}} c\rangle) \\
[\![l : e \leq 0]\!]^{\mathcal{WP}} \langle\langle l^w, f^w\rangle, c\rangle & = & \langle\langle l, f^w \vee (l < l^w)\rangle, [\![l : e \leq 0]\!]^{\mathcal{C}} c\rangle) \\
\langle w_1, c_1\rangle \sqsubseteq_{\mathcal{WP}} \langle w_2, c_2\rangle & = & c_1 \sqsubseteq_{\mathcal{C}} c_2 \\
\langle\langle l_1^w, f_1^w\rangle, c_1\rangle \sqcup_{\mathcal{WP}} \langle\langle l_2^w, f_2^w\rangle, c_2\rangle & = & \begin{cases} \langle\langle l, \mathit{false}\rangle, c_1 \nabla_{\mathcal{C}}^{l} c_2\rangle & \text{if } f_1^w \vee f_2^w \\ \langle\langle l, \mathit{false}\rangle, c_1 \sqcup_{\mathcal{C}} c_2\rangle & \text{otherwise} \end{cases} \\
 & & \text{where } l = \max(l_1^w, l_2^w)
\end{array}
$$

Figure 3.2: Lattice and transfer functions for the widening point domain.

For the sake of finding back-edges, we assume that statement labels $l \in \mathit{Lab}$ represent the code address of a statement or test. The widening points domain is given by the lattice $\langle \mathcal{WP} \rhd \mathcal{C}, \sqsubseteq_{\mathcal{WP}}, \sqcup_{\mathcal{WP}}, \sqcap_{\mathcal{WP}}\rangle$ where $\mathcal{WP} : \mathit{Lab} \times \{\mathit{true}, \mathit{false}\}$ is a tuple of the last program point and a flag indicating if a backward edge has been observed. If set, widening is applied at the next junction node at which point the loop is usually completely traversed. Figure 3.2 defines the domain operations, that is, the transfer functions for assignment and the lattice functions for subset and join. Each function operates on tuples $\langle w, c\rangle \in \mathcal{WP} \rhd \mathcal{C}$ where $w \equiv \langle l^w, f^w\rangle \in \mathcal{WP}$. The transfer functions $[\![\cdot]\!]^{\mathcal{WP}}$ on $\mathcal{WP}$ apply the corresponding operation $[\![\cdot]\!]^{\mathcal{C}}$ on the child $c \in \mathcal{C}$ while tracking the current label l and whether a backward edge $l^w \to l$ with $l < l^w$ has been observed. The subset test $\sqsubseteq_{\mathcal{WP}}$ translates to a subset test on the child, indicating that the $\mathcal{WP}$ domain does not actually infer any information about the state of the program and is therefore, per definition, stable. The only effect of the domain is that the join $\sqcup_{\mathcal{WP}}$ translates to a widening operation $\nabla_{\mathcal{C}}^{l}$ on the child if one of the flags f_i^w is true.

Consider the CFG in Fig. 3.3 showing a common translation pattern of loops to machine code. The loop test condition is not at the loop entry but at the end of the loop. The domain state for the widening points domain $\mathcal{WP}$ is shown on the edges of the CFG. The rationale behind tracking a flag f^w instead of applying widening immediately on back-edges is

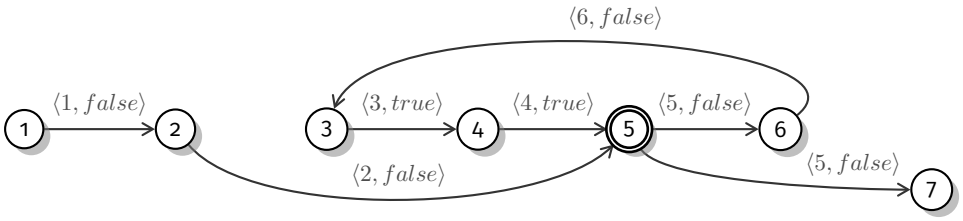


Figure 3.3: Common pattern for loops in machine code. The loop condition is at the end of the loop where the loop is also entered.

that widening should happen between the state that enters the loop and the state on the back-edge. Particularly, widening should be applied at the loop head. However, in the example in Fig. 3.3 the node reached by the back-edge l_6 to l_3 does not have an incoming edge. Widening at l_3 in the first iteration is applied between $\bot$ and the new state, thus postponing the extrapolation of widening till the second iteration. Our approach marks l_5 as widening node thus not delaying widening in such cases.

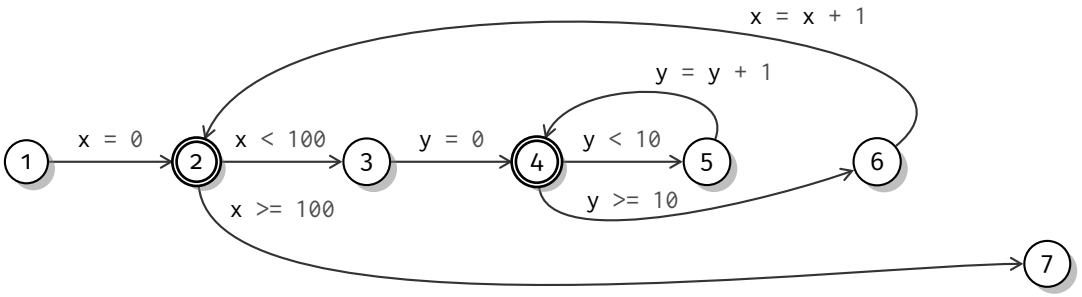


Figure 3.4: Nested loops with two widening points.

Note that this domain is more precise than the standard technique of marking nodes as widening points [Bou93] since widening is applied only after a back-edge. For instance, in Fig. 3.3, widening is only applied when updating node l_5 with a state from l_4 but no widening is applied when propagating the state from l_2 to l_5, as this path is not a back-edge. This feature to not apply widening on the edge that enters a loop [AS13] is especially crucial for the analysis of nested loops. Consider Fig. 3.4 showing the CFG of two nested loops. After the analysis infers the bounds for y using widening at l_4 the inner loop is exited and the outer loop is analyzed. After incrementing x and performing widening at l_2 the inner loop is entered again at widening point l_4. Now variable x has changed its value with regard to the previous iteration at this point. Applying widening on this edge l_3 to l_4 would lose the bounds for x. Yet, performing a join when entering the loop and applying widening only on the back-edge l_5 to l_4 maintains precision as x is not incremented in the inner loop. In particular, the value of x in the state propagated on the back-edge has not changed with regard to the value on the loop incoming edge and widening is a no-op for x.

One drawback of the simple heuristic to infer widening points is that function calls that jump to smaller addresses are recognized as back-edges. States propagated along these call-edges are widened at the function entry point. As some languages require function definitions to appear before their use, code compiled from such languages contains spurious widening points at function entries. If the call-string approach is used for the interprocedural analysis (see Sect. 2.7) the address labels $l \in \mathit{Lab}$ can be augmented to additionally contain the call-string $l \in \langle \mathit{Lab} \times K \rangle$. By taking the call-string into account procedure calls to lower addresses may not necessarily be flagged as back-edges.

The ability to add widening to an analysis without changing the fixpoint can also be carried over to various other widening heuristics, as detailed next.

3.3 Delaying Widening after Constant Assignments

It is widely acknowledged that computing a few iterations of a loop without widening can improve the precision of the computed fixpoint [Bla+02; Bag+05]. For instance, the program in Fig. 3.5 may set the variable y to 1 depending on some external event where `read()` may return the value of some sensor in a control software [Bla+03a; Cou+06]. We show an analysis given the domain stack $\mathcal{T} \triangleright (\mathcal{CR} \triangleright \mathcal{I})$ where $\mathcal{CR}$ is the domain tracking congruence information. The table in Fig. 3.5 shows how widening the state at step 7' with respect to that at step 2 yields $x \in [0,0] \nabla^l_{\mathcal{I}} [0,4] = [0,\infty]$ and $y \in [0,0] \nabla^l_{\mathcal{I}} [0,1] = [0,\infty]$. The former interval can be refined by the threshold $x \leq 103$ to $x \in [0,100]$ since the congruence $x \equiv 4$ is tracked, i.e. x is a multiple of four. The loop test `x < 100` then yields the precise value for x in step 8: $x \in [0,96]$. However, the upper bound for y is lost. The common approach to improve the precision is to delay widening [HPR97], that is, to compute another iteration of the loop using the joined state at step 7. For instance, after the second iteration, widening $y \in [0,1] \nabla^l_{\mathcal{I}} [0,1] = [0,1]$ will not change the value of y since it is stable.

3.3.1 Tracking Constant Assignments

Rather than fixing the number of times widening should be delayed, which is impossible to get right in all cases, we track if widening would alter variables that were set to a constant. To this end, we define a delaying domain given by the lattice $\langle \mathcal{WD} \triangleright \mathcal{C}, \sqsubseteq_{\mathcal{WD}}, \sqcup_{\mathcal{WD}}, \sqcap_{\mathcal{WD}} \rangle$ where $\mathcal{WD} : \wp(\mathit{Lab})$ is a set of program points with constant assignments. The transfer functions in Fig. 3.6 simply collect those program points that assign a constant to a variable. Performing widening on $\mathcal{WD}$ will check if this set has increased and, if so, perform a join instead of a widening. For example, in step 7 of Fig. 3.7, the set of constant assignment locations contains a new program point 5 when compared to the state at step 2. Thereby the domain $\mathcal{WD}$ will delay widening and the fixpoint computation is performing another iteration based on the state at step 7.

```
1 int x = 0;
2 int y = 0;
3 while (x < 100){
4   if (read())
5     y = 1;
6   x = x + 4;
7 }
8 ...
```

step	line		intervals x	intervals y	congruences	thresholds
1	2		$[0,0]$			
2	3		$[0,0]$	$[0,0]$		
3	4		$[0,0]$	$[0,0]$		$x \leq 99$
4	5		$[0,0]$	$[0,0]$		$x \leq 99$
5	6	$\sqcup$	$[0,0]$	$[0,1]$		$x \leq 99$
6	7		$[4,4]$	$[0,1]$		$x \leq 103$
7	3	$\sqcup$	$[0,4]$	$[0,1]$	$x \equiv 4$	$x \leq 103$
7'	3'	∇	$[\mathbf{0},\mathbf{100}]$	$[\mathbf{0},\infty]$	$x \equiv 4$	$x \leq 103$
8	4		$[0,96]$	$[0,\infty]$	$x \equiv 4$	$x \leq 99, x \leq 103$
9	5		$[0,96]$	$[0,\infty]$	$x \equiv 4$	$x \leq 99, x \leq 103$
10	6	$\sqcup$	$[0,96]$	$[0,\infty]$	$x \equiv 4$	$x \leq 99, x \leq 103$
11	7		$[4,100]$	$[0,\infty]$	$x \equiv 4$	$x \leq 103$
12	3	$\sqsubseteq$	$[0,100]$	$[0,\infty]$	$x \equiv 4$	$x \leq 103$
13	8		$[100,100]$	$[0,\infty]$		$x \leq 103$

Figure 3.5: Widening after one iteration loses the bound on y.

$$
\begin{aligned}
[\![l : x = e]\!]^{\mathcal{WD}}\langle d, c\rangle &= \langle d \cup \bar{l}, [\![l : x = e]\!]^{\mathcal{C}} c\rangle \text{ where } \bar{l} = \begin{cases} \{l\} & \text{if } e \in \mathbb{Z} \\ \emptyset & \text{otherwise} \end{cases} \\
\langle d_1, c_1\rangle \sqsubseteq_{\mathcal{WD}} \langle d_2, c_2\rangle &= c_1 \sqsubseteq_{\mathcal{C}} c_2 \\
\langle d_1, c_1\rangle \sqcup_{\mathcal{WD}} \langle d_2, c_2\rangle &= \langle d_1 \cup d_2, c_1 \sqcup_{\mathcal{C}} c_2\rangle \\
\langle d_1, c_1\rangle \nabla^l_{\mathcal{WD}} \langle d_2, c_2\rangle &= \langle d_1 \cup d_2, c\rangle \text{ where } c = \begin{cases} c_1 \nabla^l_{\mathcal{C}} c_2 & \text{if } d_2 \backslash d_1 = \emptyset \\ c_1 \sqcup_{\mathcal{C}} c_2 & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 3.6: Lattice and transfer functions for the delaying domain.

Note that the delaying of widening is possible at any iteration, not only the first. For example, consider the condition in line 4 being `if (x == 40)` in which case widening is applied in the first loop iteration and ensures that in the next iteration line 5 is reached by the analysis. Only now variable y is set, causing widening to be suppressed for the current iteration by the delaying domain, which in turn preserves the exact bounds for y.

3.3.2 Syntactic vs. Semantic Constants

In the transfer functions in Fig. 3.6 an assignment is added to the set of constant assignments if the right-hand-side expression e syntactically consists of only a constant. However, a slightly modified version of the example in Fig. 3.5 as shown in Fig. 3.8 a) motivates an extension to this simple approach. In this example the right-hand-side of the assignment `y = z + 1;` in line 6 is not a constant expression syntactically but evaluates to a constant expression. Similarly, the example in Fig. 3.8 b), as shown in [McM11], requires an evaluation of the assignment `y = 1 - y;` in line 4 before it can be recognized as a constant assignment. We implemented this so-called "semantic constants" recognition by querying the child domain for the value of an expression, that is, we check if $[\![e]\!]^{\mathcal{C}} c \in \mathbb{Z}$. Using this semantic method we were able to improve the precision for some of our benchmarks

```
1 int x = 0;
2 int y = 0;
3 while (x < 100){
4   if (read())
5     y = 1;
6   x = x + 4;
7 }
8 ...
```

step	line		intervals x	intervals y	congr.	thresholds	delayed
1	2		$[0,0]$				$\{1\}$
2	3		$[0,0]$	$[0,0]$			$\{1,2\}$
3	4		$[0,0]$	$[0,0]$		$x \leq 99$	$\{1,2\}$
4	5		$[0,0]$	$[0,0]$		$x \leq 99$	$\{1,2\}$
5	6	$\sqcup$	$[0,0]$	$[0,1]$		$x \leq 99$	$\{1,2,5\}$
6	7		$[4,4]$	$[0,1]$		$x \leq 103$	$\{1,2,5\}$
7	3	$\sqcup$	$[0,4]$	$[0,1]$	$x \equiv 4$	$x \leq 99, x \leq 103$	$\{1,2,5\}$
8	4		$[0,4]$	$[0,1]$	$x \equiv 4$	$x \leq 99, x \leq 103$	$\{1,2,5\}$
9	5		$[0,4]$	$[0,1]$	$x \equiv 4$	$x \leq 99, x \leq 103$	$\{1,2,5\}$
10	6	$\sqcup$	$[0,4]$	$[0,1]$	$x \equiv 4$	$x \leq 99, x \leq 103$	$\{1,2,5\}$
11	7		$[4,8]$	$[0,1]$	$x \equiv 4$	$x \leq 103$	$\{1,2,5\}$
12	3	$\sqcup$	$[0,8]$	$[0,1]$	$x \equiv 4$	$x \leq 103$	$\{1,2,5\}$
12'	3'	∇	$[\mathbf{0}, \mathbf{100}]$	$[\mathbf{0}, \mathbf{1}]$	$x \equiv 4$	$x \leq 103$	$\{1,2,5\}$
...			...			...	
17	3	$\sqsubseteq$	$[0,100]$	$[0,1]$	$x \equiv 4$	$x \leq 103$	$\{1,2,5\}$
18	8		$[100,100]$	$[0,1]$		$x \leq 103$	$\{1,2,5\}$

Figure 3.7: Delaying widening until y is stable maintains the precise bounds.

a)

```
1 int x = 0;
2 int y = 0;
3 int z = 0;
4 while (x < 100) {
5   if (...)
6     y = z + 1;
7   x = x + 4;
8 }
```

b)

```
1 int x = 0;
2 int y = 0;
3 while (x < 100) {
4   y = 1 - y;
5   x = x + 4;
6 }
```

Figure 3.8: Assignments in loops that require delayed widening to not lose precision.

as compared to the syntactic constants. In our experiments the delaying domain tracks 10 times more constant assignments than with the syntactic constants method. Most of these additional constants originate from assignments in incrementation code such as `i = i + 1;` that are considered constant during the first loop iteration. As a consequence widening is sometimes unnecessarily delayed for the first loop iteration which slows down the convergence but also improves precision.

3.3.3 Conclusion

Delaying widening on constant assignments is a good heuristic especially for code that implements state machines, e.g. code used in parsers. Hence, the delaying domain is necessary to prove correct the email address parsing code discussed in Sect. 9. Nevertheless, it is a heuristic and in general it is not possible to automatically decide when widening has to be delayed to maintain precision and a rapid fixpoint convergence. Most analyzers therefore employ user defined rules to delay widening for the first n loop iterations. We

show that for some common cases widening delays can be automatically inferred and widening can be suspended in any iteration of the fixpoint computation, not only the first iterations.

3.4 Widening with Thresholds

$$
\begin{array}{lcl}
[\![l : x = e]\!]^{\mathcal{T}}\langle t, c\rangle & = & \langle [p \mapsto \bar{l}_t \in t \mid x \notin \mathit{vars}(p)] \cup \\
& & \quad [\sigma^{-1}(p) \mapsto \bar{l}_t \cup \{l\} \mid p \mapsto \bar{l}_t \in t \wedge l \notin \bar{l}_t \wedge \sigma = [x/e] \wedge \sigma^{-1}(p) \text{ exists}], \\
& & \quad [\![l : x = e]\!]^{\mathcal{C}} c\rangle \\
[\![l : e \leq 0]\!]^{\mathcal{T}}\langle t, c\rangle & = & \langle \mathit{filter}(t[e \leq 0 \mapsto \emptyset]), [\![l : e \leq 0]\!]^{\mathcal{C}} c\rangle \\
\langle t_1, c_1\rangle \sqsubseteq_{\mathcal{T}} \langle t_2, c_2\rangle & = & c_1 \sqsubseteq_{\mathcal{C}} c_2 \\
\langle t_1, c_1\rangle \sqcup_{\mathcal{T}} \langle t_2, c_2\rangle & = & \langle \mathit{filter}(t, c), c\rangle \text{ where } t = \mathit{join}(t_1, t_2) \text{ and } c = c_1 \sqcup_{\mathcal{C}} c_2 \\
\langle t_1, c_1\rangle \nabla^l_{\mathcal{T}} \langle t_2, c_2\rangle & = & \langle \mathit{filter}(t, c), c\rangle \text{ where } t = \mathit{join}(\mathit{filter}(t_1, c_2), \mathit{filter}(t_2, c_1)) \text{ and} \\
& & c = [\![l : p_1]\!]^{\mathcal{C}} \ldots [\![l : p_n]\!]^{\mathcal{C}} (c_1 \nabla^l_{\mathcal{C}} c_2) \wedge p_i \in \{e_1 \leq 0, \ldots, e_n \leq 0\} = \mathit{dom}(t)
\end{array}
$$

Figure 3.9: Transfer and lattice functions for the threshold domain.

Widening is necessary to ensure termination when a fixpoint is computed over a domain of infinite height. One problem of widening is that the obtained fixpoint is almost always a post-fixpoint, that is, it is larger than the least fixpoint. This section shows how predicates occurring in tests can be used as *thresholds* to restrict the widened state, thereby often giving better results than narrowing can provide.

3.4.1 Tracking Widening Thresholds

Let *Pred* be a set of predicates that are used as conditions in tests. We require that the negation $\neg p$ of $p \in \mathit{Pred}$ exists and that $\neg p \in \mathit{Pred}$ where $\neg(\neg p) \equiv p$. In practice, we gather all tests convertible to linear inequalities and assume integer arithmetic, that is: $\neg(a_1 x_1 + \ldots + a_n x_n \leq c) \equiv a_1 x_1 + \ldots + a_n x_n \geq c + 1$.

The threshold domain is given by the lattice $\langle \mathcal{T} \triangleright \mathcal{C}, \sqsubseteq_{\mathcal{T}}, \sqcup_{\mathcal{T}}, \sqcap_{\mathcal{T}}\rangle$ where the universe $\mathcal{T} : \mathit{Pred} \dashrightarrow \wp(\mathit{Lab})$ is a partial map from redundant tests $p \in \mathit{Pred}$ to a set of program points. This tracked set contains the program points where p has been transformed due to assignments on a variable in p. We update $t \in \mathcal{T}$ to $t' = t[p \mapsto l] \in \mathcal{T}$ with $t'(p) = l$ and $t'(p) = t(q)$ for $q \neq p$. We use $[p \mapsto \ldots]$ to construct a new mapping and $\emptyset$ for the empty map. We enforce the invariant that all tests $p \in \mathit{Pred}$ are redundant in the child domain by applying $\mathit{filter} : \mathcal{T} \times \mathcal{C} \to \mathcal{T}$ which is defined as:

$$\mathit{filter}(t, c) = [p \mapsto t(p) \mid p \in \mathit{dom}(t) \wedge [\![p]\!]^{\mathcal{C}} c = c]$$

where $[\![p]\!]^{\mathcal{C}} c \in \mathcal{C}$ computes a state of the child domain in which the test p has been applied. Note that we may use the cheaper test $[\![\neg p]\!]^{\mathcal{C}} c = \bot_{\mathcal{C}}$ to check for redundancy of p instead of $[\![p]\!]^{\mathcal{C}} c = c$, although this test is less precise on some child domains $\mathcal{C}$.

Figure 3.9 presents the transfer functions and lattice operations of the threshold domain. An assignment $x = e$ at program point $l \in \mathit{Lab}$ is forwarded to the child. All thresholds that are not affected by the write to x are kept as is while predicates p that mention x are kept if an inverted substitution σ^{-1} exists where $\sigma = [x/e]$.
However, transforming a threshold infinitely many times in a loop can lead to non-termination as the transformed threshold remains redundant and is therefore repeatedly applied. Thus, we allow only one transformation for each program point. To ensure this we track a set of locations $\bar{l}$ where a threshold has been transformed. A repeated transformation at a program point l is disallowed by checking if the threshold has already been transformed at the current location l, that is, checking if $l \in \bar{l}$. For instance, consider the assignment `x = x + 1;` in line 4 in Fig. 3.1 where $t = [x \leq 99 \mapsto \emptyset] \in \mathcal{T}$ and $\mathcal{C} = \mathcal{I}$ the *interval* domain with $x \in [0, 1]$. With $\sigma = [x/x + 1]$, we obtain $\sigma^{-1} = [x/x - 1]$ and $\sigma^{-1}(x \leq 99) = x \leq 100$. Thus, the state after the assignment is $\langle [x \leq 100 \mapsto \{4\}], x \in [1, 2] \rangle$. Note that the resulting threshold is again 98 units away from the current state space. Indeed, transforming thresholds ensures that each threshold remains redundant. Transformation is necessary to infer the precise fixpoint at the loop head: $x \in [0, 100]$. Only using the constants from the loop conditions as thresholds, i.e. 99, does not give a stable state after widening.
If $x \notin \mathit{vars}(e)$ or if e is not linear, σ^{-1} does not exist and the threshold is removed. However, it is possible to leverage the child domain to ask for σ^{-1} and thus keep a threshold. For example, the *affine* domain or the *polyhedra* domain track relations between variables that can be used to generate σ^{-1}. Consider again, $t = [x \leq 99 \mapsto \emptyset] \in \mathcal{T}$ but $\mathcal{C} = \mathcal{A} \triangleright \mathcal{I}$ and the *affine* domain tracking $\{x = z\}$. For the assignment `x = 10;` there exists no σ^{-1} in the *threshold* domain but we obtain $\sigma^{-1} = [x/z]$ from the child domain and thus $t = [z \leq 99 \mapsto \emptyset] \in \mathcal{T}$. This feature is of great use for the analysis of machine code. Here, memory variables are loaded into registers before being tested in loop conditions. Because of reuse of registers the tested register is often overwritten by another memory load before the widening point. Hence, affine equalities are necessary to propagate thresholds from loop conditions to widening points.

The next transfer function is for tests. Redundant tests are collected as new thresholds whereas tests that happen to actually restrict the incoming state space c are removed by *filter*. Note that we split disequalities $e \neq 0$ in two inequalities: $e < 0$ and $0 < e$ before we consider them as thresholds. With respect to the lattice operation, the entailment test $\langle t_1, c_1 \rangle \sqsubseteq_{\mathcal{T}} \langle t_2, c_2 \rangle$ reduces to an entailment test on the child. The join $\langle t_1, c_1 \rangle \sqcup_{\mathcal{T}} \langle t_2, c_2 \rangle$ uses a function *join* that merges the program points tracking transformations point-wise as follows:

$$\mathit{join}(t_1, t_2) = \left[p \mapsto \bar{l}_{t1} \cup \bar{l}_{t2} \mid \bar{l}_{ti} = \begin{cases} t_i(p) & \text{if } p \in \mathit{dom}(t_i) \\ \emptyset & \text{otherwise} \end{cases} \right]_{p \in \mathit{dom}(t_1) \cup \mathit{dom}(t_2)}$$

Again, applying *filter* removes thresholds that are not redundant in $c_1 \sqcup_{\mathcal{C}} c_2$. Given the collected thresholds, widening $\langle t_1, c_1 \rangle \nabla^l_{\mathcal{T}} \langle t_2, c_2 \rangle$ is now able to refine the widened child state $c_1 \nabla^l_{\mathcal{C}} c_2$ by applying all the tracked predicates $e_1 \leq 0, \ldots, e_n \leq 0$ that are redundant in both domains.

We will now demonstrate an analysis using thresholds with the interval domain as child yielding the domain stack $\mathcal{T} \triangleright \mathcal{I}$. The code in Fig. 3.10 shows two nested loops as commonly used in matrix computations, e.g. performing a triangular scanning of a 2-dimensional array that represents the matrix. We omitted the array accesses for brevity. The table shows the analysis using the domain stack $\mathcal{T} \triangleright \mathcal{I}$. In the presentation of the thresholds domain state $\mathcal{T}$ we do not show the tracked set of transformation points for each threshold. However, in steps 9, 20 and 23 we remove thresholds that have already been transformed once at this point. In step 15 the threshold $j \leq i$ is removed due to j being overwritten by the assignment `j = 0;` in line 4.

```
1  int i = 0;
2  int j = 0;
3  while (i <= 99){
4    j = 0;
5    while (j <= i){
6      j = j + 1;
7    }
8    i = i + 1;
9  }
10 ...
```

step	line		intervals		thresholds
			i	j	
1	2		$[0,0]$		
2	3		$[0,0]$	$[0,0]$	
3	4		$[0,0]$	$[0,0]$	$i \leq 99$
4	5		$[0,0]$	$[0,0]$	$i \leq 99$
5	6		$[0,0]$	$[0,0]$	$i \leq 99, j \leq i$
6	7		$[0,0]$	$[1,1]$	$i \leq 99, j-1 \leq i$
7	5	$\sqcup$	$[0,0]$	$[0,1]$	$i \leq 99, j-1 \leq i$
7'	5'	∇	$[0,0]$	$[\mathbf{0},\mathbf{1}]$	$i \leq 99, j-1 \leq i$
8	6		$[0,0]$	$[0,0]$	$i \leq 99, j-1 \leq i, j \leq i$
9	7		$[0,0]$	$[0,1]$	$i \leq 99, j-1 \leq i$
10	5	$\sqsubseteq$	$[0,0]$	$[0,1]$	$i \leq 99, j-1 \leq i$
11	8		$[0,0]$	$[1,1]$	$i \leq 99, j-1 \leq i$
12	9		$[1,1]$	$[1,1]$	$i \leq 100, j \leq i$
13	3	$\sqcup$	$[0,1]$	$[0,1]$	$i \leq 100, j \leq i$
13'	3'	∇	$[0,100]$	$[\mathbf{0},\mathbf{100}]$	$i \leq 100, j \leq i$
14	4		$[0,99]$	$[0,100]$	$i \leq 99, i \leq 100, j \leq i$
15	5		$[0,99]$	$[0,0]$	$i \leq 99, i \leq 100$
16	6		$[0,99]$	$[0,0]$	$i \leq 99, i \leq 100, j \leq i$
17	7		$[0,99]$	$[1,1]$	$i \leq 99, i \leq 100, j-1 \leq i$
18	5	$\sqcup$	$[0,99]$	$[0,1]$	$i \leq 99, i \leq 100, j-1 \leq i$
18'	5'	∇	$[0,99]$	$[\mathbf{0},\mathbf{100}]$	$i \leq 99, i \leq 100, j-1 \leq i$
19	6		$[0,99]$	$[0,99]$	$i \leq 99, i \leq 100, j-1 \leq i, j \leq i$
20	7		$[0,99]$	$[1,100]$	$i \leq 99, i \leq 100, j \leq i$
21	5	$\sqsubseteq$	$[0,99]$	$[0,100]$	$i \leq 99, i \leq 100, j-1 \leq i$
22	8		$[0,99]$	$[100,100]$	$i \leq 99, i \leq 100, j-1 \leq i$
23	9		$[1,100]$	$[100,100]$	$i \leq 100, j \leq i$
24	3	$\sqsubseteq$	$[0,100]$	$[0,100]$	$i \leq 100, j \leq i$
25	10		$[100,100]$	$[100,100]$	$i \leq 100, j \leq i$

Figure 3.10: Applying widening with thresholds on nested loops.

The example shows that thresholds will be applied several times at a widening point if they are still redundant. The threshold $j \leq i$ transformed to $j - 1 \leq i$ is applied during the first analysis iteration of the inner loop at step 7' restricting j to $[0, 1]$. After another iteration the loop is stable in step 10 and we analyze the outer loop where widening with thresholds infers the bounds $i \in [0, 100]$ in step 13' due to the threshold $i \leq 100$. Note that we also track the threshold $j \leq i$ which restricts the value of j in the same widening step.

Next we enter the inner loop again with a greater value for i. In this second iteration of the inner loop we apply widening in step 18' and use the threshold $j - 1 \leq i$ a second time to restrict the values for variable j. Finally, the inner loop becomes stable (step 21)

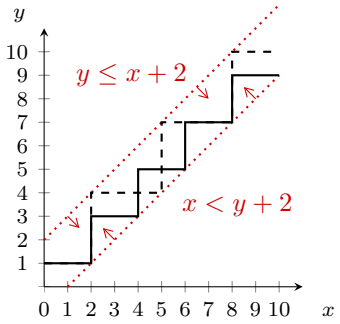
and the outer loop stabilizes next (step 24). We were able to infer the precise fixpoint for both loops.

Note that it is necessary to propagate thresholds in and out of loops to maintain the precision for variable j. In step 13' widening at the outer loop would remove the upper bound of j if not for the threshold $j \leq i$ which is applied in conjunction with $i \leq 100$. Hence, we are able to infer that $j \in [100, 100]$ outside of the loop in step 25.

However, allowing a repeated application of widening thresholds as long as they are redundant is not without problems as will be discussed next.

3.4.2 Ensuring Termination

```
1 int x = 0;
2 int y = 1;
3 while (true) {
4   if (x < y)
5     x = x + 2;
6   else
7     y = y + 2;
8 }
```



—— state development - - - with widening ····· widening thresholds

Figure 3.11: Infinite application of widening thresholds.

Given the domain as defined in Fig. 3.9, an infinite application of thresholds and thus non-termination can occur. Because thresholds are inequalities over arbitrary program variables it is possible to have thresholds that remain redundant throughout the whole analysis thus being applicable infinitely many times. The issue is demonstrated in Fig. 3.11. The diagram shows the state and inferred thresholds during the analysis (with domain stack $\mathcal{T} \triangleright \mathcal{I}$) of the program on the left. The variables x and y are incremented alternately in each loop iteration resulting in a staircase function. Widening extrapolates the state change in each iteration as shown by the dashed line. The problem is that the widening state never "escapes" the thresholds, that is, render the thresholds non-redundant. For each iteration it is possible to apply either the lower or the upper threshold. As only one variable is modified in each loop iteration the extrapolated value can be bound using the value of the other variable. Widening with thresholds over-approximates the staircase function but will not eventually stabilize.

This observation requires to add a limit to the application of thresholds. If thresholds cannot bind extrapolated values infinitely many times, widening will stabilize and termination is ensured. Instead of fixing the number of times that thresholds can be applied up-front, we choose the limit depending on program properties. The main idea is to allow the application of all thresholds at each widening point. If the state does not stabilize after

applying all the thresholds it probably won't stabilize by applying them again. Hence, a strategy is to allow threshold application during widening as many times as the number of thresholds tracked by the domain. To account for state changes due to nested loops that might require a re-application of a set of thresholds the limit we chose is this: number of widening points $\times$ widening thresholds. This upper limit is inferred by the threshold domain during the analysis. As both numbers are finite given a finite program, termination is guaranteed.

3.4.3 Limitations of Narrowing

```
1  int n = 0;
2  while (true) {
3    if (!read_sec())
4      continue;
5    if (n < 60) {
6      n = n + 1;
7    } else {
8      n = 0;
9    }
10 }
```

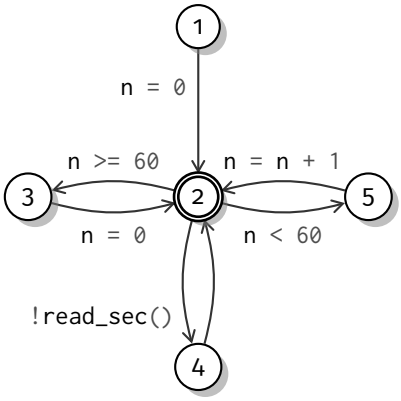


Figure 3.12: A loop whose fixpoint cannot be obtained by narrowing.

Widening with thresholds can find least fixpoints where narrowing cannot [HH12]. Consider the program in Fig. 3.12 that tracks the seconds within a minute. The loop repeatedly waits for a seconds signal that causes `read_sec` to return 1. The simplified CFG of the program contains three loops. After propagating $n = 0$ to node 2, the loops through node 3 and 4 are stable. The loop via node 5 yields $n \in [0, 1]$ in node 2 which is widened to $n \in [0, \infty]$. The threshold $n \leq 59$ is transformed by `n = n + 1;` to $n \leq 60$ and is applied after widening, yielding $n \in [0, 60]$. Narrowing, however, cannot deduce this fixpoint due to the cycle via node 4 that introduces an extensive path [HH12] through the loop. During the descending sequence of narrowing the values from nodes 3 and 5 are joined with the value from node 4. The contributions from nodes 3, 5 improve the precision with respect to widening, however, node 4 does not. Joining these states at node 2 does not lead to a more precise state than the widened state, thus defeating the purpose of narrowing.

Similarly, when wrapping semantics are used for the values of variables (see Sect. 2.5.3) narrowing cannot recover the precision loss introduced by wrapping [SMS11]. Consider the simple program in Fig. 3.13 analyzed using classic widening and narrowing. We assume the range of variable `n` to be in $-2^{31} \ldots 2^{31} - 1$ as is usual for signed 32 bits integer values. Widening `n` in step 4' to $[0, +\infty]$ and testing its range at the loop condition will result in the values $n \in [-2^{31}, 100]$ containing spurious negative values due to wrapping. Applying narrowing later in step 13 in the descending sequence cannot remove these introduced negative values. Using the *threshold* domain widening will extrapolate the values of `n` to

```
1 int n = 0;
2 while (n <= 100) {
3   n = n + 1;
4 }
```

step	line		intervals n
1	2		$[0, 0]$
2	3		$[0, 0]$
3	4		$[1, 1]$
4	2	$\sqcup$	$[0, 1]$
4'	2'	∇	$[0, +\infty]$
5	3	$\circlearrowleft$	$[-2^{31}, 100]$
6	4		$[-2^{31}+1, 101]$
7	2	$\sqcup$	$[-2^{31}+1, +\infty]$
7'	2'	∇	$[-\infty, +\infty]$
8	3	$\circlearrowleft$	$[-2^{31}, 100]$
9	4		$[-2^{31}+1, 101]$
10	2	$\sqsubseteq$	$[-\infty, +\infty]$
11	3	$\circlearrowleft$	$[-2^{31}, 100]$
12	4		$[-2^{31}+1, 101]$
13	2	Δ	$[\mathbf{-2^{31}+1, 101}]$
14	3		$[-2^{31}+1, 100]$
15	4		$[-2^{31}+2, 101]$
16	2	$\sqsubseteq$	$[-2^{31}+1, 101]$

Figure 3.13: A loop for which narrowing cannot recover precision loss due to wrapping $\circlearrowleft$.

$[0, 101]$ as the threshold $n \leq 100$ is transformed to $n \leq 101$ inside the loop. Subsequently applying the loop condition test $n \leq 100$ at the loop header does not cause wrapping of n. The *threshold* domain therefore allows to infer a more precise fixpoint than narrowing. Additionally, widening with thresholds is faster and stabilizes in fewer iterations, here 7 steps, because wrapping and thus a second widening (step 7') and narrowing is avoided.

Using narrowing to recover the precision loss of widening if often not possible. For example narrowing has problems with disequalities as loop conditions, a common artifact in machine code. Another problem is the spillage of the precision loss to other domains [SK06]. When combining numeric domains with symbolic domains, e.g. a points-to domain, narrowing may recover the precision loss in the numeric domain but cannot remove the spurious aliasing information that has been introduced due to widening.

Besides the precision improvements of thresholds widening over narrowing, we will show next that thresholds are a versatile concept.

3.4.4 Using Thresholds to Restrict Widening after Constant Assignments

As an alternative approach to the delaying domain presented in Sect. 3.3 we implemented a domain that synthesizes widening thresholds when seeing constant assignments. Hence, widening is not delayed on the whole domain while improving the precision for variables that are assigned constant values in loops. Consider again the example in Fig. 3.5 where the assignment y = 1 inside a loop leads to widening of y and thus loss of precision. Using a modified *delay* domain $\mathcal{WD}'$ with the *threshold* domain as child $\mathcal{WD}' \triangleright \mathcal{T}$ we can restrict widening on y using synthesized thresholds. The domain $\mathcal{WD}'$ observes the constant assignment and applies the assignment as test $y = 1$ to the child domain. The *threshold*

domain as child then splits up the test and tracks two thresholds: $y \leq 1$ and $y \geq 1$ for widening. When widening occurs at the loop header y, does not lose the upper bound because of the threshold $y \leq 1$.
Another positive effect of using thresholds to restrict widening is that we are more robust to wrong choices of widening points. Consider the example below in Fig. 3.14 for which an additional widening point is chosen for line 6. Applying widening on the **else**-branch as in $[1,1] \nabla_{\mathcal{T}}^{l} ([0,0] \sqcup [1,1]) = [-\infty, 1]$ will lose the lower bound of the loop counter as we only track the threshold $x < 10$. With synthesized thresholds for constant assignments we would additionally track the threshold $x \geq 0$ generated in line 1 and thus be more precise.

```
1 int i = 0;
2 while (i < 10) {
3   if (...) {
4     i++;
5   }
6   ...
7 }
```

Figure 3.14: Choosing line 6 as widening point may lose the lower bound of variable `i`.

Although this approach has benefits over the delaying domain it unnecessarily delays widening in the first iteration. Due to loop counters often being initialized with a constant the domain will introduce widening thresholds that restrict widening in the first iteration. Only in subsequent iterations will widening extrapolate the value of the loop counter. This shows the main drawback of the domain, namely the introduction of more iterations that slow down convergence. Some of our experiments showed a 4 times slowdown compared to the delaying domain. Additionally, as each constant assignment results in the application of a test on the child domain the slowdown is even more evident when combined with further domains that track tests. For example when combining this approach with the *phased* domain discussed next, which uses tests to track separate child states, we noticed an up to 10 times slower convergence. Furthermore, suppressing widening on the whole domain state as done by the delaying domain improves precision in some cases as it restricts widening for all modified variables not only the ones with constant assignments.

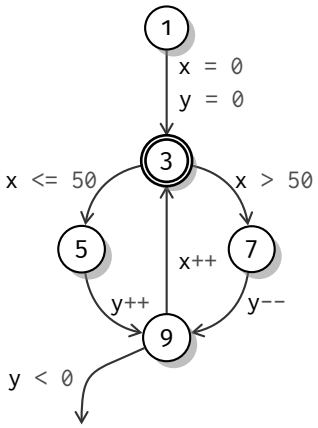
3.4.5 Conclusion

Widening with thresholds infers precise fixpoints that narrowing cannot infer and is faster as it requires fewer iterations. Especially when analyzing machine code where wrapping must be assumed everywhere, narrowing cannot recover the precision loss incurred by widening and widening with thresholds is required to recover even simple loop invariants. The propagation and transformation of thresholds is necessary to infer precise fixpoints but may lead to non-termination as it allows to restrict widening indefinitely. Hence, care must be taken to allow only finitely many applications of thresholds. A benefit of implementing

a)

```
1  int x = 0;
2  int y = 0;
3  while (true) {
4    if (x <= 50) {
5      y++;
6    } else {
7      y--;
8    }
9    if (y < 0)
10     break;
11   x++;
12 }
```

b)



c)

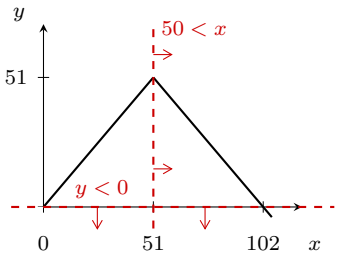


Figure 3.15: A loop containing phase transitions.

widening with thresholds as a separate abstract domain is that it allows to use relational predicates as thresholds on top of domains that cannot express relational information, such as intervals.

3.5 Guided Static Analysis

Numeric domains are usually convex approximations of the possible set of numeric values. One drawback of convexity is that joining two states can incur a precision loss that cannot later be recovered from. For example, the join of two intervals $[0, 5] \sqcup_{\mathcal{I}} [15, 20] = [0, 20]$ adds the spurious values $6, \ldots, 14$ and applying $x \leq 10$ to this state is less precise than applying it to the individual intervals. The idea of guided static analysis [GR07] is to avoid this kind of precision loss by identifying different phases of a loop and to track a separate state for each phase. The original proposal is formulated in terms of operations that restrict the CFG to increasingly larger sub-graphs and to perform widening/narrowing on these sub-graphs. In this section, we show that the same effect can be obtained by adding a cofibered *phase domain* into the domain hierarchy, thereby avoiding any modification to the fixpoint engine or to the handling of states.

Consider the loop in Fig. 3.15 a) that increments x, starting from zero. For the first fifty iterations, y is incremented while in the next fifty iterations y is decremented. The loop exits in the 102nd iteration when y becomes negative. The state space is depicted in Fig 3.15 c) where the two hyperplanes annotated with the predicates $p_x \equiv x > 50$ and $p_y \equiv y < 0$ mark the different phase transitions. In particular, observe that the three phases can be characterized by the predicates that hold: for the first phase $\neg p_x \wedge \neg p_y$ holds, for the second phase $p_x \wedge \neg p_y$ and for the third phase $p_x \wedge p_y$. Thus, rather than characterizing the loop phases by enabled sub-graphs of the CFG, we construct an abstract domain that tracks a different child domain for each feasible valuation of the predicates. In a child c that is tracked for the predicates $p_1, \ldots p_n$, we assume that each predicate p_i holds and, lest the

domain is imprecise, $[\![\neg p_i]\!]^{\mathcal{C}} c = \bot_{\mathcal{C}}$ for all $i \in [1, n]$. Thus, in the example, the predicates $\neg p_x \wedge \neg p_y$ hold in the state of the first phase c_1 and propagating c_1 over the edge from CFG node 3 to 7 in Fig. 3.15 b) yields an empty state, thereby simulating the fact that this sub-path of the CFG is disabled. Analogous, a state c_2 in which $p_x \wedge \neg p_y$ holds has the path $3 \to 5 \to 9$ disabled since it is guarded by $p_x \equiv x > 50$.

We implement the ideas of tracking several children depending on which predicates hold in the cofibered *phase* domain that is given by the lattice $\langle \mathcal{PH} \rhd \mathcal{C}, \sqsubseteq_{\mathcal{PH}}, \sqcup_{\mathcal{PH}}, \sqcap_{\mathcal{PH}} \rangle$ where $\mathcal{PH} : \mathcal{C} \times (\mathit{Pred} \times \mathcal{PH})^* \times \wp(\mathit{Pred})$ is a recursive type, representing a multi-way decision tree. A node in this tree $\langle c; p_1 : t_1; \ldots p_n : t_n; \bar{p} \rangle \in \mathcal{PH}$ contains a child domain c in which predicates $p_1, \ldots, p_n \in \mathit{Pred}$ do not hold. The node has n sub-trees $t_1, \ldots t_n \in \mathcal{PH}$ where p_i holds in t_i. The set $\bar{p} \subseteq \mathit{Pred}$ is a set of predicates that are unsatisfiable and represent phases that have not (yet) been entered. Before we detail the transfer and lattice functions, we consider the fixpoint computation in Fig. 3.16 using a domain stack $\mathcal{T} \rhd \mathcal{PH} \rhd \mathcal{A} \rhd \mathcal{I}$, that is, thresholds wrapping the phase domain, that wraps affine and intervals. We omitted the states for some lines (4, 11) as the propagated states do not differ from the lines before.

Initially, the phase domain contains a single child domain c_1 and no sub-trees as shown in step 1 of Fig. 3.16. The idea of the phased domain is to gather all unsatisfiable tests as possible phase predicates, adding them to the set $\bar{p}$. Thus, step 2 adds the predicate $x > 50$ and step 4 adds $y < 0$. Note that, unlike the threshold predicates, the phase predicates are not transformed. Once widening is applied in step 6', the subtree $t_2 = c_2; \{y < 0\}$ is added. This new subtree is immediately disabled in step 7 and 8 due to the test `x <= 50`. Analogously, only the subtree t_2 is enabled in steps 9 and 10. Both states are joined in step 11. Incrementing `x` to obtain step 12 poses the challenge that x in c_1 straddles the phase bound $x > 50$. Thus, the state $c_e = [\![\texttt{x++}]\!]^{\mathcal{A} \rhd \mathcal{I}} c_1$ is split into $c_1' = [\![x \leq 50]\!]^{\mathcal{A} \rhd \mathcal{I}} c_e$ and $\tilde{c} = [\![x > 50]\!]^{\mathcal{A} \rhd \mathcal{I}} c_e = \langle x = 51, y = 51 \rangle$. The latter is joined with the updated state of the subtree $[\![\texttt{x++}]\!]^{\mathcal{A} \rhd \mathcal{I}} c_2 = \langle x = 52, y = 50 \rangle$ yielding the downward slope $x + y = 102$ in the second line of step 12. Widening is applied again, thereby applying the threshold $y \geq 0$. The same state is propagated in steps 14 and 15 whereas the **`else`**-branch sees a larger state in step 16. Indeed, decrementing `y` in c_2 surpasses the phase threshold $y < 0$, thereby creating a third subtree $t_3 = c_3; \emptyset$ in step 17. Step 18 computes the joined state from which the state at loop exit is split off (step 19). Step 20 increments `x` which again propagates the point $\langle x = 51, y = 51 \rangle$ from c_1 to c_2 as for step 12. A fixpoint is observed in step 21.

The domain operations are formally defined in Fig. 3.17. We allow for several subtrees per program point to cater for sequences of **`if`**-statements. The assignment $l : x = e$ first computes the effect on the state in the current node c, yielding c_e, and its subtrees t_i, yielding $\tilde{c}_e^i$. The state space that spills over the phase predicates $p_1, \ldots p_n$ is cut off and merged into the respective parent or subtree. Any previously unsatisfiable phase predicates are checked against the new node state c'' and new subtrees $p_{n+1} : \langle c_{new}^{n+1} \rangle; \ldots p_{n+k} : \langle c_{new}^{n+k} \rangle;$ are added. Much simpler is the test $l : x \leq e$ which is applied recursively and is also added as phase predicate to $\bar{p}$ if it is unsatisfiable. The domain operations all rely on a function *compatible* that recursively adds missing phases by adding a subtree $p_i : \bot_{\mathcal{C}}; \ldots \bar{p}$ whenever $p_i : c_i; \ldots; \bar{p}$ only exists in the respective other domain. The lattice operations $t_1 \sqsubseteq_{\mathcal{PH}} t_2$ and $t_1 \sqcup_{\mathcal{PH}} t_2$ then reduce to a point-wise lifting of the respective operations on the child

step	line		intervals x	intervals y	affine	phased $c; p_1 : t_1, \dots p_n : t_n; \bar{p}$	thresholds
1	3		$[0,0]$	$[0,0]$	$x=0, y=0$	$c_1; \emptyset$	
2	5		$[0,0]$	$[0,0]$	$x=0, y=0$	$c_1; \{x>50\}$	$x \leq 50$
3	6		$[0,0]$	$[1,1]$	$x=0, y=1$	$c_1; \{x>50\}$	$x \leq 50$
4	11		$[0,0]$	$[1,1]$	$x=0, y=1$	$c_1; \{x>50, y<0\}$	$x \leq 50$
5	12		$[1,1]$	$[1,1]$	$x=1, y=1$	$c_1; \{x>50, y<0\}$	$x \leq 51, y \geq 0$
6	3	$\sqcup$	$[0,1]$	$[0,1]$	$x=y$	$c_1; \{x>50, y<0\}$	$x \leq 51, y \geq 0$
6'	3'	∇	$[0,50]$	$[0,50]$	$x=y$	$c_1; x>50 : t_2; \{y<0\}$	$x \leq 51, y \geq 0$
	3'		$[51,51]$	$[51,51]$	$x=51, y=51$	$c_2; \{y<0\}$	
7	5		$[0,50]$	$[0,50]$	$x=y$	$c_1; x>50 : t_2; \{y<0\}$	$x \leq 50, x \leq 51, y \geq 0$
	5					$\bot_{\mathcal{A}\triangleright\mathcal{I}}; \{y<0\}$	
8	6		$[0,50]$	$[1,51]$	$x=y-1$	$c_1; x>50 : t_2; \{y<0\}$	$x \leq 50, x \leq 51, y \geq 1$
	6					$\bot_{\mathcal{A}\triangleright\mathcal{I}}; \{y<0\}$	
9	7					$\bot_{\mathcal{A}\triangleright\mathcal{I}}; x>50 : t_2; \{y<0\}$	$x \leq 50, x \leq 51, y \geq 0$
	7		$[51,51]$	$[51,51]$	$x=51, y=51$	$c_2; \{y<0\}$	
10	8					$\bot_{\mathcal{A}\triangleright\mathcal{I}}; x>50 : t_2; \{y<0\}$	$x \leq 50, x \leq 51, y \geq -1$
	8		$[51,51]$	$[50,50]$	$x=51, y=50$	$c_2; \{y<0\}$	
11	9	$\sqcup$	$[0,50]$	$[1,51]$	$x=y-1$	$c_1; x>50 : t_2; \{y<0\}$	$\dots, y \geq -1, y \geq 1$
	9	$\sqcup$	$[51,51]$	$[50,50]$	$x=51, y=50$	$c_2; \{y<0\}$	
12	12		$[1,50]$	$[1,50]$	$x=y$	$c_1; x>50 : t_2; \{y<0\}$	$x \leq 51, \dots$
	12		$[51,52]$	$[50,51]$	$x+y=102$	$c_2; \{y<0\}$	
13	3	$\sqcup$	$[0,50]$	$[0,50]$	$x=y$	$c_1; x>50 : t_2; \{y<0\}$	$x \leq 51, y \geq -1, y \geq 0$
	3	$\sqcup$	$[51,52]$	$[50,51]$	$x+y=102$	$c_2; \{y<0\}$	
13'	3'	$\sqsubseteq$	$[0,50]$	$[0,50]$	$x=y$	$c_1; x>50 : t_2; \{y<0\}$	$x \leq 50, x \leq 51, \dots$
	3'	∇	$[51,102]$	$[0,51]$	$x+y=102$	$c_2; \{y<0\}$	
14	5	$\sqsubseteq$	$[0,50]$	$[0,50]$	$x=y$	$c_1; x>50 : t_2; \{y<0\}$	$x \leq 50, x \leq 51, \dots$
	5					$\bot_{\mathcal{A}\triangleright\mathcal{I}}; \{y<0\}$	
15	6	$\sqsubseteq$	$[0,50]$	$[1,51]$	$x=y-1$	$c_1; x>50 : t_2; \{y<0\}$	$\dots$
	6					$\bot_{\mathcal{A}\triangleright\mathcal{I}}; \{y<0\}$	
16	7					$\bot_{\mathcal{A}\triangleright\mathcal{I}}; x>50 : t_2; \{y<0\}$	$\dots$
	7		$[51,102]$	$[0,51]$	$x+y=102$	$c_2; \{y<0\}$	
17	8					$\bot_{\mathcal{A}\triangleright\mathcal{I}}; x>50 : t_2; \{y<0\}$	
	8		$[51,101]$	$[0,50]$	$x+y=101$	$c_2; y<0 : t_3; \emptyset$	$\dots$
	8		$[102,102]$	$[-1,-1]$	$x=102, y=-1$	$c_3; \emptyset$	
18	9	$\sqcup$	$[0,50]$	$[1,51]$	$x=y-1$	$c_1; x>50 : t_2; \{y<0\}$	
	9	$\sqcup$	$[51,100]$	$[0,50]$	$x+y=101$	$c_2; y<0 : t_3; \emptyset$	
	9	$\sqcup$	$[102,102]$	$[-1,-1]$	$x=102, y=-1$	$c_3; \emptyset$	$\dots$
19	10		$[102,102]$	$[-1,-1]$	$x=102, y=-1$	$c_3; \emptyset$	$\dots$
20	12		$[1,50]$	$[1,50]$	$x=y$	$c_1; x>50 : t_2; \{y<0\}$	
	12		$[51,102]$	$[0,51]$	$x+y=102$	$c_2; y<0 : \bot_{\mathcal{A}\triangleright\mathcal{I}}; \emptyset$	$\dots$
21	3	$\sqsubseteq$	$[1,50]$	$[1,50]$	$x=y$	$c_1; x>50 : t_2; \{y<0\}$	
	3	$\sqsubseteq$	$[51,102]$	$[0,51]$	$x+y=102$	$c_2; y<0 : \bot_{\mathcal{A}\triangleright\mathcal{I}}; \emptyset$	$\dots$

Figure 3.16: Computing the fixpoint for the example in Fig. 3.15.

domain. Widening is defined similarly to join, however, the phase boundaries are enforced after widening in order to ensure that the various states remain separated by the phase predicates. If widening makes unsatisfiable phase predicates satisfiable, new subtrees are added.

3.6 Experimental Results

We evaluated the presented domains in our analyzer for machine code [SMS11], using a domain stack $\mathcal{WP} \triangleright \mathcal{WD} \triangleright \mathcal{T} \triangleright \mathcal{A} \triangleright \mathcal{CR} \triangleright \mathcal{I}$ where $\mathcal{CR}$ tracks congruences, except for examples marked with * that use $\mathcal{WP} \triangleright \mathcal{WD} \triangleright \mathcal{T} \triangleright \mathcal{PH} \triangleright \mathcal{A} \triangleright \mathcal{CR} \triangleright \mathcal{I}$. The benchmarks in

$[\![l : x = e]\!]^{\mathcal{T}}\langle c; p_1 : t_1; \ldots p_n : t_n; \bar{p}\rangle =$
  let $c_e = [\![l : x = e]\!]^{\mathcal{C}} c$ and $c' = [\![l : \neg p_i]\!]^{\mathcal{C}} \cdots [\![l : \neg p_n]\!]^{\mathcal{C}} c_e$ and $c_i = [\![l : p_i]\!]^{\mathcal{C}} c_e$
  and $\langle \tilde{c}_e^i; \mathit{part}^i\rangle = [\![l : x = e]\!]^{\mathcal{T}} t_i$
  and $\tilde{c}_{res}^i = c_i \sqcup_{\mathcal{C}} [\![l : p_i]\!]^{\mathcal{C}} \tilde{c}_i$ and $c'' = c' \sqcup_{\mathcal{C}} [\![l : \neg p_1]\!]^{\mathcal{C}} \tilde{c}_e^1 \sqcup_{\mathcal{C}} \cdots \sqcup_{\mathcal{C}} [\![l : \neg p_n]\!]^{\mathcal{C}} \tilde{c}_e^n$
  and $\bar{p}^{red} = \{p \in \bar{p} \mid [\![l : \neg p]\!]^{\mathcal{C}} c'' = \bot_{\mathcal{C}}\}$ and $\langle p_{n+1}, \ldots p_{n+k}\} = \bar{p} \setminus \bar{p}^{red}$
  and $c_{res} = [\![l : \neg p_{n+1}]\!]^{\mathcal{C}} \cdots [\![l : \neg p_{n+k}]\!]^{\mathcal{C}} c''$and $c_{new}^{n+j} = [\![l : p_{n+j}]\!]^{\mathcal{C}} c''$ for $j = 1 \ldots k$
  in $\langle c_{res}; p_1 : \langle \tilde{c}_{res}^1; \mathit{part}^1\rangle; \ldots p_n : \langle \tilde{c}_{res}^n; \mathit{part}^n\rangle; p_{n+1} : \langle c_{new}^{n+1}\rangle; \ldots p_{n+k} : \langle c_{new}^{n+k}\rangle; \bar{p}^{red}\rangle$
$[\![l : e \leq 0]\!]^{\mathcal{T}}\langle c; p_1 : t_1; \ldots p_n : t_n; \bar{p}\rangle =$
  let $\langle \tilde{c}_e^i : \mathit{part}^i\rangle = [\![l : e \leq 0]\!]^{\mathcal{T}} t_i$ and $c_e = [\![l : e \leq 0]\!]^{\mathcal{C}} c$
  in if $\bigwedge_{i=1}^n \tilde{c}_e^i = \bot_{\mathcal{T}} \wedge c_e = \bot_{\mathcal{C}}$ then $\bot_{\mathcal{T}}$ else
  $\langle c_e; p_1 : \langle \tilde{c}_e^1; \mathit{part}^1\rangle; \ldots p_n : \langle \tilde{c}_e^n; \mathit{part}^n\rangle;$ if $[\![l : e > 0]\!]^{\mathcal{C}} c = \bot_{\mathcal{C}}$ then $\bar{p} \cup \{e \leq 0\}$ else $\bar{p}\rangle$
$\langle c_1; \mathit{part}_1\rangle \sqsubseteq_{\mathcal{T}} \langle c_2; \mathit{part}_2\rangle =$
  let $\langle\langle p_1^1 : t_1^1; \ldots p_n^1 : t_n^1; \bar{p}^1\rangle, \langle p_1^2 : t_1^2; \ldots p_n^2 : t_n^2; \bar{p}^2\rangle\rangle = \mathit{compatible}(\mathit{part}_1, \mathit{part}_2)$
  in $c_1 \sqsubseteq_{\mathcal{C}} c_2 \wedge \bigwedge_{i=1}^n t_i^1 \sqsubseteq_{\mathcal{T}} t_i^2$
$\langle c_1; \mathit{part}_1\rangle \sqcup_{\mathcal{T}} \langle c_2; \mathit{part}_2\rangle =$
  let $\langle\langle p_1^1 : t_1^1; \ldots p_n^1 : t_n^1; \bar{p}^1\rangle, \langle p_1^2 : t_1^2; \ldots p_n^2 : t_n^2; \bar{p}^2\rangle\rangle = \mathit{compatible}(\mathit{part}_1, \mathit{part}_2)$
  in $\langle c_1 \sqcup_{\mathcal{C}} c_2; p_1^1 : t_1^1 \sqcup_{\mathcal{T}} t_1^2; \ldots p_n^1 : t_n^1 \sqcup_{\mathcal{T}} t_n^2; \bar{p}^1\rangle$
$\langle c_1; \mathit{part}_1\rangle \nabla_{\mathcal{T}}^l \langle c_2; \mathit{part}_2\rangle =$
  let $\langle\langle p_1^1 : t_1^1; \ldots p_n^1 : t_n^1; \bar{p}^1\rangle, \langle p_1^2 : t_1^2; \ldots p_n^2 : t_n^2; \bar{p}^2\rangle\rangle = \mathit{compatible}(\mathit{part}_1, \mathit{part}_2)$
  and $c_e = [\![l : \neg p_1^1]\!]^{\mathcal{C}} \cdots [\![l : \neg p_n^1]\!]^{\mathcal{C}} c_1 \nabla_{\mathcal{T}}^l c_2$ and $\tilde{c}_e^i; \mathit{part}^i\rangle = \langle [\![l : p_i^1]\!]^{\mathcal{T}} (t_i^1 \nabla_{\mathcal{T}}^l t_i^2)$;
  and $\bar{p}^{red} = \{p \in \bar{p} \mid [\![l : \neg p]\!]^{\mathcal{C}} c'' = \bot_{\mathcal{C}}\}$ and $\langle p_{n+1}, \ldots p_{n+k}\} = \bar{p} \setminus \bar{p}^{red}$
  and $c_{res} = [\![l : \neg p_{n+1}]\!]^{\mathcal{C}} \cdots [\![l : \neg p_{n+k}]\!]^{\mathcal{C}} c_e$and $c_{new}^{n+j} = [\![l : p_{n+j}]\!]^{\mathcal{C}} c_e$ for $j = 1 \ldots k$
  in $\langle c_{res}; p_1^1 : \langle \tilde{c}_e^1; \mathit{part}^1\rangle \ldots p_n^1 : \langle \tilde{c}_e^n; \mathit{part}^n\rangle; p_{n+1} : \langle c_{new}^{n+1}\rangle; \ldots p_{n+k} : \langle c_{new}^{n+k}\rangle; \bar{p}^{red}\rangle$

Figure 3.17: Transfer and lattice functions for the phase domain.

Fig. 3.18 represent challenging loops that were mostly put forth in the literature [GR06a; HH12; LJG11]. Our own "nested loops" increase two variables, with various bounds and resets. Examples marked with "(mod)" are modifications of the same problem. These include changing the loop exit conditions in nested loops or adding loop exit points (**`break`**, **`continue`**), adding further variables or loop counter increments on separate paths through the loop. We also modified examples, where applicable, to contain non-deterministic paths and multiple widening points inside the loops, both features that can be found in irreducible graphs. The measurements are as follows: *insns.* gives the number of instructions in the program; *#wp* is the number of widening back-edges; *steps* the number of instructions the analyzer evaluated to reach the fixpoint; *iter.* is the maximum number of fixpoint iterations at any program point; *exact* denotes if the best interval bounds were found; *time* shows the analysis time in milliseconds. The time shown is the median of 2000 runs on a 3.2 GHz Core i7 machine running Linux.

We compared our results with those of the `Interproc` and `ConcurInterproc` analyzers [LJG11]. For both we used polyhedra with congruences which is the domain that is closest to our domain stack. `Interproc` can count iteration steps but only uses narrowing to refine the post-fixpoint. The table shows that the number of iterations in our analysis is usually

example	our analysis time	insns.	#wp	steps	iter.	exact	interproc iter.	exact	conc. exact
simple loop Fig. 3.1	7	14	1	23	2	✓	3+1	✓	✓
nested loops random	7	20	2	42	3	✓	5+2	✓	✓
nested loops random (mod)	7	20	2	43	3	✓	5+2		✓
nested loops medium	4	18	2	39	3	✓	4+2		✓
nested loops hard	4	19	2	40	3	✓	4+2		✓
nested loops hard (mod 1)	5	19	2	48	4	✓	4+2		✓
nested loops hard (mod 2)	10	19	2	96	8		4+3		✓
Halbwachs Fig. 1a [HH12]	2	9	1	15	2	✓	3+2	✓	✓
Halbwachs Fig. 1b	5	17	2	51	4	✓	4+2		✓
Halbwachs Fig. 1b (mod)	4	17	2	49	4	✓	5+2		✓
Halbwachs Fig. 2a	2	10	1	23	3	✓	3+2	✓	✓
Halbwachs Fig. 2b	3	12	1	28	3	✓	4+1		✓
Halbwachs Fig. 2b (mod)	4	14	1	46	4	✓	3+2	✓	✓
Halbwachs Fig. 4	18	18	2	84	9		4+2		✓
*Gopan Fig. 1a [GR06a]	15	14	1	36	4	✓	5+2		
*Gopan Fig. 1a (mod)	13	14	1	33	4	✓	5+1		
Chaouch Fig. 2 [LJG11]	2	12	1	19	2	✓	3+2		✓
Chaouch Fig. 3	7	23	1	83	6	✓	4+2		✓
Chaouch Fig. 3 (mod)	4	25	3	66	3	✓	4+1		
Chaouch Fig. 4	2	10	1	22	3	✓	4+1		✓
Chaouch Fig. 5	2	17	2	51	4	✓	4+2		✓
*Chaouch Fig. 6	13	14	1	36	4	✓	5+2		

Figure 3.18: Widening examples.

smaller than that of `Interproc`, even without the narrowing iterations (which are indicated by $+n$). In all benchmarks, we used no explicit delay. Since most examples are engineered not to work with narrowing, the least fixpoint is rarely obtained. `ConcurInterproc` uses a pre-analysis to infer thresholds but does not perform an iteration count. Assuming that these thresholds are applied to the states after widening, `ConcurInterproc` must require at least as many iterations as the number of upward iterations of `Interproc`. Our precision and that of the threshold widening in `ConcurInterproc` match. Entries where our analysis is less precise than `ConcurInterproc` require a polyhedral invariant that our domain stack cannot express. For the examples requiring disjunctive invariants `ConcurInterproc` is imprecise in that it infers, for example, $x \in [51, 102]$ for line 10 in Fig. 3.15. Our benchmarks used for `Interproc` are available on-line at `http://tinyurl.com/cwdg5qr`.

3.7 Related Work

Many authors address the task of improving widening, be it for specific domains such as polyhedra [Bag+05; JM09], or by altering the way fixpoints are inferred. With respect to the latter, Halbwachs pioneered the idea of using thresholds to refine widening and to delay widening [HPR97]. In the so-called "widening up-to" or "limited widening" thresholds over variables are created from a set of constants, an idea later successfully used in the large: The most common way to extract thresholds is to use the constants occurring in loop conditions [HPR97; Bal07]. Others [Bla+02; Kir+12; LL11], additionally consider user

provided constants for common numeric boundaries, e.g. $-1, 0, 1$, arithmetic progressions, e.g. powers of 2, or the range of a variable given by its type.

Chaouch et al. [LJG11] recently proposed a pre-analysis to infer thresholds automatically by extracting individual inequalities from the resulting state after two iterations. While their approach is similar to our thresholds domain they use the numeric domain to extract the thresholds. If the numeric domain is expressive, e.g. the polyhedron abstract domain [CH78], it has the benefit of inferring more complex thresholds than our *threshold* domain. On the other hand, for non-relational domains such as intervals the extracted thresholds are less precise, allowing only constants as thresholds. In the benchmarks in Fig. 3.18 we thus compared our analyzer using intervals to their analyzer using polyhedra.

Rather than extracting thresholds, widening with landmarks [SK06] measures the distance of the current state space to the loop condition and extrapolates the state space accordingly. Both approaches require special domain functions, e.g. for widening, and are thus not easily portable between different numeric domains. Our *threshold* domain is easier to use as it is agnostic to the underlying domain and infers the possible thresholds by itself.

Bagnara et al. generalizes the idea of delaying widening from [Bla+02] by using a finite number of tokens: a widening may use any non-terminating strategy if there are still tokens to consume [Bag+05]. Rather than requiring the user to fix the number of delays or provide a set of tokens, our *delay* domain in Sect. 3.3 uses program points denoting new assignments instead of tokens, thereby ensuring termination without depending on user input. This allows for interleaving sequences of widenings and delays depending on new program behavior.

One challenge of using convex numeric domains is the problem of spillage of state into branches of the program or behaviors of the transfer function that cannot be recovered from by narrowing. In this context, Halbwachs et al. [HH12] propose to re-start the analysis at a different pre-fixpoint from which widening and narrowing infer a new post-fixpoint. The intersection of the previous and the new post-fixpoint is still sound (still a post-fixpoint) and may be more precise. As they use classic widening for the ascending sequences combined with narrowing for the descending sequences they still suffer from loss of precision in case of wrapping. However, as their last example shows, due to the projection of the post-fixpoint onto a pre-fixpoint they are able to infer complex invariants that don't appear as conditions in the program. This is mainly due to the ability to partially remove the precision loss introduced by the join. Hence, combining their approach with our thresholds based widening would gather the benefits of both techniques while also fixing certain shortcomings. Implementing an abstract domain that emulates their approach, however, remains for future work.

Amato et al. improve the widening precision by not applying widening on edges that enter the loop [AS13]. Our *widening points* domain in Sect. 3.2 realizes the same by applying widening on back-edges only. They extend this "localized widening" with a "localized narrowing" that recovers precision loss in inner loops by restarting a new analysis that ignores previously inferred values. This is similar to Halbwachs' method and is able to discover invariants not present as loop conditions. However, their method requires

modifications to the fixpoint and knowledge about the CFG structure, such as strongly connected components and a weak topological order [Bou93]. In the analysis of machine code the CFG is not known up-front thus making such modifications difficult.

Rather than removing the spillage into branches, Gopan et al. propose to avoid spillage into currently unreachable branches immediately after widening [GR06a]. They require one state to determine which branches of the loops are enabled and a second state to compute widening and narrowing on the enabled part of the loop. Instead of duplicating the analysis cost by tracking a second abstract state, the authors later propose to directly track which parts of the CFG are enabled [GR07]. They generalize their idea to track different states for each phase, that is, for each set of enabled branches in a loop. This is implemented by analyzing modified versions of the CFG.
While none of the three approaches require changes to the transfer functions of the domains as was the case for the original widening with thresholds [HPR97], each approach requires intrusive changes to the fixpoint engine and the handling of states. Our *phase* domain in Sect. 3.5 has the same functionality as the Guided Static Analysis approach [GR07] but requires no changes to the way states are handled. Interestingly, the transfer functions of our *phase* domain are similar to those of the decision tree domain of Astrée [Cou+06]. However, the latter tracks Boolean flags as predicates and requires a user-supplied limit to avoid an exponential explosion. Since our domain creates a tree that mirrors the finite branching inside the loop body, its size is always limited by the program.

3.8 Conclusion

Implementing widening strategies as abstract domains is beneficial due to its modularity and independence of the fixpoint engine. Due to the cofibered construction our approach does not require modifications to the transfer functions of other domains. Complex widening strategies can thus be retrofitted to existing fixpoint analyses.

Our approach provides equal or better precision combined with fewer iterations required to obtain stability. One important aspect is that we do not suffer from the shortcomings of narrowing (see Sect. 3.4.3) as thresholds are applied immediately during the widening. Another benefit is that our domains do not have to implement narrowing at all as thresholds are tests that can be applied using domain transfer functions.

After showing various improvements to widening we will next consider improvements to counter the precision loss in convex numeric domains. Contrary to widening, the join does not suffer from precision loss due to extrapolation. However, numeric domains are compact representations of a set of values. This compactness is achieved by tracking convex approximations. Joining two states thereby requires a convex approximation of the union and may introduce imprecision. The next chapter shows how the precision loss of the approximation can be recovered from and how even non-convex spaces can be expressed by a generic extension of convex numeric domains.

4 The *Predicate* Abstract Domain

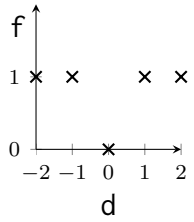
4.1 Introduction

Verification by means of a reachability analysis is based on abstract domains that over-approximate the possible concrete states that a program can reach. The forte of abstract domains is their ability to synthesize new invariants that are not present in the program. However, their inherent approximation may mean that the invariant required to verify a program cannot always be deduced. On the contrary, the strength of predicate abstraction used in software model checking is that predicates precisely partition the state space of a program. The challenge here is to synthesize new predicates that eventually suffice to verify a program. This work combines the benefits of both approaches: we synthesize new predicates by observing the precision loss in numeric domains and refine the precision of the numeric domains using the predicates. Our technique is particularly useful for expressing non-convex invariants that are commonly lost when using off-the-shelf numeric abstract domains that are based on convex approximations.

a)

```
1 f = d!=0;
2 ...
3 if (f) {
4   assert(d != 0);
5   y = x / d;
6 }
```

b)



c)

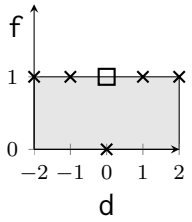


Figure 4.1: Avoiding a division by zero. The state space and its approximation.

The necessity of non-convex invariants is illustrated by the C code in Fig. 4.1 a). Here, line 1 computes a flag `f` that is true if the divisor `d` of the expression in line 5 is non-zero. Assuming that the initial value of `d` lies in $[-2, 2]$, the possible values when evaluating the conditional are shown in Fig. 4.1 b). Abstracting this set of discrete points using, say, the abstract domain of intervals yields the state in Fig. 4.1 c). This state space is too imprecise to deduce that `d` is non-zero if `f` is one. As a consequence, testing that f is one in line 3 does not restrict the abstract state sufficiently to show that the assertion holds.

Interestingly, analyzing the same example using predicate abstraction does not suffer from this imprecision as non-convex spaces can naturally be represented using disjuncts. In

the example, a predicate $p_f \equiv d \neq 0$, which is equivalent to the disjunction $d \leq -1 \vee d \geq 1$, suffices to verify the assertion since testing f in line 3 results in p_f being true.

A common approach to enriching numeric abstract domains to allow expressing non-convex states is to use disjunctive completion [CC79a], that is, a set of states. In particular, several works have proposed some variant of a binary decision-diagram (BDD) where decision nodes are labeled with predicates and the leaves are abstract domains [GC10a; MR05]. A similar effect is obtained by duplicating the control flow graph (CFG) for each subset of satisfied predicates [GR07; San+06]. In both settings, the number of numeric domains that are tracked may be exponential in the number of predicates. Our work improves over this setup by combining classic predicate abstraction [Bal+01] with a single numeric domain, thereby avoiding this exponential duplication of the numeric state. In particular, we present a generic *combinator domain* that is parameterized over any numeric abstract domain and allows any predicate expressible by the abstract domain. We thereby also generalize over bespoke domains that explicitly track specific disjunctive information, such as disequalities [PH07].

4.2 Definition of the Domain

We present our *predicate* domain $\mathcal{P}$ as a cofibered domain [Ven96] combined with a child domain $\mathcal{C}$, yielding a stack of domains $\mathcal{P} \triangleright \mathcal{C}$. A state $\langle \bar{\iota}, c \rangle$ contains the individual domain states $\bar{\iota} \in \mathcal{P}$ and $c \in \mathcal{C}$. The predicate domain is given by the lattice $\langle \mathcal{P} \triangleright \mathcal{C}, \sqsubseteq_{\mathcal{P}}, \sqcup_{\mathcal{P}}, \sqcap_{\mathcal{P}} \rangle$ where the universe $\mathcal{P} : \wp(Pred \times Pred)$ is a finite set of implications $p_1 \to p_2$ over predicates $p_i \in Pred$ as defined in Fig. 2.5 in Sect. 2.2. Predicates relate linear expressions over the program variables $\mathcal{X}_V$ using a comparison operator $\bowtie$. This set of operators is closed under negation so that the universe of predicates is closed under negation. The choice of implications between only two predicates allows for a simple yet effective propagation of information, as detailed in the next section.

4.3 Transfer Functions and Reductions

This section details the transfer functions and presents the flow of information between the predicate domain and the numeric child domains.

4.3.1 Transfer Functions

The transfer functions of the combined domain state $\langle \bar{\iota}, c \rangle \in \mathcal{P} \triangleright \mathcal{C}$ are given in Fig. 4.2. In general, a transfer function $[\![l]\!]^{\mathcal{P}} \langle \bar{\iota}, c \rangle$ applies the corresponding transfer function on the child domain $c \in \mathcal{C}$, yielding $\langle \bar{\iota}', [\![l]\!]^{\mathcal{C}} c \rangle$ where $\bar{\iota}'$ is the new state of the predicate domain. We distinguish three forms of assignments. The first, $[\![x = a \bowtie b]\!]^{\mathcal{P}}$, assigns the result of a comparison to a variable x. Here, the predicate domain removes any predicate that mentions x and adds new predicates based on the comparison. We assume that x is set to one if test $a \bowtie b$ holds and to zero otherwise. Thus, the predicates $x = 0$ and $x = 1$ are used

$$
\begin{aligned}
&[\![x = a \bowtie b]\!]^{\mathcal{P}}\langle \bar{\iota}, c\rangle = \langle \bar{\iota}', [\![x = a \bowtie b]\!]^{\mathcal{C}} c\rangle \\
&\quad \text{where } \bar{\iota}' = \{p \to q \in \bar{\iota} \mid x \notin \mathit{vars}(p) \cup \mathit{vars}(q)\} \\
&\quad \cup \{x = 1 \to a \bowtie b, x = 0 \to a \not\bowtie b, a \bowtie b \to x = 1, a \not\bowtie b \to x = 0\} \\
&[\![x = \mathit{NonLin}]\!]^{\mathcal{P}}\langle \bar{\iota}, c\rangle = \langle \bar{\iota}', [\![x = \mathit{NonLin}]\!]^{\mathcal{C}} c\rangle \\
&\quad \text{where } \bar{\iota}' = \{p \to q \in \bar{\iota} \mid x \notin \mathit{vars}(p) \cup \mathit{vars}(q)\} \\
&[\![x = \mathit{Lin}]\!]^{\mathcal{P}}\langle \bar{\iota}, c\rangle = \langle \bar{\iota}', [\![x = \mathit{Lin}]\!]^{\mathcal{C}} c\rangle \\
&\quad \text{where } \bar{\iota}' = \{p \to q \in \bar{\iota} \mid x \notin \mathit{vars}(p) \cup \mathit{vars}(q)\} \\
&\quad \cup \{\mathit{transform}(p \to q) \mid p \to q \in \bar{\iota}\} \text{ and } \sigma = [x/\mathit{Lin}] \\
&\quad \text{and } \mathit{transform}(p \to q) = \begin{cases} \sigma^{-1}(p) \to \sigma^{-1}(q) & \text{if } \sigma^{-1}(p) \wedge \sigma^{-1}(q) \text{ exists} \\ \mathit{true} \to \mathit{true} & \text{otherwise} \end{cases} \\
&[\![a \bowtie b]\!]^{\mathcal{P}}\langle \bar{\iota}, c\rangle = \langle \bar{\iota} \cup \{\mathit{true} \to a \bowtie b\}, \mathit{fixapply}(\{a \bowtie b\}, \emptyset, c)\rangle \\
&\quad \text{where } \mathit{fixapply}(\bar{p}, \bar{u}, c') = \text{if } \bar{p} \subseteq \bar{u} \text{ then } c' \text{ else} \\
&\quad \text{let } t \in \bar{p} \setminus \bar{u} \text{ and } \bar{n} = \{t\} \cup \mathit{consequences}^{\mathcal{C}}(t, c') \\
&\quad \text{and } \bar{n}' = \{q \mid p \to q \in \bar{\iota} \wedge n \in \bar{n} \wedge n \vdash p\} \cup \{\neg p \mid p \to q \in \bar{\iota} \wedge n \in \bar{n} \wedge n \vdash \neg q\} \\
&\quad \text{in } \mathit{fixapply}(\bar{p} \cup \bar{n}', \bar{u} \cup \{t\}, [\![t]\!]^{\mathcal{C}} c')
\end{aligned}
$$

Figure 4.2: Assignments and branch transfer functions for the predicates domain. The comparison operator $\bowtie$ in a predicate is one of $\leq, \not\leq, <, \not<, =, \neq$.

to encode the value of x in the implications. Specifically, the two outcomes $x = 1 \leftrightarrow a \bowtie b$ and $x = 0 \leftrightarrow a \not\bowtie b$ are stored using four implications.

The transfer function $[\![x = \mathit{NonLin}]\!]^{\mathcal{P}}$ for non-linear assignment removes all implications in the predicate domain containing x. An assignment $[\![x = \mathit{Lin}]\!]^{\mathcal{P}}$ of a linear expression to x tries to transform implications containing x if Lin contains x, e.g. `x=x+1`. For example, consider the predicates state $\bar{\iota} = \{f = 0 \to x \leq 5, x \not\leq 10 \to y = 10\}$ and the assignment `x=x+1` mentioned above. Given the substitution $\sigma = [x/x + 1]$ that describes the change of the state space, we compute $\sigma^{-1} = [x/x - 1]$ that describes how predicates can be transformed so that they are valid in the new state. In the example, applying σ^{-1} to the implications yields $\bar{\iota}' = \{f = 0 \to x \leq 6, x \not\leq 11 \to y = 10\}$. In all three assignments, more predicates can be retained by testing if they are still valid after the assignment.

We now consider the transfer function for an assumption $[\![a \bowtie b]\!]^{\mathcal{P}}$. The predicate domain tracks the information of the assumption as a new symbolic fact using an implication $\{\mathit{true} \to a \bowtie b\}$. Furthermore, the information from the test $a \bowtie b$ is used by the domain to gather further facts about the state. The process of applying these facts to the child domain is called reduction [CC79a]. The reduction is performed as a fixpoint computation and can be seen as an instance of Granger's framework for reduction by local iteration [Gra92]. Specifically, the function *fixapply* gathers a set of deduced predicates $\bar{p}$ and a set of predicates $\bar{u}$ that have already been applied on the child. In each iteration a predicate $t \in \bar{p} \setminus \bar{u}$ is applied to the child state c', yielding $[\![t]\!]^{\mathcal{C}} c'$. Furthermore, a set of new predicates that are implied by t are computed in two steps. First, t is combined with a set $\bar{n}$ of semantic consequences which is computed by $\mathit{consequences}^{\mathcal{C}}$ as detailed below. Second, a set of

syntactically implied predicates $\bar{n}'$ is computed from $\bar{n}$ by inspecting the implications in the predicate domain. We use modus ponens resolution to deduce q from an implication $p \to q \in \bar{\iota}$ where $t \vdash p$ and deduce $\neg p$ if $t \vdash \neg q$. Here, the syntactic entailment $\vdash$ is defined as follows:

Definition 4.1 (Syntactic Predicate Entailment $\vdash$) A predicate q is entailed by another predicate p, written as $p \vdash q$, if $p \equiv q$ or if p describes a weaker condition that subsumes the condition of q. We use the following syntactic entailment rules:
(x, y are variables and c is a constant; analogous definitions exist for the negations of the comparison operators $\not\leq, \not<$)

$$
\begin{array}{lll}
p \vdash x \neq c & \text{if} & p \in \{x = c' \mid c' \neq c\} \cup \{x \leq c' \mid c' < c\} \cup \{x < c' \mid c' \leq c\} \\
p \vdash x \leq c & \text{if} & p \in \{x = c' \mid c' \leq c\} \cup \{x \leq c' \mid c' \leq c\} \cup \{x < c' \mid c' - 1 \leq c\} \\
p \vdash x < c & \text{if} & p \in \{x = c' \mid c' < c\} \cup \{x \leq c' \mid c' < c\} \cup \{x < c' \mid c' \leq c\} \\
p \vdash x = y & \text{if} & p \in \{x = y\} \\
p \vdash x \neq y & \text{if} & p \in \{x \neq y\} \cup \{x < y\} \\
p \vdash x \leq y & \text{if} & p \in \{x \leq y\} \cup \{x = y\}
\end{array}
$$

The set of syntactically implied predicates $\bar{n}'$ is added to $\bar{p}$ and, hence, eventually applied to the child state. Since at most two predicates for each implication in $\bar{\iota}$ can be added to $\bar{p}$, this iterative reduction terminates.

Although not strictly necessary, the $consequences^{\mathcal{C}}$ function allows information to flow from the child domain to the predicate domain. The function synthesizes new predicates that become valid after applying the test t. It is different for each child domain. An implementation for the interval domain $\mathcal{I}$ is as follows:

$$
consequences^{\mathcal{I}}(t, c) = \text{ let } c' = [\![t]\!]^{\mathcal{I}} c \text{ in } \{x = l \mid c(x) \neq c'(x) \wedge c'(x) = [l, l]\}
$$

Here, $c(x)$ is the interval of the variable x in the state c. The insight in this definition is that the only additional information inferable by the interval domain is that a variable x may have become constant due to a test such as $x \leq c$. Returning these equality predicates may allow additional deductions in the predicate domain. Note that other child domains may deduce different facts.

4.3.2 Example for the Reduction after Executing Assumptions

We illustrate the reduction when applying an assumption $[\![a \bowtie b]\!]^{\mathcal{P}}$ using an example. Consider applying the test $f < 1$ to the state $s = \langle \bar{\iota}, c \rangle$ that consists of the predicates $\bar{\iota} \in \mathcal{P}$ and the intervals $c \in \mathcal{I}$ as child domain. Let $\bar{\iota} = \{f = 0 \to x \leq 0\}$ and $c = \{f \in [0, 1], x \in [-1, 1]\}$. The first step in the transfer function is to infer the consequences of the test: $\bar{n} = consequences^{\mathcal{I}}(f < 1, c)$. As the child state becomes $c' = \{f \in [0, 0], x \in [-1, 1]\}$, the consequences are $\bar{n} = \{f = 0\}$. The synthesized predicate in $\bar{n}$ syntactically entails the left-hand side of the implication $f = 0 \to x \leq 0$ that is tracked in the predicate domain. Thus, *fixapply* calls itself recursively with the new predicate $x \leq 0$ which results in a call to $consequences^{\mathcal{I}}(x \leq 0, c') = \emptyset$. Now, the set of implied predicates $\bar{n}'$ is empty and a

fixpoint is reached since $\bar{p} = \bar{u} = \{f < 1, x \leq 0\}$. Thus, the result of the transfer function is $[\![f < 1]\!]^{\mathcal{P}} s = \langle \{\mathit{true} \to f < 1, f = 0 \to x \leq 0\}, \{f \in [0,0], x \in [-1,0]\} \rangle$.
This recursive reduction mechanism implements all required reductions between the predicate and the child domain. The next section illustrates how this reduction mechanism is used to preempt the loss of precision due to convexity.

4.3.3 Application to Non-Convex Spaces

Reconsider the example in Fig. 4.1 where a division by zero is prevented by a guard. The problem here is that the state space for d is non-convex and cannot be expressed with the intervals domain $\mathcal{I}$. However, using the predicate domain $\mathcal{P}$ we are able to prove the invariant at program point 4 even though the interval value for d at that point is $d \in [-2, 2]$. We illustrate an analysis of the program for an initial state where the interval domain tracks d with the value $d \in [-2, 2]$. By executing line 1, the four implications for the assignment of a comparison are added to the predicate domain, yielding the state $\bar{\iota} = \{f = 1 \to d \neq 0, f = 0 \to d = 0, d \neq 0 \to f = 1, d = 0 \to f = 0\}$. On entering the **then**-branch, the test $f = 1$ in line 3 restricts the variable f in the interval domain to $f \in [1, 1]$. The predicate domain uses the first implication to deduce $d \neq 0$, which is also applied to the child domain. However, the child domain $\mathcal{I}$ is not able to express the disjunction $d \in [-2, -1] \vee [1, 2]$ thus the state after applying $d \neq 0$ remains $d \in [-2, 2]$. The assertion in line 4 translates to an edge to the dedicated error node that is labeled with the test $d = 0$. Hence, the assertion fails if $d = 0$ is satisfiable. The predicate domain observes that the right-hand side $d \neq 0$ of the implication $f = 1 \to d \neq 0$ is *false* and thus adds the negated left-hand side $f \neq 1$ to $\bar{n}'$. Once the predicate domain applies $f \neq 1$ to the child state $c = \{f \in [1, 1], d \in [-2, 2]\}$, the result is $\bot$, the unreachable state. Thus, the error node is not reachable in the program and the assertion is verified even though the convex numeric domain is not precise enough to express $d \neq 0$. The reduction mechanism is able to exploit the information in the implications for verifying assertions without requiring more complex (i.e. non-convex) numeric domains.

4.3.4 Symbolic Reasoning for Unbounded Spaces

If the numeric domain does not track any useful information about variable bounds the predicate domain can still prove assertions using only the syntactic predicate entailment. Consider the example from [LL11] shown in Fig. 4.3. The analysis commences in line 1 with a state where the interval domain tracks the most imprecise information for each of the variables: $c = \{x \in [-\infty, +\infty], y \in [-\infty, +\infty], z \in [-\infty, +\infty]\}$. Consequently, the tests in lines 3, 4, 7 and 8 do not restrict the interval bounds of the variables and we cannot prove the assertions in lines 11 and 12. More complex numeric domains that are capable of symbolic reasoning, e.g. polyhedra [CH78], are able to prove the assertions. Our predicate domain $\mathcal{P}$ performs limited symbolic reasoning due to its syntactic entailment mechanism. In the above example using the predicates domain we collect the tests from the assumptions as facts and track the state $\bar{\iota}_1 = \{\mathit{true} \to x = y, \mathit{true} \to y \leq z\}$ for the **then**-branch and $\bar{\iota}_2 = \{\mathit{true} \to x \leq y, \mathit{true} \to y = z\}$ for the **else**-branch. The join

```
1  ...
2  if (...) {
3    assume (x == y);
4    assume (y <= z);
5    ...
6  } else {
7    assume (x <= y);
8    assume (y == z);
9    ...
10 }
11 assert (x <= y);
12 assert (y <= z);
```

Figure 4.3: Using syntactic entailment to prove inequalities.

$\bar{\iota}_1 \sqcup_{\mathcal{P}} \bar{\iota}_2 = \{\mathit{true} \to x \leq y, \mathit{true} \to y \leq z\}$ in line 11 keeps the predicates that are syntactically entailed in both states (we will describe the join in more detail in the next section). To prove the assertions correct the predicate domain uses the syntactic entailment in $\mathit{fixapply}$ to entail the negated assertion conditions: $x \not\leq y$ and $y \not\leq z$. This in turn entails the negated premise $\neg \mathit{true}$ resulting in $\bot$ and thus proving the assertion.

In general, observing predicates from assignments and assumptions is only a syntactic technique that may fail for more complex disjunctive invariants. The next section therefore illustrates how the reduction mechanism implemented by $\mathit{fixapply}$ naturally combines with a more sophisticated way of inferring new implications.

4.4 Lattice Operations and Predicate Synthesis

$$
\begin{aligned}
&\langle \bar{\iota}_1, c_1 \rangle \sqsubseteq_{\mathcal{P}} \langle \bar{\iota}_2, c_2 \rangle = c_1 \sqsubseteq_{\mathcal{C}} c_2 \wedge \mathit{entailed}(\bar{\iota}_2, \bar{\iota}_1, c_1) = \bar{\iota}_2 \\
&\quad \text{where } \mathit{entailed}(\bar{\iota}', \bar{\iota}, c) = \{p' \to q' \in \bar{\iota}' \mid (\exists p \to q \in \bar{\iota}. p' \vdash p \wedge q \vdash q') \vee ([\![p']\!]^{\mathcal{C}} c \vDash q')\} \\
&\langle \bar{\iota}_1, c_1 \rangle \sqcup_{\mathcal{P}} \langle \bar{\iota}_2, c_2 \rangle = \langle \mathit{join}(\bar{\iota}_1, \bar{\iota}_2) \cup \mathit{synth}^{\mathcal{C}}(c_1, c_2), c_1 \sqcup_{\mathcal{C}} c_2 \rangle \\
&\quad \text{where } \mathit{join}(\bar{\iota}_1, \bar{\iota}_2) = \mathit{entailed}(\bar{\iota}_1, \bar{\iota}_2, c_2) \cup \mathit{entailed}(\bar{\iota}_2, \bar{\iota}_1, c_1) \\
&\langle \bar{\iota}_1, c_1 \rangle \nabla^l_{\mathcal{P}} \langle \bar{\iota}_2, c_2 \rangle = \langle \mathit{join}(\bar{\iota}_1, \bar{\iota}_2) \cup \mathit{synth}^{\mathcal{C}}(c_1, c_2), c_1 \nabla^l_{\mathcal{C}} c_2 \rangle
\end{aligned}
$$

Figure 4.4: Lattice operations for the predicate domain.

We present entailment test, join and widening operations of the predicate domain. Moreover, we introduce a novel synth function that synthesizes new implications between predicates that counteract the loss of precision in numeric domains.

4.4.1 Lattice Operations

We commence by detailing the entailment test $\langle \bar{\iota}_1, c_1 \rangle \sqsubseteq_{\mathcal{P}} \langle \bar{\iota}_2, c_2 \rangle$ in Fig. 4.4. It performs the entailment test $c_1 \sqsubseteq_{\mathcal{C}} c_2$ on the child domain and tests if all the implications in the right argument $\bar{\iota}_2$ are entailed by the left argument by calling the function $\mathit{entailed}(\bar{\iota}', \bar{\iota}, c)$. The latter function returns an implication $p' \to q' \in \bar{\iota}'$ if it is either syntactically entailed in $\bar{\iota}$ or semantically entailed in the state c. Semantic entailment $\vDash$ is defined as follows:

Definition 4.2 (Semantic Predicate Entailment $\vDash$) A predicate q is entailed in a state c, written $c \vDash q$, if testing $\neg q$ in c yields an empty state, i.e., $[\![\neg q]\!]^{\mathcal{C}} c = \bot$.

By this definition, the test $[\![p']\!]^{\mathcal{C}} c \vDash q'$ in the body of $\mathit{entailed}$ reduces to checking whether $[\![\neg q']\!]^{\mathcal{C}}([\![p']\!]^{\mathcal{C}} c) = \bot$. Thus, if the predicate p' on the left-hand side of the implication $p' \to q'$ is false in c then $[\![\neg q']\!]^{\mathcal{C}} \bot = \bot$ follows and the implication is entailed in c (known in logic as "ex falso quodlibet"). The two tests $[\![\cdot]\!]^{\mathcal{C}}$ on the child domain c can be avoided if the implication is syntactically entailed by an implication in $\bar{\iota}$. Here, the implication $p \to q \in \bar{\iota}$ entails $p' \to q'$ if the premise p' is stronger and the conclusion q' is weaker which is expressed by $p' \vdash p \wedge q \vdash q'$. Note that neither the syntactic nor the semantic entailment test subsumes the other as both approximate the test differently.

The join $\langle \bar{\iota}_1, c_1 \rangle \sqcup_{\mathcal{P}} \langle \bar{\iota}_2, c_2 \rangle$ independently computes a join on the predicate domain and on the child domain. In oder to join the implication sets $\bar{\iota}_1$ and $\bar{\iota}_2$, we define a function join that keeps all implications that hold in the respective other state using the $\mathit{entailed}$ function described above. Note that the semantic entailment test in $\mathit{entailed}$ is particularly important for the join as one of the predicate domain states may be empty so that the syntactic entailment would discard all implications. The semantic join is able to retain newly inferred predicates in, for example, loop bodies as illustrated later.

In addition to the predicates returned by the join function, new implications are synthesized from the child domain states using the $\mathit{synth}^{\mathcal{C}}$ function. The idea is to synthesize implications that characterize the approximation that occurred as part of the $\sqcup_{\mathcal{C}}$ operation. Which synthesized implications are generated depends on the numeric domain. If the predicate language is sufficiently expressive, a domain could potentially characterize all precision losses that occur during a join. The following $\mathit{synth}^{\mathcal{I}}$ function for the interval domain is an example that generates implications for all changing bounds. Moreover, by relating changes of interval bounds between different variables, it generates relational information that cannot be expressed within the interval domain itself. It is defined as follows:

$$
\begin{aligned}
&\mathit{synth}^{\mathcal{I}}(c_1, c_2) = \text{let } c = c_1 \sqcup_{\mathcal{I}} c_2 \\
&\quad \text{and } \bar{m} = \{x \in \mathit{vars}(c_1) \cap \mathit{vars}(c_2) \mid c_1(x) \neq c_2(x)\} \text{ and } i \in \{1, 2\} \\
&\quad \text{and } \bar{u_i} = \{u_{xi} \mid x \in \bar{m} \wedge c_i(x) \in [l_{xi}, u_{xi}] \wedge c(x) \in [l_x, u_x] \wedge u_{xi} < u_x\} \\
&\quad \text{and } \bar{l_i} = \{l_{xi} \mid x \in \bar{m} \wedge c_i(x) \in [l_{xi}, u_{xi}] \wedge c(x) \in [l_x, u_x] \wedge l_x < l_{xi}\} \\
&\quad \text{in } \{u_{x1} < x \to l_{y2} \leq y, u_{y1} < y \to l_{x2} \leq x \mid x, y \in \bar{m} \wedge u_{xi}, u_{yi} \in \bar{u_i} \wedge l_{xi}, l_{yi} \in \bar{l_i}\}
\end{aligned}
$$

Let $\mathit{vars}(c)$ return all the variables $\bar{x} \subseteq \mathcal{X}_V$ tracked in the state c and let $c(x)$ denote the interval of the variable x. The set of variables $\bar{m}$ that are not equal in both states are

those whose joined value is an approximation of the input intervals. For these variables we compute a set of changing lower and upper bounds $\bar{l}_i$ and $\bar{u}_i$ whose indices indicate the variable and origin of the bound. For example, when joining $c_1(x) \in [0, 5]$ with $c_2(x) \in [10, 15]$, resulting in $c(x) \in [0, 15]$, the upper bound $u_{x1} = 5$ of $c_1(x)$ and the lower bound $l_{x2} = 10$ of $c_2(x)$ are lost whereas the other bounds are retained in $c(x)$. These changing bounds are used for generating implications. Specifically, each implication correlates a lost upper bound u_{xi} from c_i with a lost lower bound $l_{y(2-i)}$ from c_{2-i} where $i = 1, 2$. For the example above $x = y$, thus the only generated implication is $u_{x1} < x \to l_{x2} \leq x$, that is, $5 < x \to 10 \leq x$. The implication allows that a test such as $7 < x$ is refined to $10 \leq x$, thereby recovering the precision loss in the join that is due to the convexity of the interval domain. In general, the bounds of several variables can be related, thereby even generating relational information.

One drawback of the definition above is that implications are added for each pair of variables from $\bar{m}$, thus, the returned set of implications is quadratic in $|\bar{m}|$. This quadratic growth can be avoided by not generating a redundant implication $a \to c$ if both $a \to b$ and $b \to c$ are already present. Specifically, by sorting $\bar{m}$ using some total ordering, we only emit implications over variables that are adjacent in this ordering, as well as an implication relating the largest variable with the smallest. As the predicate domain performs a transitive closure on application of a test predicate (through *fixapply*), adding only implications between adjacent variables is sufficient to recover all information expressed in a chain of implications. Using this optimization, we are able to reduce the number of synthesized implications to be linear in the number of changed variables $|\bar{m}|$.

Before we consider further examples, we consider the widening operation, defined in Fig. 4.4. The definition is analogous to the join operation but applies widening on the child states c_1, c_2. One caveat of this definition is that termination is not guaranteed. Consider an implication $p' \to q'$ at a loop head and assume that a conditional in the loop refines the child state by using the $[\![a \bowtie b]\!]^{\mathcal{P}}$ transformer in Fig. 4.2 which, in turn, may use the information in $p' \to q'$. Suppose that joining the two branches of the conditional creates a new implication $p \to q$ by means of the $synth^{\mathcal{C}}$ function that is syntactically weaker than $p' \to q'$. If $[\![p']\!]^{\mathcal{C}} c_1 \not\models q'$ (the previous implication cannot be shown to hold in the new state) then the loop is not stable. If furthermore $[\![p]\!]^{\mathcal{C}} c_2 \models q$ (the new implication holds in the previous state), the loop is analyzed with the new implication. Thus, one implication may be replaced by another one, possibly indefinitely so. In order to ensure termination, standard widening techniques can be used, such as eventually disallowing new implications from loop bodies (see Sect. 3.4). This can be implemented by using the definition $\langle \bar{\iota}_1, c_1 \rangle \nabla^l_{\mathcal{P}} \langle \bar{\iota}_2, c_2 \rangle = \langle entailed(\bar{\iota}_1, \bar{\iota}_2, c_2), c_1 \nabla^l_{\mathcal{C}} c_2 \rangle$ after k iterations. So far, we were unable to find examples that exhibit this non-terminating behavior.

4.4.2 Application to Non-Convex Spaces

As stated in Sect. 4.3.3 the analysis of the introductory example relies on the predicate $d \neq 0$ that is observed syntactically in the program. We use this observed predicate to later prove that the convex interval approximation $d \in [-2, 2]$ does not contain the value 0. A

```
1 if (x < 0) {
2   sgn = -1;
3 } else {
4   sgn = 1;
5 }
6 assert (sgn != 0);
```

Figure 4.5: Computing the sign of a variable.

similar example from [MR05] is shown in Fig. 4.5. Here, the condition of the assertion does not appear syntactically in the code. Due to the join at the end of the **if**-branch we face again a precision loss due to the convex approximation. Hence, using intervals alone the assertion cannot be proved as $sgn \in [-1, 1]$ in line 6. Although, syntactically we do not have a predicate on variable sgn in the program, the $synth^{\mathcal{I}}$ function synthesizes the implication $-1 < sgn \to 1 \leq sgn$ during the interval join in line 6. This is sufficient to prove the assertion as now testing $sgn = 0$ in line 6 yields $\bot$. The test reduces the tracked interval to $sgn \in [0, 0]$ after which it syntactically entails the premise of $-1 < sgn \to 1 \leq sgn$. In *fixapply*, the consequence of the implication is applied to the reduced state yielding $\bot$. Hence, variable sgn is not equal to 0 at this program point.

In this example, synthesizing implications on a single variable allowed us to represent non-convex states. We will show next that the $synth^{\mathcal{I}}$ function is even more useful when relating the precision loss of different variables.

4.4.3 Recovering Precision using Relational Information

One strength of our $synth^{\mathcal{I}}$ function is that it creates relational information, that is, it generates implications between different variables. This relational information enables *fixapply* to deduce, from a test of one variable, more precise ranges for other variables. In particular, a test t that separates two states, i.e. $[\![t]\!]^{\mathcal{I}} c_1 = c_1$ and $[\![t]\!]^{\mathcal{I}} c_2 = \bot$ is enriched by the relational implications so that all losses due to convexity are recovered, that is, $[\![t]\!]^{\mathcal{P}}(\langle \bar{\iota}_1, c_1\rangle \sqcup_{\mathcal{P}} \langle \bar{\iota}_2, c_2\rangle) = \langle \bar{\iota}'_1, c_1\rangle$.

	c_1	c_2	$synth^{\mathcal{I}}(c_1, c_2)$	$c_1 \sqcup_{\mathcal{I}} c_2$
$x \in$	$[0, \mathbf{5}]$	$[\mathbf{10}, 15]$	$\{5 < x \to 2 \leq y,$	$[0, 15]$
$y \in$	$[-5, \mathbf{-1}]$	$[\mathbf{2}, 3]$	$-1 < y \to 10 \leq x\}$	$[-5, 3]$

Figure 4.6: The join of two states in the intervals domain $\mathcal{I}$ and the synthesized implications correlating the bounds lost due to the convex approximation.

We illustrate this ability using two states $s_1 = \langle \emptyset, \{x \in [0, 5], y \in [-5, -1]\}\rangle$ and $s_2 = \langle \emptyset, \{x \in [10, 15], y \in [2, 3]\}\rangle$. The joined state $s = s_1 \sqcup_{\mathcal{P}} s_2$ is given by $s = \langle \{5 < x \to 2 \leq y, -1 < y \to 10 \leq x\}, \{x \in [0, 15], y \in [-5, 3]\}\rangle$. This operation is illustrated in Fig. 4.6 where the bounds in bold are those that are lost and the arrows

indicate which bounds are related by the generated implications. We now show how applying the test $0 < y$ on s recovers the numeric state in s_2 and, analogously, that applying $y \leq 0$ recovers the numeric state of s_1. Specifically, when applying the test $0 < y$ on state s, the left-hand side of the implication $-1 < y \rightarrow 10 \leq x$ is syntactically entailed, so that $10 \leq x$ is also applied to the child state, yielding the precise value $[10, 15]$ for x. The predicate $10 \leq x$ syntactically entails the other implication $5 < x \rightarrow 2 \leq y$. Thus, the predicate $2 \leq y$ is applied to the child state, yielding the precise value $[2, 3]$ for y. After that no new predicates are entailed and the recursive predicate application in the function *fixapply* stops with the state $s_2' = \langle\{5 < x \rightarrow 2 \leq y, -1 < y \rightarrow 10 \leq x\}, \{x \in [10, 15], y \in [2, 3]\}\rangle$. Observe that all losses due to the join have been recovered so that the values tracked in s_2' are identical to that of s_2. Analogously, we get a state s_1' in which the interval for x is $[0, 5]$ and for y is $[-5, -1]$ after applying the opposing condition $y \leq 0$.

In summary, the predicate domain improves the precision of a child domain by tracking precision losses that are reported by the child. In particular, the domain-specific $synth^{\mathcal{C}}$ function can generate implications between predicates that cannot be expressed in the domain itself. This allows the predicate domain to maintain enough disjunctive information to recover the state before the join whenever a test is able to separate the two states. Note though that there exist cases when this is not completely possible, namely when the value of x in one state overlaps the value in the other state. Consider $c_1(x) \in [0, 4]$ and $c_2(x) \in [2, 8]$. A test $x < 3$ does not separate the two states. However, any test outside the overlapping range $[2, 4]$ is able to separate the two states which, in turn, leads to the refinement of other bounds.

4.4.4 Application to Path-Sensitive Invariants

```
1  FILE *out;
2  out->is_open = 1;
3  assert(out->is_open == 1);
4  out->is_open = 0;
5  ...
6  if (flag)
7    out->is_open = 1;
8  ...
9  if (flag)
10   assert(out->is_open == 1);
```

Figure 4.7: A locking scheme example: accessing a file only if it was already opened.

This section illustrates how our domain can verify an example taken from [FJM05]. The challenge of analyzing the code in Fig. 4.7 is that the join of different paths loses precision and the invariant that a file is only accessed if it was opened before cannot be proved. For the sake of presentation, we use *open* to denote the value of `out->is_open`. Note that the

assertion in line 3 can be proved by using the interval domain alone, as $open$ is $[1,1]$ due to line 2. However, the assertion in line 10 cannot be proved by using intervals alone: observe that $open$ is set to $[0,0]$ in line 4 and that the join of this value with the value $[1,1]$ from line 7 yields the convex approximation of $[0,1]$ in line 10 of the assertion. As a consequence, the assertion cannot be proved since the edge to the error state with assumption $open = 0$ is satisfiable. Now consider analyzing the example using the predicate domain with the interval domain as child. Then the join of the **then**-branch in line 7 and the state before line 6 creates an implication $0 < flag \rightarrow 1 \leq open$. When applying the branch condition $flag = 1$ of line 9, the implied predicate $1 \leq open$ is used to reduce the state, yielding $open \in [1,1]$ in the interval domain. Thus, the assertion can be proved since the edge to the error state with assumption $open = 0$ is unreachable. The example illustrates how numeric domains may lose precision when joining paths [MR05] and, thus, may fail to express a path-sensitive invariant which is crucial to prove assertions in the branch of a conditional.

Fischer et al. [FJM05] prove the assertion in line 10 by not joining the states after the conditional in line 6, thus keeping the states where $open = 0$ and $open = 1$ separate. They associate a predicate with a numeric state and join numeric states only if they are associated with the same predicate. Thus, their abstract state before the conditional in line 9 is $\{\langle flag = 0, open \in [0,0]\rangle, \langle flag = 1, open \in [1,1]\rangle\}$ which reduces to $\{\langle flag = 1, open \in [1,1]\rangle\}$ inside the conditional. Although their approach is able to prove the assertion, it is more costly as it tracks several numeric states. Even if sharing can reduce the resource overhead of tracking multiple states [GC10b], the cost of tracking several states is generally higher [MR05]. Our approach retains the conciseness of a single convex numeric state and merely adds the implications necessary to express certain disjunctive information. In particular, we only infer disjunctive information for variables that actually differ in the join of two numeric states rather than duplicating the information on all variables.

4.4.5 Application to Separation of Loop Iterations

```
1  p = &some_var;
2  n = 5;
3  while (n >= 0) {
4    assert(p != 0);
5    ... = *p;
6    ...
7    if (n == 0)
8      p = 0;
9    n--;
10 }
```

Figure 4.8: A challenging example: freeing a pointer in the last loop iteration.

A particularly challenging example from the literature [HHP13] requires that variable values of certain loop iterations are distinguished. The example in Fig. 4.8 is prototypical

for a loop that frees a memory region in its last iteration. The assertion in line 4 expresses that the memory region pointed-to by p has not yet been deallocated. In order to prove this assertion, an analysis needs to separate the value of the pointer p in the last loop iteration from its value in all previous iterations. In particular, the example is difficult to prove using convex numeric domains due to a precision loss that occurs when joining the point $\langle p, n\rangle = \langle 0, -1\rangle$ at line 10 of the last loop iteration with the earlier states where $p \neq 0$ and $n \geq 0$.

step	line		intervals		implications
			p	n	
1	2		$[99, 99]$		
2	3		$[99, 99]$	$[5, 5]$	
3	4		$[99, 99]$	$[5, 5]$	
...			...		...
6	7		$[99, 99]$	$[5, 5]$	
7	9		$[99, 99]$	$[5, 5]$	
8	10		$[99, 99]$	$[4, 4]$	
9	3	$\sqcup$	$[99, 99]$	$[4, 5]$	
9'	3'	∇	$[99, 99]$	$[-1, 5]$	
10	4		$[99, 99]$	$[\mathbf{0}, \mathbf{5}]$	
...			...		...
14	**8**		$[99, 99]$	$[0, 0]$	
15	9	$\sqcup$	$[\mathbf{0}, \mathbf{99}]$	$[0, 5]$	$\{\mathbf{0 < n \rightarrow 99 \leq p, 0 < p \rightarrow 0 \leq n}\}$
16	10		$[0, 99]$	$[-1, 4]$	$\{-1 < n \rightarrow 99 \leq p, 0 < p \rightarrow -1 \leq n\}$
17	3	$\sqcup$	$[0, 99]$	$[-1, 5]$	$\{-1 < n \rightarrow 99 \leq p, 0 < p \rightarrow -1 \leq n\}$
18	4		$[\mathbf{99}, \mathbf{99}]$	$[0, 5]$	$\{\mathbf{-1 < n \rightarrow 99 \leq p}, 0 < p \rightarrow -1 \leq n\}$
...			...		...
25	3	$\sqsubseteq$	$[0, 99]$	$[-1, 5]$	$\{-1 < n \rightarrow 99 \leq p, 0 < p \rightarrow -1 \leq n\}$

Figure 4.9: States during the analysis of the loop example in Fig. 4.8.

However, using the simple interval numeric domain and our predicate domain, the example is proved using the fixpoint computation detailed in Fig. 4.9. In step 1 of the table, p is initialized to a non-zero address of a variable, which we illustrate by using the value 99. After initializing the loop counter n in step 2, the loop is entered as the loop condition `n >= 0` is satisfied. In step 6, it is determined that the **then**-branch in line 8 is not reachable. After decrementing n, the state is propagated to the loop head via the back-edge in step 9. At this point, widening is applied. Our thresholds widening (see Sect. 3.4) ensures that the interval $[-1, 5]$ is tried for n, rather than widening n immediately to $[-\infty, 5]$. By applying the loop condition `n >= 0`, a new state for the loop body is obtained in step 10. In step 14, it is observed that the **then**-branch in line 8 is reachable. In the next step in line 9 the states of both branches are joined and the interval domain approximates p with $[0, 99]$. In the same step, the implications $0 < n \rightarrow 99 \leq p, 0 < p \rightarrow 0 \leq n$ are synthesized. In step 16 these predicates are transformed using $\sigma^{-1} = [n/n + 1]$. This state is joined with the previous state at the loop head at step 17. Our widening heuristic from Sect. 3.3 suppresses widening since a new branch with a constant assignment has become live. Since the resulting numeric state has changed due to the new value of p, the fixpoint computation continues. Note that during the join in step 17, both implications

$-1 < n \to 99 \leq p, 0 < p \to -1 \leq n$ are semantically entailed in the current state at the loop head (as computed in step 9') and therefore kept in the joined state. Evaluating the loop condition in step 18 enforces that $n \geq 0$, that is, $0 \leq n$. The latter predicate syntactically entails the predicate $-1 < n$. Thus, the *fixapply* function deduces that $99 \leq p$ holds, yielding $p \in [99, 99]$. The assertion holds since intersecting the state at step 18 with $p = 0$ yields $\bot$. Thus, at line 4, p cannot be 0 and the assertion holds. Continuing the analysis of the loop observes a fixpoint in step 25. Note that the assertion can also be shown when using standard widening that sets n to $[-\infty, 0]$ in step 8' or not delaying widening in step 17 resulting in $p \in [-\infty, 99]$. However, for the sake of presentation, we illustrated the example with the more precise states.

4.5 Experimental Results

benchmark suite	programs	lines	lines avg.	time avg.	time avg. ($\mathcal{P}$)	time avg. ($\mathcal{PH}$)
literature	9	9–17	14	38 ms	99 ms	381 ms
test	8	66–274	115	393 ms	1658 ms	-

Figure 4.10: Evaluation of our implementation. Due to technical reasons, the "test" benchmark suite could not be analyzed using the disjunctive domain ($\mathcal{PH}$).

The Predicate abstract domain was inspired by our weaker *flags* domain (see Sect. 2.5.3) that tracked bi-implications of the form $f \leftrightarrow x \leq c$. This domain is useful in the analysis of machine code where conditional branches are encoded using two separate instructions. The first instruction is a comparison that stores the result of $x \leq c$ in a processor flag f. The second instruction is a branch instruction that determined the jump target based on f. By tracking an association between the comparison result f and the predicate $x \leq 0$, the edge of the jump with the assumption $f = 1$ can be made more precise by also assuming $x \leq 0$ and analogously for $f = 0$. However, the use of simple bi-implications only states additional invariants rather than predicates that hold conditionally. Hence, disjunctive information cannot be described by using only bi-implications.

We evaluated our combined predicate/numeric domain on several examples in the literature, including the ones presented in this paper, shown as "literature" in Fig. 4.10. We also evaluated larger examples shown as "test". All examples from the literature required the predicate domain to verify except for the example in Fig. 4.1 that our weaker predicate domain [SMS11] already handles. The times are shown when running without and with the predicate domain "($\mathcal{P}$)". The last column shows the running time with a disjunctive domain "($\mathcal{PH}$)" that tracks different numeric states depending on the index ranges of a loop (see Sect. 3.5). Due to this, only one example in the "literature" benchmark suite could possibly profit. A precision comparison of our disjunctive and the predicate domain can therefore not be conclusive for the disjunctive domains in the literature [GC10b; MR05].

4.6 Related Work

The idea of abstracting a system relative to a set of predicates was first applied by Graf and Saïdi to state graphs created during model checking [GS97]. This approach has later been generalized to software model checking by Ball et al. [Bal+01]. Here, an abstraction tool C2BP translates a C program to a program over Boolean variables. The value of a Boolean variable is true if the corresponding predicate holds in the input C program. The universe of possible predicates is very large as the semantics of each assignment and test is expressed by predicates. For scalability, C2BP abstracts the input C program only with respect to a few predicates.
The idea of counter-example driven refinement is to increase this set of predicates by deducing which additional predicates are needed to discharge a verification condition. This deduction is performed on a path through the program on which the current Boolean abstraction is insufficient to prove a verification condition. There are two ways in which this refinement may fail: Firstly, the translation of C statements and tests into predicates may be inaccurate or the logic of the predicates may be insufficient to represent the C semantics precisely. Secondly, a set of predicates that suffices to discharge the verification condition on the chosen path may be insufficient when considering the whole program.

An abstract interpretation over domains that lose precision due to convexity is naturally improved by making the abstraction closed for disjunction, that is, disjunctions can be expressed without approximation. This approach is commonly known as disjunctive completion [CC79a]. In practice, the disjunctions are qualified by a set of predicates and are stored in a binary decision-diagram (BDD) where decision nodes are labeled with predicates and the leaves are convex numeric abstract domains [GC10b; MR05]. The challenge in implementing these domains is that the evaluation of transfer functions in one leaf may lead to a result that has to be propagated to many other leaves. A particular challenge is the widening operator and the reduction between predicates and states [GC10a; MSS13]. One drawback of using a BDD as state is that computing a fixpoint of a loop will perform all operations on each leaf of the BDD, even those that are stable within, say, the current loop. This can be avoided by lifting the fixpoint computation from tracking a map $P \to S$ to $P \times C \to S$ where P are program points, S are states and C is a context. By using the predicates on a path in the decision diagram as context, the whole decision tree can be encoded by using one context per path. The advantage of this encoding is that stable leaves in the original decision tree are no longer propagated since they are each checked for stability by the fixpoint engine [San+06]. Using predicates as context can be seen as an elegant way of duplicating the CFG which is a technique sometimes used to improve widening [GR07].

Beyer et al. combine abstract domains with predicates [BHT08]. Their framework associates a precision level Π with each domain that can be adjusted based on observed values in the program. A value-set analysis, for instance, may specify that only variables with less than five values are tracked while a predicate domain will store the set of inferred predicates in Π. They propose to change this precision level during the analysis, so that a precision loss in one domain can be met with a precision increase in another. They instantiate their

framework by an analysis that switches from tracking value sets to tracking predicates once the former becomes too expensive. Their states are tuples of the precision levels and the domain states so that a different domain state is tracked for each precision level. Their approach thereby resembles the disjunctive completion approaches discussed earlier. Interestingly, they propose the use of a function *abstract* to synthesize predicates from an abstract state. However, in their implementation it only returns predicates occurring in the current program.

Laviron and Logozzo [LL11] use a similar approach to refine numeric domains. Their refinement predicates are called "hints" and can either be supplied by the user or extracted from the program, e.g. from assumptions. Using templates [SSM05] they are able to also synthesize hints during the join of numeric domains. However, their templates do not intend to recover the precision loss of convex numeric domains but are used to infer new predicates thereby enriching the numeric state with tighter invariants. This is a similar approach like in their earlier work on the pentagons domain [LF08] but extended to sub-polyhedra. As their hints are not implications they cannot express non-convex spaces and thus cannot prove correct the example in Fig. 4.5 but require a backwards analysis to refine the over-approximation [LF08] using disjuncts.

Further afield are techniques to refine abstract interpretations based on counter examples [LL05; GR06b]. The idea here is to re-run the abstract interpretation once a verification condition cannot be discharged. An improved precision of the abstract interpreter is obtained by improving the widening or the abstract state based on the path of the counter example. Our work can be seen as dual to counterexample-driven refinement as we employ predicates to avoid a precision loss rather than to refine a state that is too coarse. An approach that uses counterexample-driven refinement and which is seemingly close to ours is that of Fischer et al. [FJM05] who propose a domain containing a map from a predicate to a numeric abstract domain. Like our setup, their construction is a reduced cardinal power domain [CC79a] or, more generally, a cofibered domain [Ven96]. However, since they track one numeric abstract domain for each predicate, there is no bound on the number of states that they infer.

Interestingly, when state spaces are bounded, disjunctive invariants can be encoded using integral polyhedra [Sim08a]. However, since even rational polyhedra are expensive, storing disjunctive information explicitly seems to be preferable.

4.7 Conclusion

We presented a cofibered domain that tracks implications between predicates. This domain takes a single numeric abstract domain as child and thereby avoids tracking several child domains which is the most prominent way to encode disjunctive information. Compared to trace partitioning [MR05] our approach dynamically synthesizes predicates to separate states only where the states are different. No user supplied hints are necessary to find such points of precision loss.
Besides the ability to express non-convex invariants the domain enriches numeric domains with relational (dis-)inequalities. The deduction mechanism is kept simple for scalability

thus we cannot infer new inequalities on joins as e.g. polyhedra [CH78]. However, one possible extension to our domain is to track predicates in conjunctive normal form (CNF) and use an SMT solver for the entailment test to improve $\mathit{fixapply}$.

We illustrated that our domain solves challenging verification examples form the literature while using a simple deduction and reduction mechanism in form of the two novel functions synth and $\mathit{fixapply}$. Especially for path-sensitive invariants, necessary for e.g. locking schemes [Sch+14] the domain significantly improves precision.

One shortcoming of our *predicate* domain is that it has difficulties separating two joined states s_1 and s_2 if one state subsumes the other, e.g. $s_1 \sqsubseteq s_2$. This situation is fairly common when joining program paths with a variable v not being defined on all paths. In those cases the defined value needs to be joined with an undefined v, that is, $v = \top$. As $\top$ subsumes any other values for v the join incurs a maximal precision loss. In the next chapter we show how it is possible by using relational information to recover the value of a variable even after a join with $\top$.

5 The *Undefined* Domain

5.1 Introduction

Static analyses that are based on relational numeric domains are often restricted to programs with limited dynamic memory allocation and without recursive functions [Bla+03a]. In particular, problems occur when the numeric domain has to track a changing number of memory cells or when it has to deal with uninitialized variables. The following example illustrates the problem.

```
1 if (rnd()) {
2   x = 1;
3   y = 2;
4 } else {
5   x = 0;
6 }
7 ...
```

Polyhedra/ Intervals	$\{x = 1, y = 2\} \sqcup \{x = 0, y = \top\}$ $= \{x \in \{0, 1\}, y = \top\}$
Polyhedra as Undefined child ($\mathcal{U} \triangleright \mathcal{PO}$)	$\{x = 1, f_x = 1, y = 2, f_y = 1\}$ $\sqcup \{x = 0, f_x = 1, y = 2, f_y = 0\}$ $= \{x \in \{0, 1\}, f_x = 1, y = 2, f_y = x\}$

Figure 5.1: Non-initialized variables.

In the C program in Fig. 5.1 a variable y may remain undefined. Here, one conditional branch initializes variables x and y, whereas the other branch only initializes variable x, leaving y undefined. When the resulting states are joined, y has to be introduced in the latter state with an unrestricted value $\top$, giving the joined state $\{x = 1, y = 2\} \cup \{x = 0, y = \top\}$. However, introducing variables with value $\top$ can lead to a loss of precision. In particular, the implication $x = 1 \Rightarrow y = 2$ is lost in domains whose state is a convex set. For instance, when using the relational *polyhedra* domain, the joined state $\{x \in \{0, 1\}, y = \top\}$ (shown in the first row of the table) is only as precise as the join over the intervals, in that any relation between x and y is lost.

As a solution to this problem of non-initialized variables, we propose a dedicated abstract domain called the *Undefined* domain which is a cofibered domain. The child can be an arbitrary numeric domain. We require that non-initialized and non-existent variables are introduced as $\top$. The *Undefined* domain then transparently inserts placeholder values

using a so-called copy-and-paste operation. It additionally tracks a flag f_x that indicates if variable x is defined, thereby enabling the child domain to infer relations with this flag, e.g. "y is defined iff $x = 1$".

We illustrate the *Undefined* domain by performing an abstract interpretation of the program in Fig. 5.1 using the *Undefined* domain with the *polyhedra* domain as its child ($\mathcal{U} \triangleright \mathcal{PO}$). The resulting state of the **then**-branch is represented by the child state $\{x = 1, f_x = 1, y = 2, f_y = 1\}$. Here, flags f_x and f_y have value 1, indicating that x and y are defined. The resulting state of the **else**-branch is modeled by the child state $\{x = 0, f_x = 1, y = 2, f_y = 0\}$. Flag f_y has value 0, indicating that y appears to have the value $\top$ at the interface of the *Undefined* domain. As before, the *Undefined* domain has used the value of y from the **then**-branch as a placeholder value. As shown in the second row of the table, the joined child state $\{x \in \{0, 1\}, f_x = 1, y = 2, f_y = x\}$ now indicates that $x = 1$ implies $f_y = 1$ and thus $y = 2$, an invariant that is retained although the state is approximated by the *polyhedra* domain.

Any existing numeric domains can be wrapped by the *Undefined* domain. The resulting domain is a drop-in replacement for the original numeric domain. The *Undefined* domain transparently manages flags for all variables that may be undefined, thereby ensuring that all operations on the domain are sound even if some of the variables mentioned in the operations have been replaced by placeholder values. We provide an implementation of the *Undefined* domain that partitions the flags into groups of flags with equal valuations. By collapsing each group into one single flag, it minimizes the required number of flag variables.

5.2 The *Undefined* Domain

Numeric domains may provide operations that change the *support set* of a numeric state, that is, the set of variables for which the domain holds numeric valuations. Joining and comparing states with different support sets is often preceded by a process that makes their support sets equal. As described in Sect. 2.5 we use cofibered domains [Ven96] to implement modular domains that may have different support-sets. This construction allows to systematically derive variants of the compare and join operations that adjust the support sets themselves.

Let $\mathcal{X}_V$ be the set of program variables and $\mathcal{D}$ an abstract domain. In this work, we assume that each numeric state $d \in \mathcal{D}$ has a *support set* $\chi(d) \subseteq \mathcal{X}_V$ that represents the set of variables for which state d holds valuations. Then each state $d \in \mathcal{D}$ represents a set of vectors of dimension $|\chi(d)|$. Since program variables may be introduced and removed during a program run, the numeric domain must provide operations that add or remove variables to and from the support set. Removing a variable x from a state $d \in \mathcal{D}$ with $x \in \chi(d)$ is denoted by a function $drop_{\mathcal{D},x} : \mathcal{D} \to \mathcal{D}$. Adding an unrestricted variable x to a state $d \in \mathcal{D}$ with $x \notin \chi(d)$ is denoted by a function $add_{\mathcal{D},x} : \mathcal{D} \to \mathcal{D}$. These functions are lifted to sets of variables by repeated application of the add and $drop$ operations, that is, $add_{\mathcal{D},X} := \bigcirc_{x \in X} add_{\mathcal{D},x}$ and $drop_{\mathcal{D},X} := \bigcirc_{x \in X} drop_{\mathcal{D},x}$ where $\bigcirc$ denotes function composition. These are required as comparing and joining two states $d_1 \in \mathcal{D}$ and $d_2 \in \mathcal{D}$

with different support sets requires to add missing variables to d_1 and d_2 beforehand.

5.2.1 Definition of the Domain

The *Undefined* domain is a cofibered domain [Ven96] or functor domain [Cou+06]: Each state holds a state of a *child* domain $\mathcal{D}$, and domain operations are forwarded to domain operations on this child domain. We denote the *Undefined* domain as a cofibered domain $(\mathcal{U} \triangleright \mathcal{D}, \sqsubseteq_{\mathcal{U}}, \sqcup_{\mathcal{U}}, \sqcap_{\mathcal{U}})$ that transforms a child domain $(\mathcal{D}, \sqsubseteq_{\mathcal{D}}, \sqcup_{\mathcal{D}}, \sqcap_{\mathcal{D}})$. An element of the *Undefined* domain that has a child state $d \in \mathcal{D}$ is denoted by $\langle u, d\rangle \in \mathcal{U} \triangleright \mathcal{D}$. The state u denotes the mapping from each x to its flag f_x. Here, for each variable $x \in |\chi(\langle u, d\rangle)|$ of the *Undefined* domain, its child domain holds a variable x and a flag f_x. When $f_x = 1$ in the child domain, the value of x is given by the value of x in the child domain. When $f_x = 0$, variable x is unrestricted and the value stored for x in the child domain is just a placeholder. As a consequence, every numeric state of dimension n is modeled by a child state of dimension $2n$. We later detail how fewer dimensions suffice.

Adding and Removing Dimensions Removing a variable x from a state $\langle u, d\rangle \in \mathcal{U} \triangleright \mathcal{D}$ consists of removing variable x and the corresponding flag f_x from the child state, where $f_x = u(x)$. Thus, we define $drop_{\mathcal{U},x}(\langle u, d\rangle) := \langle u, drop_{\mathcal{D},\{x,f_x\}}(d)\rangle$. Adding a variable x to a state $\langle u, s\rangle$ is done by simply adding an unrestricted variable x and the corresponding flag f_x with value one to the child state d.

Joining, Widening and Comparing States Two states $\langle u_1, d_1\rangle$ and $\langle u_1, d_2\rangle$ with equal support sets $\chi(\langle u_1, d_1\rangle) = \chi(\langle u_2, d_2\rangle)$ are compared, joined or widened by performing these operations on the respective child state d_1 and d_2. For states $\langle u_1, d_1\rangle$ and $\langle u_2, d_2\rangle$ with different support sets $\chi(d_1) \neq \chi(d_2)$, their support sets are made equal by performing $add_{\mathcal{D},x}$ operations on d_1 and d_2 before they can be compared, joined or widened.

$$
\begin{aligned}
&[\![y := f(x_1, \ldots, x_n)]\!]^{\mathcal{U}}\langle u, d\rangle := \\
&\qquad \langle u, [\![y := f(x_1, \ldots, x_n);\ f_y := \bigwedge_{i=1}^{n} f_{x_i}]\!]^{\mathcal{D}} d\rangle \\
&[\![f(x_1, \ldots, x_n) \leq 0]\!]^{\mathcal{U}}\langle u, d\rangle := \\
&\qquad \langle u, ([\![f(x_1, \ldots, x_n) \leq 0; \bigwedge_{i=1}^{n} f_{x_i} = 1]\!]^{\mathcal{D}} d \ \sqcup_{\mathcal{D}}\ [\![\bigvee_{i=1}^{n} f_{x_i} = 0]\!]^{\mathcal{D}} d)\rangle
\end{aligned}
$$

Figure 5.2: Transfer functions for unary operations.

Transfer Functions Figure 5.2 shows the transfer functions for tests and assignments as applied on a state $\langle u, s\rangle \in \mathcal{U} \triangleright \mathcal{D}$. An assignment $y := f(x_1, \ldots, x_n)$ is directly executed on the child domain. Since the resulting value y is only valid if all variables $x_1, \ldots, x_n$

are defined (that is, if all $f_i = 1$), the flag f_y is set to the conjunction $\bigwedge_{i=1}^{n} f_{x_i}$. A test $f(x_1, \ldots, x_n) \leq 0$ is performed by first splitting the state into one state where all $f_i = 1$ and one state where $f_i = 0$ for some i. The test is then performed on the former state, while the latter state remains unchanged. After that, both states are joined.

The given semantics of the *Undefined* domain is still impractical, as it stores one additional flag variable for each variable in the child state, and its specification is incomplete, as it does not fully specify how missing variables are added. The next section describes how the number of flag variables can be reduced and suggests a copy-and-paste operation that adds missing variables in a clever way: it copies relations between those variables that are missing in the respective other domain.

5.3 Practical Implementation of the *Undefined* Domain

In this section we propose an implementation of the *Undefined* domain that is practical in the following two senses: firstly, it associates a flag with a set of variables rather than with each variable, thus yielding a more scalable domain; secondly, it uses a copy-and-paste operation that transfers the valuations of whole sets of variables to another domain, thereby allowing for retaining relational information between variables of a partition. After some definitions, we consider each aspect in turn.

5.3.1 Definition of Partitions

Let $\mathcal{X}_V$ denote the program variables and $\mathcal{F} \subseteq \mathcal{X}_V$ the variables used as flags. A state of the *Undefined* domain $\mathcal{U} \rhd A$ is given by $\langle u, a \rangle$ with child state $a \in A$ and a partial mapping $u : \mathcal{X}_V \dashrightarrow \mathcal{F}$. This mapping takes each variable in the state's support set to a flag that tracks whether this variable is defined. Thus, the support set of child state a is $\chi(a) = \mathit{dom}(u) \cup \mathrm{img}(u)$ where $\mathit{dom}(u)$ denotes the domain of u and $\mathrm{img}(u)$ denotes the image of u. We allow several program variables to map to the same flag variable, thereby inducing a partitioning of program variables. For each mapping u this partitioning is given by $\Pi(u) := \{u^{-1}(f) \mid f \in \mathrm{img}(u)\}$, where $u^{-1} : \mathcal{F} \to \wp(\mathcal{X}_V)$ is the reverse relation of u. For better legibility, we sometimes denote u by its reverse relation. Thus, for $u = [x_0 \mapsto f_0, x_1 \mapsto f_1, x_2 \mapsto f_0, x_3 \mapsto f_1]$ we write $[f_0 \mapsto \{x_0, x_2\}, f_1 \mapsto \{x_1, x_3\}]$. We now detail how to manage flags when partitions change.

5.3.2 Making Partitions Compatible

Whenever two states $\langle u_1, a_1 \rangle$ and $\langle u_2, a_2 \rangle$ are compared or joined, their partitioning $\Pi(u_1)$ and $\Pi(u_2)$ must agree. To this end, the coarsest partitioning $P := \{p_1 \cap p_2 \mid p_1 \in \Pi(u_1),\, p_2 \in \Pi(u_2)\}$ whose partitions can be merged to give either $\Pi(u_1)$ or $\Pi(u_2)$ is calculated. We then associate each partition $p \in P$ with a fresh flag f_p, thereby obtaining a new state $u_{12}^{-1} = \bigcup_{p \in P} [f_p \mapsto p]$. Let $u_{12} = \mathit{common}(u_1, u_2)$ abbreviate this operation. Since u_{12} associates different (and possibly more) flags with its partitions than u_1 and u_2, the flags

$x \in \mathcal{X}_V$	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
$u_1(x)$	f_1	f_1	f_1	f_1	f_2	f_2	f_2	f_2	f_3	f_3
$u_2(x)$	f_4	f_4	f_5	f_5	f_5	f_5	f_6	f_6	f_7	f_7
$u_{12}(x)$	f_8	f_8	f_9	f_9	f_{10}	f_{10}	f_{11}	f_{11}	f_{12}	f_{12}

Figure 5.3: Making compatible of two partitions $u_{12} = \mathit{common}(u_1, u_2)$.

stored in a_1 and a_2 have to be adjusted. Thus, let $\mathit{trans}_{u_i}^{u_{12}}(f) := \{u_{12}(x) \mid x \in u_i^{-1}(f)\}$ denote the flags of those partitions in u_{12} whose union is associated with f in u_i. We transfer the value of f to the flags $\{f_1, \ldots, f_n\} \in \mathit{trans}_{u_i}^{u_{12}}(f)$ using the assignment $\mathit{adjOne}_{u_i}^{u_{12}}(f) := [\![f_1 := f]\!] \cdots [\![f_n := f]\!]$. The assignment for all partitions is then given by the composition $\mathit{adjust}_{u_i}^{u_{12}} := \bigcirc_{f \in \mathrm{img}(u_i)} \mathit{adjOne}_{u_i}^{u_{12}}(f)$. Making two child states a_1 and a_2 compatible with u_{12} requires that the flags $\mathrm{img}(u_{12})$ are introduced, the renaming $\mathit{adjust}_{u_i}^{u_{12}}$ is applied, and that the now stale flags $\mathrm{img}(u_i)$ are removed. These operations are aggregated by the function $\mathit{compat}_{u_i}^{u_{12}} = \mathit{drop}_{\mathrm{img}(u_i)} \circ \mathit{adjust}_{u_i}^{u_{12}} \circ \mathit{add}_{\mathrm{img}(u_{12})}$.

Consider the task of making two domains, $\langle u_1, a_1\rangle$ and $\langle u_2, a_2\rangle$ compatible where u_1 and u_2 are given by the first two rows in Fig. 5.3. First, the new partition $u_{12} = \mathit{common}(u_1, u_2)$ is calculated as shown in the last line of Fig. 5.3. In order to adjust a_1 to be compatible with u_{12}, we compute $a_1' = \mathit{compat}_{u_1}^{u_{12}}(a_1) = \mathit{drop}_{\{f_1,f_2,f_3\}}(\mathit{adjust}_{u_1}^{u_{12}}(\mathit{add}_{\{f_8,\ldots f_{12}\}}(a_1)))$. The function $\mathit{adjust}_{u_1}^{u_{12}}$ expands to $\mathit{adjOne}_{u_1}^{u_{12}}(f_1) \cdots \mathit{adjOne}_{u_1}^{u_{12}}(f_3) = [\![f_8 = f_1, f_9 = f_1]\!] \cdot [\![f_{10} = f_2, f_{11} = f_2]\!] \cdot [\![f_{12} = f_3]\!]$. Computing $a_2' = \mathit{compat}_{u_2}^{u_{12}}(a_2)$ analogously suffices to perform any operation that requires $\chi(a_1') = \chi(a_2')$, such as $\langle u_1, a_1\rangle \sqcup_{\mathcal{U}} \langle u_2, a_2\rangle = \langle u_{12}, a_1' \sqcup_A a_2'\rangle$. This concludes the process of making domains compatible which allows us to associate a flag with a partition rather than a single variable. While tracking fewer flags improves performance, we now detail how precision can be improved.

5.3.3 Rescuing Relational Information

Tracking whether a set of variables is undefined is only useful if the content of undefined variables is replaced by other values that lead to less precision loss. In order to distinguish variables that are always undefined, we use a special flag f_{undef} whose value is always zero in the child domain. The variables $u^{-1}(f_{\mathit{undef}})$ associated with f_{undef} are omitted from the child domain. Due to this, computing the join of two states $\langle u_1, a_1\rangle \sqcup_{\mathcal{U}} \langle u_2, a_2\rangle$ requires that the variables $X_{12} = u_1^{-1}(f_{\mathit{undef}}) \setminus u_2^{-1}(f_{\mathit{undef}})$ that are undefined in a_1 but not in a_2 are added to a_1 before the child states a_i can be joined (and vice-versa). To this end, define a function $\mathit{copyAndPaste}_{\mathcal{D},X} : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ with $r = \mathit{copyAndPaste}_{\mathcal{D},X}(a_1, a_2)$ such that variables X are copied from a_1 into a_2, yielding r where $X \subseteq \chi(a_1)$, $\chi(a_2) \cap X = \emptyset$ and $\chi(r) = \chi(a_2) \cup X$. We illustrate *copyAndPaste* with an example.

Suppose the following modified version of the introductory example in Fig. 5.1 is given in Fig. 5.4 where `rnd()` returns a random number. Consider analyzing this program with a state $\langle u_1, a_1\rangle$ where $u_1 = [f_{\mathit{undef}} \mapsto \{x, y\}]$ and $a_1 = \{f_{\mathit{undef}} = 0\}$ is a convex polyhedron. Note that $\chi(a_1) = \{f_{\mathit{undef}}\}$ since the variables $x, y \in u_1^{-1}(f_{\mathit{undef}})$ are not stored in a_1 as explained above. The state at line 5 becomes $\langle u_2, a_2\rangle$ where $u_2 = [f_{xy} \mapsto \{x, y\}]$ and

```
1 int x,y;
2 if (rnd(0, 1)) {
3   x = rnd(0, 10);
4   y = x;
5 }
6 ...
```

Figure 5.4: Assigning a random value in a loop.

$a_2 = \{x = y, x \in [0, 10], f_{xy} = 1\}$. The benefit of not storing x, y in a_1 is that they can be introduced using $a_1' = \mathit{copyAndPaste}_{A,\{x,y\}}(a_2, a_1) = \{x = y, x \in [0, 10], f_{\mathit{undef}} = 0\}$ that extracts all information over x, y in a_2 and adds it to a_1. In order to state that these variables should be considered as undefined in a_1', we add a new flag $f_{xy} = 0$ yielding $\langle u_1', a_1' \rangle$ with $u_1' = [f_{xy} \mapsto \{x, y\}]$ and $a_1' = \{x = y, x \in [0, 10], f_{xy} = 0\}$. Now the state after line 5 can be computed as $\langle u_1', a_1' \rangle \sqcup_{\mathcal{U}} \langle u_2, a_2' \rangle = \langle u_{12}, a_1' \sqcup_A a_2' \rangle$ where $u_{12} = u_1'$ and a_2' is a_2 in which $f_{xy} = 1$ is added. The result $a_1' \sqcup_A a_2' = \{x = y, x \in [0, 10], 0 \leq f_{xy} \leq 1, f_{\mathit{undef}} = 0\}$ retains the equality $x = y$, thereby improving the additional relational information.

5.3.4 Transfer Functions

Figure 5.5 illustrates the implementation of the $\boxdot = \sqcup, \sqsubseteq, \nabla$ functions which use *copyAndPaste* on the child domain A. We define $r = \mathit{copyAndPaste}_{\mathcal{D},X}(a_1, a_2)$ as $r = a_1 \sqcap_{\mathcal{D}} \mathit{drop}_{(\mathcal{D},\chi(a_2)\setminus X)}(a_2)$ where $\sqcap_{\mathcal{D}}$ is a greatest lower bound on two abstract states that adds missing dimensions as needed. The idea is to remove all dimensions from a_2 that should not be copied before merging the remaining relations over X into a_1 using the meet $\sqcap_{\mathcal{D}}$. For each binary operation $\boxdot$, Eq. 5.1 shows how the states are made compatible as described above before applying $\boxdot$ on the child domains.

Figure 5.5 also defines other transfer functions of the *Undefined* domain. Adding an unrestricted dimension x using add merely adds a mapping $f_{\mathit{undef}} \mapsto x$ to the undefined mapping (Eq. 5.2). Removing a variable x using drop needs to check if x is not stored in a (Eq. 5.3), or if it was the last variable in its partition (Eq. 5.4). Assigning to a variable y computes the set of flags Ψ that must be one to make the result defined (Eq. 5.5). If $f_{\mathit{undef}} \in \Psi$ then y is always undefined and executing the assignment on the child is not necessary. If a single flag f suffices to make y defined, y is added to the partition of f. In the general case, a new flag f_y is created (Eq. 5.6) that represents the validity of the new partition $\{y\}$ (Eq. 5.7). Applying a test (Eq. 5.8) partitions the child state a into one where all variables occurring in the test are defined ($\psi = |\Phi|$) and one where they are possibly undefined ($\psi < |\Phi|$). Only in the former case, the test is applied.

$$
\begin{aligned}
\langle u_1, a_1\rangle \boxdot_{\mathcal{U}} \langle u_2, a_2\rangle \quad &= \quad \text{let for } i = 1, 2 && (5.1)\\
&\qquad X_i = u_i^{-1}(f_{\mathit{undef}})\\
&\qquad u_i' = u_i[x \mapsto f_i]_{x \in X_i \setminus X_{3-i}} \text{ where } f_i \text{ fresh}\\
&\qquad u_{12} = \mathit{common}(u_1', u_2')\\
&\qquad a_i' = \mathit{copyAndPaste}_{A, X_{3-i}\setminus X_i}(a_{3-i}, a_i)\\
&\quad \text{in} \quad \langle u_{12}, (\mathit{compat}_{u_1}^{u_{12}}(a_1') \boxdot_A \mathit{compat}_{u_2}^{u_{12}}(a_2'))\rangle\\
\mathit{add}_x(\langle u, a\rangle) \quad &= \quad \langle u[x \mapsto f_{\mathit{undef}}], a\rangle && (5.2)\\
\mathit{drop}_x(\langle u, a\rangle) \quad &= \quad \text{if } u(x) = f_{\mathit{undef}} \text{ then } \langle (u \setminus x), a\rangle \text{ else} && (5.3)\\
&\qquad \text{if } |\{y \in \mathit{dom}(u) \mid u(x) = u(y)\}| > 1\\
&\qquad \text{then } \langle (u \setminus x), \mathit{drop}_{A,x}(a)\rangle\\
&\qquad \text{else } \langle (u \setminus x), \mathit{drop}_{A,\{x,u(x)\}}(a)\rangle && (5.4)\\
[\![y := f(x_1, \ldots, x_n)]\!]^{\mathcal{U}}\langle u, a\rangle \quad &= \quad \text{let } \Phi := \{u(x_1), \ldots u(x_n)\} \text{ in} && (5.5)\\
&\quad \text{if} \quad f_{\mathit{undef}} \in \Phi \text{ then } \mathit{add}_{\mathcal{U},y}(\mathit{drop}_{\mathcal{U},y}(\langle u, a\rangle)) \text{ else}\\
&\quad \text{if} \quad \Phi = \{f\} \text{ then } \langle u[y \mapsto f], [\![y := f(x_1, \ldots, x_n);]\!]^A a\rangle\\
&\qquad \text{else let } f_y \text{ fresh and } u' = u[y \mapsto f_y] \text{ in} && (5.6)\\
&\qquad \langle u', [\![y := f(x_1, \ldots, x_n);\ f_y := \sum_{f \in \Phi} f = |\Phi|]\!]^A a\rangle && (5.7)\\
[\![f(x_1, \ldots, x_n) \le 0]\!]^{\mathcal{U}}\langle u, a\rangle \quad &= \quad \text{let } \Phi := \{u(x_1), \ldots u(x_n)\} \text{ and } \psi = \sum_{f \in \Phi} f \text{ in} && (5.8)\\
&\qquad \langle u, [\![f(x_1, \ldots, x_n) \le 0; \psi = |\Phi|]\!]^A a \ \sqcup_A \ [\![\psi < |\Phi|]\!]^A a\rangle
\end{aligned}
$$

Figure 5.5: Transfer functions for binary operations $\boxdot = \sqcup, \sqsubseteq, \nabla$, and unary operations.

5.4 Applications to Interprocedural Analysis

We now illustrate the utility of the *Undefined* domain by using examples from the interprocedural analysis of function calls. For the sake of limiting the memory consumption of an analyzer, it is desirable to merge the states of certain call sites of a function `f` into one. To this end, we use a stack functor domain $\mathcal{S} \triangleright \mathcal{C}$ (with child domain C) that manages a set of stack frames. Here, $\mathcal{S}$ tracks one dedicated *active* stack frame that represents the currently executed function `f`. In order to track to which stack frame the analysis has to return to when leaving the current function, the state $s \in \mathcal{S}$ is a directed graph with stack frames as nodes, where the more recently called function points to its caller. Consider for example the program in Fig. 5.6. Here, function `f` is called twice. First, it is called by function `a`, which in turn is called from `main`. Figure 5.7 a) shows how the first call path via `a` forms a linked list of stack frames, which we denote by s_a. Figure 5.7 b) shows the graph of stack frames for the second call to `f` via `b`, denoted by s_b. In order to combine two graphs s_a and s_b, we

```
1 main() {          5 a(int x) {        8 b(int y) {        11 f(int z) {
2     a(0);         6     f(x);         9     f(y);         12     ...
3     b(1);         7 }                 10 }                13 }
4 }
```

Figure 5.6: Function calls example.

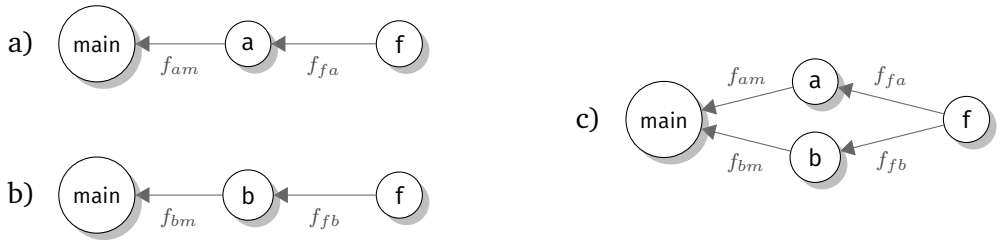


Figure 5.7: Combining several call sites into one state.

follow [SS13] in qualifying the graph edges by numeric flags, that is, numeric variables that can take on the values 0 or 1. Let $\langle s_a, c_a \rangle$ with $c_a = \{x = z = 0, f_{fa} = 1, f_{am} = 1\} \in \mathcal{PO}$ denote the abstract state (here $c_a \in \mathcal{PO}$ are convex polyhedra [CH78]) on entry to `f` for the path in Fig. 5.7 a). In c_a, the flag f_{fa} has value one, indicating that the node (stack frame) of `a` is the predecessor of the node (stack frame) of `f`. Analogous for f_{am} that qualifies the edge between the stack frame of `f` and of `main`. Symmetrically, for the path shown in Fig. 5.7 b) the state is $c_b = \{y = z = 1, f_{fb} = 1, f_{bm} = 1\}$.

The two graphs s_a and s_b are merged into the combined graph of stack frames s in Fig. 5.7 c). In order to capture that the `b` node is not a predecessor of `f` in s_a, we add the flag $f_{fb} = 0$ to c_a and analogously we add $f_{bm} = 0$, yielding $c'_a = \{x = z = 0, f_{fa} = 1, f_{fb} = 0, f_{am} = 1, f_{bm} = 0\}$. Symmetrically, we enrich c_b to $c'_b = \{y = z = 1, f_{fa} = 0, f_{fb} = 1, f_{am} = 0, f_{bm} = 1\}$. Overall, we obtain the state $\langle s, c'_a \sqcup_{\mathcal{C}} c'_b \rangle = \langle s, \{x = \top, y = \top, 0 \leq z \leq 1, z = f_{bm} = f_{fb} = 1 - f_{am} = 1 - f_{fa}\} \rangle$.

Note that all information within the stack frames, namely x and y is lost. The *Undefined* domain can improve this situation: we re-analyze the example using the domain $\mathcal{U} \triangleright \mathcal{S} \triangleright \mathcal{PO}$. The net effect is that in the last step, instead of $\langle s, c'_a \sqcup_{\mathcal{C}} c'_b \rangle$ we compute $\langle s, \langle u, c'_a \rangle \sqcup_{\mathcal{U}} \langle u, c'_b \rangle \rangle$ where $u \in \mathcal{U}$ is the mapping stating that all variables are defined. By the definition of $\sqcup_{\mathcal{U}}$ the missing variable x is added to $\langle u, c'_b \rangle$ giving $\langle u_b, c'_b \rangle$ with $u_b = [x \mapsto f_{undef}]$ and, analogously, the left argument becomes $\langle u_a, c'_a \rangle$ with $u_a = [y \mapsto f_{undef}]$. Computing the join $\langle u_a, c'_a \rangle \sqcup_{\mathcal{U}} \langle u_b, c'_b \rangle$ makes the two undefined states u_a and u_b compatible to $u = [x \mapsto f_x, y \mapsto f_y]$. The numeric state c'_a is modified by adding $f_x = 1, f_y = 0$ and

copying $y = 1$ from c'_b whereas s'_b is modified by adding $f_x = 0, f_y = 1$ and copying $x = 0$ from c'_a. The state that `f` is analyzed with is thus $\langle s, \langle u, \{x = 0, y = 1, 0 \leq z \leq 1, z = f_x = f_{bm} = f_{fb} = 1 - f_y = 1 - f_{am} = 1 - f_{fa}\}\rangle\rangle$.

The benefit of the *Undefined* domain is thus that, upon returning from `f`, the content of the predecessor stack frames is still available since $x = 0, y = 1$ is retained in the join of the two call sites. Our analysis infers more intricate invariants if pointers are passed, since the flags of the *Undefined* domain form an equality relation with the points-to flags [SMS13].

5.5 Experimental Results

We evaluated the Undefined domain in our analyzer using a domain stack $\mathcal{S} \rhd \mathcal{U} \rhd \mathcal{A} \rhd \mathcal{CR} \rhd \mathcal{I}$ where $\mathcal{S}$ maintains stack frames and memory allocated on the stack and $\mathcal{U}$ is the *Undefined* domain. The remaining domains are numeric; they track affine equalities $\mathcal{A}$, congruences $\mathcal{CR}$ and intervals $\mathcal{I}$. In order to estimate the performance of the *Undefined* domain, we also evaluated the examples with domain stack $\mathcal{S} \rhd \mathcal{A} \rhd \mathcal{CR} \rhd \mathcal{I}$ that is, without domain $\mathcal{U}$.

The stack domain $\mathcal{S}$ recognizes function boundaries by observing the stack pointer whenever the control flow changes through a jump, call or return instruction (see Sect. 2.5.1). The examples were analyzed using executables compiled for the Intel x86 platform. On this platform the return instruction is translated into a read access to the previous stack frame in order to retrieve the return address and a jump to this address. The *Undefined* domain is thereby key to infer a precise address since, for Fig. 5.7, stack frames a and b are both read and joined before the jump is executed.

example	$\mathcal{U}$	insns.	time	memory	variables	undef. flags	warnings
call stack 1 (Fig. 5.6)		76	377	41.9	48	–	7
call stack 1 (Fig. 5.6)	✓	114	450	42.0	50	2	0
call stack 2		178	416	42.6	72	–	7
call stack 2	✓	254	641	42.0	74	2	0
call stack 3		76	422	41.9	48	–	7
call stack 3	✓	153	718	42.4	66	4	0
call stack 4		88	920	42.5	52	–	8
call stack 4	✓	128	702	42.2	54	2	0
call stack 5		90	709	42.0	54	–	8
call stack 5	✓	173	1455	47.3	75	4	0

Figure 5.8: Evaluation of the *Undef* domain.

Figure 5.8 shows the experimental results. Column $\mathcal{U}$ indicates whether the *Undefined* domain is used, followed by the number of machine instructions in the program that were analyzed; columns *time* and *memory* show the runtime in milliseconds and the memory consumption in megabytes, averaged over several runs on a 3.2 GHz Core i7 Linux machine. The next column shows the total number of variables tracked, followed by the number of flag variables used by the *Undefined* domain and the number of warnings emitted by the analyzer.

The first line shows the call stack example of Sect. 5.4, followed by variations with more functions and call paths. Call stack examples 4 and 5 differ in that they use pointers to stack variables to pass parameter values. Note that the call stack examples exhibited shorter runtimes without the *Undefined* domain, because precision loss made it impossible to resolve the return addresses, so that the examples could only partially be analyzed. This is reflected in the number of analyzed instructions. For the same reason the number of total variables in the call stack of example 3 and 5 without the domain are much lower than with the *Undefined* domain. The examples show that the additional variables that are necessary as flags for the *Undefined* domain are only few compared to the total number of variables in the program.

5.6 Related Work

We addressed the challenge of tracking the content of memory that does not exist in all traces. Many existing analyses use some ad-hoc methods to approximate what we have put on a sound mathematical basis: the ability to store both, precise and undefined values for variables in a single state. For instance, recency abstraction [BR06] implicitly retains the defined value when the state is joined. When a purely logic description is used [Rey02; SRW02], the distinction between defined and undefined content is simply expressed using disjunction. In Astrée [Bla+03a], disjunction is expressed using the decision tree domain that tracks two separate child domains depending on the value of a flag. The effect is similar to standard path-sensitive analyses in that tracking two states duplicates analysis time. More sophisticated analyses merge states on different paths if a finite abstraction determines that they are similar [DLS02]. Future work will determine whether this technique can be implemented as a combinator in our domain stack.

The Undefined domain partially allows the encoding of conditional invariants. While this problem has been studied for logical domains [GMT08], we provide a solution that enables existing numeric domains to infer certain conditional invariants, e.g. those guarded by the existence of objects. For overly complex invariants, our approach exploits the ability of numeric domains to gradually lose precision.

5.7 Conclusion

We addressed the task of storing a single state in cases where a piece of memory has undefined content. We illustrated the power of this domain by defining a specific instance, namely the *Undefined* domain, that improves precision in common program analysis tasks. Its novel copy-and-paste operation even retains relational information.

Part III

Precision Improvements using Dynamic Analysis

6 Dynamically Started Static Analysis

The strength of sound static analysis tools applied to find buffer overflows in executable programs is that many memory accesses can be shown to be within bounds which alleviates the security engineer from examining them. However, the inherent approximation applied during a static analysis implies that certain indirect jumps cannot be resolved precisely so that byte sequences are incorrectly interpreted as part of the program, leading to further imprecision and scalability problems. We present an approach that combines dynamic and static analysis with which it is possible to run the program through one path of its start-up code and to change over to a static analysis once the program reads input from a file or the network. Our approach makes potentially vulnerable memory accesses easier to find since the state space with which the remainder of the program is analyzed is generic only at those memory addresses that contain user input.

6.1 Introduction

Security engineers face the challenge of finding vulnerabilities in software that is often only available to them in the form of an executable. Many techniques have been developed to aid in this endeavor, such as fuzzing [GLM12], differencing [SMA05], monitoring execution [CV04], symbolic execution [Jaf+12] or reachability analysis as presented in this thesis. The goal of these techniques is to support the engineer in finding an input to the program so that a path is taken that leads to a vulnerable code construct (such as an array access that is out-of-bounds for certain inputs). From an academic point-of-view, a reachability analysis that is sound (i.e. it only over-approximates the reachable states of a program) is most satisfactory: if a program construct is not vulnerable even under the over-approximated state, then it is never vulnerable. Hence, a sound reachability analysis can prove the absence of vulnerabilities, an ability that is beyond other unsound (not strictly over-approximating) techniques such as fuzzing and test data generation. In practice, however, sound reachability analysis faces the problem of imprecision: a program construct may be flagged as being vulnerable due to the state space added by over-approximation. Such a warning is known as a false positive. The problem of imprecision is exacerbated in the analysis of executables due to indirect (calculated) jumps [Bal07; KVZ09]: if the target of an indirect jump is too imprecise, a plethora of instructions are analyzed that can never be reached in the actual

program, thus leading to an avalanche of false positives and scalability problems. Thus, general static analysis is currently not a good fit for finding vulnerabilities in executables.

Interestingly, in most circumstances, it is relatively easy to determine where malicious data can be fed into a program by recording an average trace through an executable and observing where it reads from a file or the network. The challenge now lies in finding the right data to trigger a vulnerability in the code. One way forward is to search the trace for patterns such as calls to string copying functions or calls to linked-in libraries that have known vulnerabilities [Orm06] and to alter the input such that a vulnerability is triggered. However, not only is it difficult to alter the input data appropriately [God07; SMA05], but furthermore, vulnerabilities that lie in hand-coded functions that do not fit any pattern cannot be found this way.

In this chapter, we propose to find new vulnerabilities by combining sound static analysis and program tracing. The idea is to utilize a single trace and to generalize the program input along this trace using static analysis. To this end, we let the user specify the point p_s in the program where potentially malicious input can be read. This point could be where a PDF or HTML file is opened. We run the program and record a trace as soon as p_s is reached. Later we start our analysis with the state of the program as we observed it at p_s. The analysis then executes the code along the path prescribed by the trace. The net effect is that the analysis now infers a state that is generic with respect to the input read after p_s. Thus any vulnerability found must either be triggered by the input or is due to over-approximation. In order to prove the absence of vulnerabilities from inputs read after p_s, we allow the user to incrementally add all other paths that were not taken after p_s. If this succeeds without triggering a warning, the user can be sure that no other input exists that could have triggered an exploit.

The power of our approach of combining dynamic and static analysis lies in sidestepping the problems of scalability and imprecision inherent in whole-program static analysis. At the same time we retain the benefit of working with a sound analysis, namely being able to (partially) verify a program.

6.2 Over-Approximating Static Analysis

Recalling the definitions from Sect. 2.1.3 that a program is correct if $\mathit{Safe}(\langle p, \gamma(s^\sharp)\rangle)$ holds for all $\langle p, s^\sharp\rangle \in \mathcal{A}$ where $p \in P$ is the set of possible instructions. In general, the approximation in an analyzer can be severe so that no information might be available on, say, a certain pointer variable. If the next instruction is a jump to the address stored in that pointer, the analysis not only has to warn that *Safe* does not hold (since the values of the pointer are not all in P) but it would have to continue at all valid instruction addresses P which is too costly and yields no useful results. These problematic precision losses are particularly common in startup code, unpackers in malware, loading of plug-ins and much other code that is executed during initialization.

We propose to circumvent this loss of precision by dynamically starting our static analysis. To this end, we let the user specifies $p_s \in P$ which denotes a point in the program where

input from a file or network is read that can be controlled by an attacker. Such a point can easily be found using e.g. Unix' `strace -i`. We run the program and record a trace $\pi \in \Pi$ that passes through p_s. At p_s, we transfer the state $\langle p_s, s\rangle \in \pi$ by saving it to disk. We continue running the program but only store $\tau = p_s \cdot p_1 \cdot p_2 \cdots \in P^*$ which corresponds to the execution $\pi \in \Pi$ without the states s_i. Given this sequence of program locations, we now define how to analyze this trace in an abstract way.

6.3 Trace Abstraction

The idea in trace abstraction is to infer an abstract state by following the execution path $\tau \in P^*$ of the concrete trace. Let $\tau(i)$ represent the ith program point (that is, $\tau(0) = p_s$) and let $\langle p_s, s\rangle \in \pi$ denote the state that was stored to disk. By choosing $s^\sharp \in S^\sharp$ and $\mathcal{A}$ such that $\langle p_s, s^\sharp\rangle \in \mathcal{A}$ and $s \in \gamma(s^\sharp)$, we now compute the trace abstraction $\mathcal{A}_k^{tr} = m_k$ of k steps as follows:

$$\begin{aligned} m_0 &= \{\langle p_s, s^\sharp\rangle\} \cup \{\langle p, \bot\rangle \mid p \in P, p \neq p_s\} \\ m_{i+1} &= \{\langle p, s^\sharp_{i+1} \sqcup m_i(p)\rangle \mid \langle p, s^\sharp_{i+1}\rangle \in \mathit{Next}^\sharp(\tau(i), m_i(\tau(i))) \wedge p = \tau(i+1)\} \end{aligned}$$

Here, $\bot$ denotes the unreachable state with $s^\sharp \sqcup \bot = s^\sharp$ for all $s^\sharp \in S^\sharp$. The idea of trace abstraction is that each transition from $\tau(i)$ to $\tau(i+1)$ in τ is evaluated by the abstract semantics while all traces that do not lead to $\tau(i+1)$ are ignored. Note that in general, the number of steps k must be provided by the user in order to limit the size of the recorded trace and thereby the analysis time.

```
1 elem_t* init_array (size_t num_elems) {
2   size_t buf_size = num_elems * sizeof(elem_t);
3   elem_t* elems = malloc(buf_size);
4
5   for (int i = 0; i < num_elems; i++)
6     elems[i] = DEFAULT;
7   return elems;
8 }
```

Figure 6.1: An overflowing multiplication may lead to a buffer that is too small.

By using this strategy we can already discover certain vulnerabilities, as illustrated by Fig. 6.1. The C program shows a common coding pattern that is vulnerable to buffer overflows. More complex variations of this pattern are discovered on occasion in current software [OS]. Suppose that `num_elems` is given as an input to the program. An attacker can exploit that the multiplication of unsigned integers in line 2 may overflow in order to allocate a smaller buffer (line 3) than intended by the programmer. For instance, on a 32 bit platform the values for the variables could be `num_elems = 0x80000001` and `sizeof(elem_t) = 4`, for which the multiplication will overflow and result in `buf_size = 4`.

As a consequence, the iteration over the allocated buffer in line 5 will write past the buffer's end and corrupt the heap. Worse, if the attacker can control the data that is written to the buffer, it might be possible to overwrite another object in the heap. If this object contains a virtual dispatch table or other function pointers, the attacker could inject and execute code by overwriting these pointers. A successful exploitation however is more complex thus a heap buffer overflow does not immediately imply code execution.

In order to analyze the example we generate a trace to the `init_array` function where e.g. $num\_elems = 10$. This trace however does not trigger the overflow. We will show the analysis of the trace using the domains stack $\mathcal{WR} \triangleright \mathcal{A} \triangleright \mathcal{I}$, that is, the cofibered *wrapping* domain with *affine* equalities and the *interval* domain as child. Note that the *wrapping* domain has no state and is thus omitted. Starting from the trace when reaching line 4 we track the state $s = \langle \{buf\_size = 4num\_elems\}, \{num\_elems \in [10, 10]\} \rangle$. Iterating the loop along the trace we infer that the loop counter values are $i \in [0, 9]$ in line 5. Next the write in line 6 involves testing if the access is inside the bounds of the $elems$ memory region, that is, we apply the test $4i < buf\_size$ to the domain state. With the state tracked by the analysis the test translates to $4i < 4num\_elems$ and is tested on the *interval* domain as $4{*}[0, 9] < 4{*}[10, 10]$. As this test always holds the analysis of this trace does not find the bug.

Now consider the overflow example mentioned above, where we choose $num\_elems = 0x80000001 = 2^{31} + 1$. The difference in the analysis is that in line 6 when applying the test $4i < buf\_size$ the *wrapping* domain first tests if buf_size is inside its defined range $[0, 2^{32}-1]$, which it is not, due to $buf\_size = 4num\_elems = 2^{33}+4$. Wrapping then assigns the wrapped value to buf_size at the same time removing the affine relation to num_elems. As a result testing $4i < buf\_size$ in line 6 does not hold after 4 iterations as $i \in [0, 2^{31} + 1]$, and $buf\_size \in [4, 4]$. The write thus causes a warning for potential out-of-bounds accesses.

Nevertheless, here our analysis only proved that the given overflow example indeed triggers the bug. However, if we do not have a concrete example that exercises the bug it is still desirable that the analysis automatically finds the bug. In order to achieve this, we must abstract the value of num_elems, that is, we set p_s before the value of num_elems is initialized by e.g. some call to the operating system. The analysis abstracts this call by setting num_elems to $\top$. When evaluating the multiplication, the wrapping domain recognizes that the result may overflow for $num\_elems > 2^{30}$ and thus infers that the buffer allocated by `malloc` has a size in $[0, 2^{32} - 1]$. Since the correctness of the array access, namely $4i < buf\_size$, does not always hold due to the overflow, writing to the array in line 6 will be flagged as being out-of-bounds. The out-of-bounds warning is inferred already in the first loop iteration, as $i = 0$ is not strictly smaller than $buf\_size = [0, 2^{32} - 1]$. Hence, by abstracting the program input that sets num_elems we were able to automatically find the bug without requiring a concrete overflow triggering example.

Suppose now that $num\_elems = 0$ in the generated trace. In this case the loop is not entered and the buffer access in line 6 is never executed. Even though we might choose to

abstract $num\_elems = \top$ the trace stops us from analyzing the loop body. Hence, by using the trace abstraction only, the analysis would miss the vulnerability because it follows the execution path of the trace which does not enter the loop. The next section explains how to discover vulnerabilities even if the trace execution path does not contain them.

6.4 Reachability Analysis

The previous section proposed an analysis that generalized the input values that were read since passing p_s. Even then, it can be that no error is found along this trace. We now allow the user to analyze other parts of the program by adding paths that were neglected so far. These paths start at $p \in \bar{p}_r$ where:

$$\begin{aligned} \bar{p}_r \quad = \quad & \{\tau(i) \mid \langle p, s^\sharp \rangle \in \mathit{Next}^\sharp(\tau(i), m_i(\tau(i))) \wedge \\ & \qquad p \neq \tau(i+1) \wedge s^\sharp \neq \bot \wedge i \in [0, k-1]\} \end{aligned}$$

The idea is to let the user choose points $p \in \bar{p}_r$ for which a reachability analysis can be started. This reachability analysis will perform a fixpoint computation of all states reachable from p, thereby inferring all behaviors that are possible when passing through point p with the generalized input. Note that the test of each loop is in $\bar{p}_r$ so that the user can trigger a fixpoint computation of every loop. We illustrate the effect of choosing e.g. $p = l_8$ in the following example.

```
1  #define MAX_BUF_LEN = 100;
2
3  char* normalize (char* source) {
4    char length = strlen(source);
5    if (length >= MAX_BUF_LEN)
6      return;
7
8    char buffer[MAX_BUF_LEN];
9    strcpy(buffer, source);
10   char* result = parse(buffer);
11   ...
12 }
```

Figure 6.2: A conversion may lead to an unexpected value range.

In Fig. 6.2, an implicit cast from **size_t** to **char** in line 4 is used before the test in line 5 checks if the length of `source` is bigger than 100. However, the **char** datatype is signed on some platforms and any negative value will pass the test. For instance, if `source` contains a string of length 128, `strlen(source)` returns an integer value of 128. Due to the cast to a signed char `length` will contain -128 which would pass the subsequent test (line 5). We assume here a signed char datatype of 8 bits size. Thus $128 = 2^7$ sets the highest bit in the char variable which, when interpreted in two's complement arithmetic equals to -128.

Next, when copying the 128 bytes in `source` to the work buffer (line 9), `strcpy` will write past the end of the allocated `buffer` and overwrite the stack.

When analyzing the example above, a generated trace is unlikely to trigger the vulnerability because `strlen(source)` might be, say, 80 in the trace. We set p_s to the location where `source` is read from some user input as before. Applying the trace abstraction technique from Sect. 6.3, the analysis would not be able to infer that `strcpy` can write past the end of `buffer`. The analysis would follow the execution path of the trace inside `strcpy` and iterate as many times over the copy loop as the trace, thus missing the possible out-of-bounds access. In order to explore paths that were not taken by the trace, the user can choose all positions in $\bar{p}_r$ before line 9 so that the analyzer performs a reachability analysis of the loop(s) within `strcpy`. The loops are then analyzed using widening and thus not bound to the number of iterations recorded in the trace. Thus, a fixpoint of the copy loop in `strcpy` is calculated that over-approximates all lengths of `source` and which infers that the size of `source` is not always smaller than the size of `buffer`.

6.5 Combining Tracing and Analysis

In order to capture a trace of the program execution we combined our existing static analysis framework with Intel's PIN binary instrumentation framework [Luk+05]. The latter provides a lightweight software virtual machine for the x86 platform and allows the insertion of instrumentation code into the executable program. Instrumentation code can be inserted for specific instructions, classes of instructions or for interactions with the operating system. Other instrumentation frameworks could have equally well been used, such as DynInst [HMC94], DynamoRIO [BGA03], or Valgrind [NS07]. A slightly different approach is to employ a complete hardware virtualization framework such as e.g. QEMU [Bel05], but we decided against the complexity of interfacing our analysis with such a framework. The downside is that we are restricted to user-space code as only hardware virtualization can execute and observe kernel-mode code.

The first phase of our analysis is to start the instrumented program which records a trace. Up to program point $p_s \in P$ we are only interested in calls to a fixed set of library functions (e.g. `malloc`, `free`). Tracking those functions allows our analysis to be aware of the location and size of the allocated memory regions. To this end, we intercept calls to those functions and save $(funname \times cs \times params \times retval)$, where $funname$ is the name of the function and cs is the address of the call-site where the function was called from; $params$ are the values of the parameters according to the function's C prototype and $retval$ is the result value that was returned by the function. Note that $funname$ can only be given as a name if the binary contains symbol information. Otherwise, an address has to be used. In the latter case, techniques exist to recover function names from the binary [JRM11], based, e.g., on the signatures of functions.

Once the execution reaching p_s we transfer the concrete state from the instrumented program to the analyzer, in particular:

1. we capture every code and data segment that is resident in memory and store their content, their size and their start address; note that this includes any segment that was loaded at runtime through dynamic linking (e.g. using `dlopen()`) as well as the stack and heap segments;
2. we capture the register file as well as administrative information about, e.g., access rights to memory regions

From p_s onwards, our static analysis tool performs a value analysis and queries the trace at each point that could lead to a control flow change. At such points we record $(p_c \times p_n \times cond)$ where p_c is the current instruction address and p_n is the address of the instruction that will be executed next. $cond$ is the evaluation of the condition for conditional jumps, i.e. $cond \in \{true, false\}$ and denotes if a conditional jump was taken or the fall-through path executed. We track p_n to catch control flow changes that occur at non-jump instructions e.g. arithmetic exceptions. Tracking the condition evaluation $cond$ is necessary for conditional moves because knowing the next address of the execution is not sufficient to infer if the condition was $true$ or $false$ and thus to know whether the move executed. The recorded values are used to guide our static analysis along the path of the actual program execution. This is done by discarding the calculated state on paths that were not taken, as defined by m_k.

During the last phase, the static analyzer performs a reachability analysis where it explores all possible paths starting at user-supplied points $p \in \bar{p}_r$. At this stage the trace is no longer needed.

6.6 Experimental Results

Our implemented static analyzer can readily point out the bug in Fig. 6.1 by e.g. choosing $p = l_5$ to perform the reachability analysis from line 5. It is also able to verify the example when adding the following lines in front of line 5:

```
if (num_elems >= MAX_INT/sizeof(elem_t))
  return NULL;
```

However, it currently lacks the precision to prove the (in-)correctness of the example in Fig. 6.2 due to the lack of abstractions for string functions. Future work should address these drawbacks.

	trace					analysis	
example	#lines	time (ms)	segments (Kb)	trace (Kb)	#insns	time (ms)	memory (Mb)
overflow	73	260	940	1	432	190	44
cast	48	254	940	2	377	50	15
libtiff	28483	1189	3700	132	36842	-	-

Table 6.1: Analyzed Examples.

Table 6.1 shows the measurements of our experiments. The trace column shows: the size of the program in C source lines of code; the time it took to execute the program and

record the trace; the size of the code segments that were captured for the trace; the size of the recorded instruction trace and the number of executed instructions during the tracing. Note that we only record instructions that lead to control flow changes. Hence the recorded instructions trace is much smaller than the number of all executed instructions during the trace. On the right the analysis column displays the time and memory consumption of our static analysis.
The benchmark "overflow" represents the example in Fig. 6.1, "cast" corresponds to Fig. 6.2 and "libtiff" to a C program calling a library with a known vulnerability [Orm06]. Our analyzer is as-of-yet not able to abstractly analyze the trace generated for "libtiff" due to the large number of calls to the operating system that such an off-the-shelf library utilizes and which have to be modeled as built-in primitives in an abstract analyzer. Future work will address this shortcoming. Nevertheless, because of these shortcomings tests on larger examples have not yet been performed.

Future work will also address the automatic choice of k, that is, the length of the recorded trace. Currently, the user does not provide k itself, but an address $p \in P$ at which tracing should stop. We envisage that the recording of the trace is controlled by the analyzer, so that tracing can be stopped when a certain amount of potential vulnerabilities (warnings) have been identified. The user then has to pick interesting warnings in order to undertake a deeper trace analysis.

6.7 Related Work

Dynamic analysis is well-established in tools to find bugs. Indeed, several approaches exist at the source code level, e.g. for Java [SC07]. However, it is at the level of binaries where dynamic analysis seems to be of particular interest due to the difficulty of resolving indirect jumps [KK12].

Besides indirect jumps, a challenge in the analysis of binaries are unpackers and other obfuscating methods. If the obfuscation was intentional it requires bespoke static analyses that do not fall for the traps hidden in the code [Coo+09]. In general, running unpackers before commencing a static analysis of the unpacked code seems to be more practical [DP10]. However, both approaches have their place since a malware might exhibit several behaviors in practice that can be found with static analysis but not dynamically. We chose to involve the user in the analysis process by letting the user select locations $p \in \bar{p}_r$ from which to explore all paths statically. We thus combine the advantages of both, static and dynamic analysis.

Dufur et al. [DRS07] and later Kinder [KK12] use dynamic analysis in order to restrict the states that the static analysis infers. Their approach is a pragmatic one in which certain goals of indirect jumps, memory locations of pointers, etc. are restricted by the values observed by a set of traces. Their static analysis thereby produces useful results when no information on a pointer is statically available but is unsound as certain behaviors are omitted. Our approach differs in that it clarifies which inferred program behaviors are restricted by the trace and which behaviors are based on sound static analysis, thereby retaining the ability to partially verify the program.

Another approach to interactively combining dynamic and static analysis for executable software is to use symbolic execution with the ability to focus onto a concrete trace. In this context, Chipounov et al. [CKC11] propose to record a tree of symbolic executions (one for each path) and to let the user instantiate one of these executions to a concrete trace with which calls to the operating system can be executed. In a sense, they allow their users to switch back and forth between a concrete execution (and hence the ability to execute calls to the operating system) and symbolic execution that treats input generally. Our approach is, so far, only one-way, in that a concrete trace can be executed more generally. However, since symbolic execution has to under-approximate the reachable set of states whenever loops are encountered, their approach cannot provide a way to verify the absence of vulnerabilities but rather serves to find new vulnerabilities.

Using symbolic execution statically has the drawback that it will sooner or later run into scalability problems. This has been addressed by concolic execution [SMA05], a technique that maintains only some input variables as symbolic and executes the program in its concrete state by picking concrete values for the input variables. Thereafter, alternative paths are sought by trying to vary inputs to the symbolically tracked variables. Given such a solution for the symbolic variables, new values for the remaining variables are inferred by running another concrete execution. This method has the merit of being fully automated but has the drawback of not being able to enumerate all paths through the program and, thus, to verify a program.

Jaffar et al. improve the scalability of symbolic execution by merging the state at the loop head using widening [Jaf+12]. A later counterexample-driven refinement phase then tries to recover a post-condition of the loop that is precise enough to verify the program. While this method has similar problems with imprecision as a general static analysis, it would be interesting to combine its refinement phase with our dynamic start approach.

Babić et al. [Bab+11] combine several traces to recover the control-flow graph (similar to [KK12]) before applying a whole-program static analysis. The idea is to use the control flow graph to perform a static pre-analysis that finds potential vulnerabilities and to then use symbolic execution to find an actual trace to a bug. Our work differs in that we never perform a static analysis of the whole program but only on those parts that are relevant to an attack.

In terms of static analysis, Wagner was the first to address the challenge of finding buffer overflows using numeric analysis [Wag00]. His approach is rather imprecise since he translates programs to purely numeric programs, thereby losing the ability to argue about the targets and offsets of pointers.

With respect to generalizing the inputs of a program, a common approach is to perform a taint analysis that tracks which paths the input to a program can take [SM03; TA05]. Our approach generalizes this idea in that it not only tracks where inputs to the program can flow to but also what these inputs are. In particular, it improves over a mere taint analysis in that its $\bar{p}_r$ set gives the user a choice of paths to which input will flow but which is neglected by the trace. Once a possible security vulnerability is found, a taint analysis can be useful to infer which input source is responsible for making this vulnerability reachable. One drawback of our current analysis is that the information on where a certain input was

read is lost. However, it would be easy to enhance our analysis to also track the possible sources of an input, thereby combining standard reachability and taint analysis. Future work will address this enhancement.

6.8 Conclusion

We proposed a combination of static and dynamic analysis through which the user can interactively explore the program behavior. It combines the advantages of static analyses (program verification) with that of dynamic analyses (scalability), thus allowing the user to focus on the relevant program parts.

Our infrastructure records only little data off the program trace, thereby allowing for good scalability. Moreover, the user is in control of the analysis cost by interactively increasing the coverage of the analyzed code. After all, our analysis is easy to use as the user only has to provide points p_s where the program reads inputs which can be found using e.g. Unix' `strace -i`. The choice of points $\bar{p}_r$ from where to start a reachability analysis can be performed iteratively by moving backwards from the end of the trace. As this can be automated future work should look into finding good heuristics, e.g. loop heads, to let the analyzer iteratively choose $\bar{p}_r$.

Another application of trace abstraction is to use the trace to debug the analyzer. As, not surprisingly, comparing the static analysis results to the trace can help a lot to discover bugs in the analyzer. During our experiments we were able to discover and fix a couple of bugs in our numeric domains.

Part IV

Implementation – The Bindead Analyzer

7 Implementation Details

The following chapter presents the implementation and features of the analyzer that have not been mentioned in previous chapters.

7.1 Front-ends

We will begin with the input interface of the analyzer.

7.1.1 Binary Format Parsers

Our analyzer accepts either raw data in form of an array of bytes or executable files. For the latter we are to parse ELF files [Too95] (commonly used in Linux and Unix systems) and PE files [Mic99] (used in Windows). The executable parser extracts the metadata, such as architecture, procedure names (symbols), exported and imported procedures and the code and data sections. We use a common abstraction for executable formats, thus adding support for the Mach-O format used on Apple computers would only require to parse and map the various sections of the format to our abstraction.

The analyzer uses the metadata and data sections from the binary to initialize the entry state of the analysis and choose the disassembler front-end and platform model. The disassembler front-end is used to translate bytes from the code section to RREIL instructions.

7.1.2 Disassembler Front-ends

We provide a set of disassemblers, mainly for the Intel x86-32 and x86-64 and for the AVR embedded platform. These disassemblers are hand-coded in Java, implementing the parsing and translation of instruction semantics to RREIL [SMS11]. During a Dagstuhl seminar on Binary Analysis, the consensus was to unify the mostly hand written disassemblers in binary analysis tools into a common framework to share implementation costs. This project has since been carried out and culminated in a disassembler framework called GDSL [SKS12]. We implemented the interface to the GDSL framework and are able to use it as disassembler front-end. However, as GDSL requires native code libraries we still keep the legacy Java front-end as a fall-back. The latest GDSL version [KSS13b] enriched the RREIL grammar to include while-loops and if-then-else branches thus producing more compact code. It also includes a series of optimizations, such as liveness and forward-expression

substitution. Both improve the scalability of the static analysis as fewer instructions need to be analyzed. These grammar extensions, however, have not yet been implemented in the analyzer but are planned for future work.

7.1.3 Assembler for RREIL

In order to be able to unit test our analyzer without resorting to compiling C code that in turn is disassembled we implemented an assembler for the RREIL grammar. This way it is possible to specifically design small test cases without the necessity to configure the C compiler to arrive at the desired translation of the high level languages in order to avoid the issues with changing compilation patterns in different compiler versions, we first used native assembler code that is compiled to an executable. However, we realized that for fast prototyping of test cases and platform independent assembler code a direct parser for the RREIL grammar is the best choice.
Another benefit is that we can modify code from disassembled executables to observe the effect of alternative translations as the disassembler front-ends produce RREIL assembler code.

7.2 Analyzer

The analyzer is implemented in Java as it seemed a good choice for a typesafe and performant language at the inception of the project. Java is platform independent, well supported, widely deployed and fast enough for our analyses. Early attempts in Scala were given up as the language and tool support seemed unstable at that time. However, Scala allows for more concise code and has better support for immutable data structure and the functional programming style. As it runs on the same platform as Java, the JVM, future features of the analyzer should re-evaluate Scala or other languages targeting the JVM.
The static analyzer together with the abstract domains is open source and available online at [Mih14a].

7.2.1 Fixpoint

Figure 2.10 shows that the fixpoint engine is the driver of the analyzer. On the one hand it drives the disassembling of instructions and the construction of the CFG. On the other hand it evaluates the abstract semantics of each instruction to compute the fixpoint for a given program. Note that due to performing a reachability analysis from a given start point, we do not consider dead-code in our analysis, that is, paths that are unreachable from the initial state.
We implemented the fixpoint computation as described in Fig. 2.8 using a map that tracks and updates the state per program point. The resulting CFG is only used for debugging, that is, to display and inspect the result, e.g. using the Visualization described in Sect. 8. A more interesting aspect is the implementation of the worklist W for which we tried several strategies in order to improve the efficiency of the fixpoint iteration before settling on the heuristic described next.

Improving the Evaluation Order The algorithm in Fig. 2.8 iteratively computes the fixpoint of the abstract interpretation of a given program. It does this by using chaotic iteration [Bou93], that is, it randomly picks a program point from the worklist W in line 4 and evaluates the abstract transformer for the instruction at this program point. This is inefficient as it does not take the program flow into account and thus ignores the dependencies between the constraints. It can be improved by using a stack instead of a worklist, which results in a depth-first traversal of the program during the fixpoint computation. However, this is still inefficient as can be seen when considering the program shown in Fig. 2.4. If a loop exists in the program a depth-first strategy will iterate the loop once and then continue with evaluating the program points after the loop. Only then will it re-iterate the loop again as the state in the loop is not yet stable. This in turn triggers another round of evaluations for the code following the loop. Hence, each loop in the program will result in redundant iterations for the program points that depend on the loop state.
A better iteration strategy would try to infer a fixpoint for a program point before considering its successors, that is, each loop is iterated until it is stable before considering other program points. In machine code analysis the structure of the CFG is not known up-front but the CFG is inferred during the reachability analysis. Hence, we cannot use topological sorting methods as described in [Bou93]. However, a good approximation for locality and strongly connected components (SCCs), is to use the code layout, that is, to use the addresses of instructions. We can define a total order on the instructions given their addresses. Using this order to implement a priority worklist W we are able to improve the efficiency of the fixpoint algorithm by requiring fewer evaluation steps before reaching the fixpoint. This heuristic also works well with **`if-else`**-branches in the code, for which depth-first iteration would consider only one path and continue with the analysis of the code following the branch.
In the end, this heuristic to use addresses works quite well for compiler generated code but may be insufficient for obfuscated code that reorders instructions and inserts spurious control flow. However, on such code we degrade to depth-first search at worst.

7.2.2 Warnings

To inform about certain conditions or assumptions during an analysis we emit a set of warnings. A warning is associated with the program location at which it was emitted. We distinguish between three warning classes:

informational messages exist mainly to inform about precision loss during the analysis. For example, when reading a range of bytes from memory that need to be approximated a message will be emitted. The user can later find out where the analysis lost precision and thus easier decide if more severe warnings are false negatives.

warning messages are more severe than informational messages as they denote rare program behavior, such as unaligned memory accesses or overflow/wraparound of variables. These might, however, be intended program semantics, thus continuing the analysis is sound as program execution is not aborted by such unexpected behavior.

state restrictions affect the soundness of an analysis. When the analysis discovers an error state such as a buffer overflow or dereference of a `NULL` pointer it raises a warning and should terminate. However, sometimes it is still desirable to continue the analysis by propagating only the non-erroneous state as the error condition might be a false negative. Doing so is unsound, hence it is necessary to raise a warning and state that the analysis results are only valid under the assumption that the warning was spurious. This feature is very powerful as it allows to continue the analysis past error conditions. The user then must decide if the warning is a false negative.

Note however that some warnings such as non-aligned accesses or wraparounds can also be seen as an error state and thus be treated as "state restrictions". Choosing the severity of some warnings is thus left for the user.
The warnings mechanism is implemented using a channel for abstract domains to emit messages on. These messages are tracked and accumulated for each program point. As an invariant, the number of warnings for a program point must never become less during the analysis. This is guaranteed by the fixpoint iteration as the state at each program point is only growing and thus warnings can not be removed during later iterations.

7.2.3 Primitive Operations

In order to implement special operations that are not easily expressible as a series of transfer functions, we provide a domain interface for so-called "primitives". These operations are of the form $primop\ (lval)+, (rval)+$, that is, a primitive is identified by its name and its output *lval* and input parameters *rval* (see Sect. 2.4.6). Primitives are broadcasted to the segment domains where each domain can implement its own semantics for a primitive. If a primitive broadcast is not handled by any of the domains we emit a warning. Currently primitives exist for `malloc` and `free` which are handled by the heap domain. Additional primitives for e.g. cryptographic operations that are now present in modern CPUs are generated by the disassembler front-end. However, no domain to handle such primitives exists yet.

7.2.4 Hooks for Procedures and Syscalls

Sometimes it is desirable to analyze only a subset of the whole program and skip complex code parts for which an approximation of the effect cab be succinctly written in RREIL. In order to achieve this, the analyzer provides a mechanism to add hooks for any program point with RREIL code that is executed instead of the code in the binary. Hooks are triggered whenever control flow reaches a given program address. For example if the binary provides symbols it is easy to replace the code of `malloc` and `free` with shorter and simpler versions that simply evaluate the primitive. Especially for the heap summarization domain it is necessary to recognize calls to `malloc` and translate them to code that manipulates the internal heap representation of the domain using primitives.

Furthermore, since hooks are snippets of RREIL code it allows us to replace any code parts that are not present in the binary, e.g. system calls, with stubs that over-approximate the effect of the missing code. We therefore provide a mechanism that infers the set of invoked system calls and uses the right stub code to simulate the effect of the system call on the current state. Furthermore, hooks are useful when dealing with IO code or initialization code that interacts with the environment and to translate known library functions such as cryptographic operations to primitives for a specialized abstract domain.

7.2.5 Interoperation with other Analyzers

It is desirable to share results with other analyzers if an equivalence mapping of the analyzed code is feasible [Riv03]. We currently express analysis results through *assertions*. These can be inserted at each program point and express range, equality or congruence constraints. To import results from other analyzers or user provided hints we use *assumptions*, that refine the range or equality relations between variables. Currently, however both *assertions* and *assumptions* are only present as RREIL statements. It remains for future work to add these as statements in higher languages such as C and use procedure hooks to translate these to RREIL statements during the disassembling process. Additionally, we want to share results with other analyzers, thus, adopting a more expressive specification language such as the one used in e.g. Frama-C [Kir+12] is a future goal.

7.2.6 Parallelization of Analyses

The analyzer was not designed with parallelization in mind. However, as most data structures are immutable (see Sect. 7.3.2) and an analysis does not make use of global and shared state it is possible to run several analyses in parallel. However, it is desirable to be able to parallelize smaller parts of the analysis algorithm to make use of all available computation resources while performing a single analysis. Future work should look into the possibility to perform parts of the fixpoint computation in parallel.

7.2.7 Tracing Programs for Dynamically Started Analyses

The tracing tool is separate from the analyzer and implemented in C++ using the PIN tracing framework [Luk+05]. The tool is available open source at [Mih14b]. It can be configured to track and store different program properties as described in Sect. 6.5. The analyzer takes the trace dump produced by the tool as input to perform the trace abstraction and dynamically started analysis described in Chap. 6.

7.3 Abstract Domains

We will next describe the implementation details of our abstract domains and the interfaces between domains.

7.3.1 Domain Interfaces

Our cofibered domains or functor domains are implemented in Java using abstract classes and Java generics. The latter are used to specify the child domain. We track a pair of local state and child domain per functor class. Transfer functions modify the local state and are forwarded to the child domain after which a new domain instance is built. The domain functor classes implement the lattice interfaces and interfaces for the intermediate language of the current tier *L(x)* with *x* one of *rreil*, *memory*, *finite*, *zeno*. Note that our domains are actually only join semi-lattices as we do not require a meet operation. With this design we are able to implement most of the domain interface in the abstract classes thus the concrete domain only needs to implement behavior specific to its semantics.
Our domains are implemented as immutable classes, where each transfer or lattice operation returns a new instance of the domain. This is to avoid unintended sharing of data and thus subtle and hard to find bugs. In order to not incur a high memory cost due to the immutable domains, we use persistent data structures as described in the next Section 7.3.2. We use a builder pattern during transfer functions when domains need to perform modifications to their data structures and collect mutations to the child, so-called child-ops, in a queue. The idea is to apply these operations in one batch and even optimize the list of child-ops using peep-hole optimizations, e.g. a sequence of an **`intro`** and **`drop`** operation can be removed.

7.3.2 Data Structures

During the early development we decided that it is a must to use immutable data structures in our abstract domains. Earlier experiences showed that using mutable data structures leads to defensive copying to avoid bugs from sharing, thus losing any performance benefits of mutable data structures over immutable data structures. To improve the memory footprint of the immutable data structures we use ordered balanced binary trees (AVL trees) that allow sharing of identical subtrees. The implementation is described in [Ada93] which is also used for the map implementation of the Haskell language. Using the AVL trees we implemented immutable map and set data structures that inherit the internal subtree sharing feature. Maps are a crucial data structure for abstract domains as a domain often maps the support set of variables it tracks to values. Besides the common set operations such as union and intersection, which are very fast due to the ordered trees, one of the most used feature is the 3-way split between two maps. A 3-way split returns the key-value pairs that are only present in the first argument or the second argument and additionally returns the common keys that are mapped to different values. Our abstract domains use the split to implement the join and subset operations, which often allows us to perform these transfer functions with a sub-linear complexity as complete subtrees that are identical are not traversed. Our design and motivation for the data structures is here similar to the ones used in Astrée [Bla+02].

Analogously to the maps above, the memory domain that tracks fields in memory uses an interval tree data structure, implemented as a balanced binary search tree. As the inferred

fields in memory may overlap, the data structure must provide fast operations for the manipulation of fields and the search for overlappings.

7.3.3 Channels

Channels between domains are necessary for two reasons. Firstly, we want to extract information from the domain state to e.g. determine the targets of an indirect jump. Secondly, by communicating information between domains we can perform reduction. Reduction is a refinement of the state tracked by one domain using information from another domain. In a modular domain design each domain tracks some properties of the program thus a "smart" combination of this properties is required (see Sect. 2.6). Reduction from a domain to its child is performed using transfer functions thus no special channel is needed. For the reduction from children to parents we implemented two channels.

Query Channel

The so-called $queryChannel$ is used to query information from child domains. Any domain can answer a query or pass it on to its child and refine the answer from the child with its own information. The channel currently allows to ask for range information for a variable through $queryRange : \mathcal{X}_M, sz, o \to Val$ and $queryRange : \mathcal{X}_V \to Val$. The former returns values for memory regions $\mathcal{X}_M$ or registers, given a size sz and offset o into these regions and is used at the top level to resolve jump targets. The latter is used for finite and numeric domains to retrieve valuations for domain internal variables. Note that the *fields* domain translates a memory regions to a set of internal variables or fields, so that the first query function only performs the resolution of variables and eventually calls the second query function on the child domain. The returned valuation depends on the precision of the numeric domains that are used and is either an interval with congruence information $Val : \mathcal{I} \times \mathcal{CR}$ or a set of intervals with congruence $Val : \mathcal{IS} \times \mathcal{CR}$. The result is abstracted under a common interface in Val.
Note that querying the valuation of a variable returns an over-approximation of the values. Thus to find out if a variable x may contain a certain value c it is more precise to apply the test $[\![x \neq c]\!]s = \bot$ on the domain state s than using $queryRange$. Each domain implements a semantic for tests thus using all the information tracked by the domain to evaluate a test. This allows to answer test queries more precisely whereas a range valuation is the set approximation of all values. However, a test is often a complex domain operation whereas $queryRange$ is a cheap operation.

Further information provided by the $queryChannel$ is a set of equalities between variables. Specifically $queryEqualities : \mathcal{X}_V \to \wp(Eq)$ returns a set of all the known affine equalities for a variable $x \in \mathcal{X}_V$. An equality is of the form $Eq : \sum_i a_i x_i = c$ with $x = x_i$ for some i. Equalities are queried in e.g. the *predicate* and *threshold* domain which both track predicates $p \in \mathcal{P}$ over variables. Whenever a variable $x \in vars(p)$ is overwritten by an assignment $x = c$ these domains try to find a substitution $\sigma = [x/eq]$ for x with $eq \in queryEqualities(x)$ in order to keep the predicate as $p' = \sigma(p)$.

As an extension to this use case, future work should implement a separate query for substitutions. The benefit is that any relational domain is able to provide a substitution for a predicate, whereas currently we are limited to equalities only. For example, given the predicate $x > z$ and the polyhedra domain tracking $x < y$. Asking for a substitute for x in the predicate the polyhedra domain would return $y > z$.

Synth Channel

The so-called $synthChannel$ is a second channel going from child domains to parent domains. While the $queryChannel$ provides information about the state of child domains the $synthChannel$ is used to retrieve changes in the child due to transfer functions. Some examples for this feature are given in Sect. 2.6.5. The channel is implemented using push semantics, that is, a child domain pushes new facts to the channel at the end of a transfer function. Parent domains then may use this information to reduce their own state or initiate new transfer functions.

Due to the cofibered design the support sets between domains do not necessarily match (see Sect. 2.5). Thus a domain must ensure to not leak variables that are local to the domain when communicating with other domains through channels. In general a domain that requires fresh variables introduces these variables also in its child domain. This allows to use the numeric and relational information stored in child domains.

Debug Channel

As the name suggests the debug channel is used to expose specific internals of domains. The interface is intended for development and thus not fixed. Analyses must not depend on information in this channel. The channel is designed to encapsulate information providers, such as abstract domains. Hence, it does not require each domain to implement specific debug interfaces. Each domain, however, may require extending the debug channel with further queries that are special to that particular domain.

8 | Visualizing Analysis Results

content/implementation/resources/p9-screenshot.png

Figure 8.1: GUI displaying the results of an analysis.

During the development of the framework and particularly the abstract domains it became clear that it is cumbersome to debug a fixpoint computation on a large set of interdependent values. Furthermore, bugs in the implementation of abstract domains are difficult to spot as they may only affect the precision of the result. Finding the places where the precision loss happened and tracking it back to errors in the implementation is a very time consuming task, where most of the time is wasted on manually analyzing the trace of the analysis output. Especially the latter can be improved upon by providing a more compact representation of the analysis results and methods to filter and search through the results. We thus decided to implement a GUI on top of the static analyzer framework that would let us experiment with novel ideas on how to debug a static analysis.

Figure 8.1 shows the GUI for the analyzer displaying the analysis of the example from Sect. 9. The GUI displays the CFG and call-graph inferred by an analysis along with the warnings that were emitted. Warnings are marked in the CFG with a red background to highlight the location of occurrence. The CFG display also allows to highlight and search parts of the code. Moreover it is possible to perform a backwards slicing given a selected register, thereby narrowing down the shown instructions to the one that contributed to the computation of the value of the register. This simplifies the search for the cause of a precision loss.
On the right the GUI displays the complete abstract domain state for a selected instruction. As the amount of information tracked by an abstract state can become big, future work should investigate how to conveniently display and filter the state for certain bits of information.

The GUI for the analyzer is open source and available at [Mih14c].

Part V

Applications and Conclusion

9 Case Study: Sendmail Crackaddr Vulnerability

We will now present the application of our analyzer to a particularly hard example. It shows that our analyzer finds a known vulnerability in the Sendmail program and at the same time is able to prove the fixed version as correct.

9.1 Problem Statement

The Sendmail program in versions 5.79 to 8.12.7 contained a vulnerability [Dow03; IBM03] that was discovered in 2003. Shortly thereafter the vulnerability was shown to be exploitable for remote code execution [LSD03]. What makes this vulnerability particularly interesting is that it has been used as an example for a difficult to analyze software bug using static analysis tools. As a consequence the vulnerability has been discussed in conference talks [Dul11; VL12] and publications [VHR12]. The existing automatic static analysis tools suffered from imprecision or could not infer the invariant, hence the consensus was that the example requires manual user hints to be proven correct [Hee11].

What makes the analysis of the code so difficult is that the code implements a parsing algorithm for email addresses which also tries to sanitize parsing mistakes. Parser code in general is considered a recurrent source for program errors that might lead to exploitable vulnerabilities [Bra+11]. It is difficult to analyze as parsers are implemented as state machines, that is, an analysis must express a relation between the state machine and other program variables. Moreover, the input is read using a loop that reads input and initiates the transitions between states.

Figure 9.1 shows the code of a simplified version of the vulnerability that maintains the challenging characteristics of the original example. The code implements a parsing routine that parses email addresses and copies the result to a buffer `localbuf` allocated on the stack with a fixed size of 200. It therefore reads a string of an arbitrary length (line 10) character by character. It copies each character to `localbuf` and tracks the parsing state using two flags: `quotation` and `roundquote`. To ensure that the buffer `localbuf` is not written past its maximum length (the input string may be longer), the state machine inside the loop increments (lines 17, 26) and decrements (lines 13, 22) a variable `upperlimit` which takes care that there is enough space in `localbuf` to write characters to it. The copy operation is performed in line 31 which is guarded by the test on `upperlimit` in the

```
#define BUFFERSIZE 200
#define TRUE 1
#define FALSE 0
int copy_it (char *input, unsigned int length) {
    char c, localbuf[BUFFERSIZE];
    unsigned int upperlimit = BUFFERSIZE - 10;
    unsigned int quotation = roundquote = FALSE;
    unsigned int inputIndex = outputIndex = 0;
    while (inputIndex < length) {
        c = input[inputIndex++];
        if ((c == '<') && (!quotation)) {
            quotation = TRUE;
            upperlimit--;
        }
        if ((c == '>') && (quotation)) {
            quotation = FALSE;
            upperlimit++;
        }
        if ((c == '(') && (!quotation) && !roundquote) {
            roundquote = TRUE;
            // decrementation was missing in vulnerable version
            upperlimit--;
        }
        if ((c == ')') && (!quotation) && roundquote) {
            roundquote = FALSE;
            upperlimit++;
        }
        // if there is sufficient space in the buffer,
        // copy the character
        if (outputIndex < upperlimit) {
            localbuf[outputIndex] = c;
            outputIndex++;
        }
    }
    if (roundquote) { // close a leftover open brace
        localbuf[outputIndex] = ')';
        outputIndex++;
    }
    if (quotation) { // close a leftover open bracket
        localbuf[outputIndex] = '>';
        outputIndex++;
    }
}
```

Figure 9.1: Simplified address parsing algorithm in Sendmail.

line before. Furthermore, after the loop is exited, lines 36 and 40 write a closing brace or bracket that was left unmatched to `localbuf`.

The vulnerable version of the code was missing the decrementation in line 22, thereby allowing a specially crafted input string to push the value of `upperlimit` beyond the size of `localbuf` and writing to the stack past `localbuf`. This allowed exploiting [LSD03] a system on which the vulnerable Sendmail versions were running by sending it an email with a specially crafted email address.
In Fig. 9.2 we show the state machines that are implemented by the vulnerable and non-vulnerable version of the code. The automaton in the vulnerable version shown in a) can be traversed between the nodes $!q!r$ and $!qr$ multiple times, thereby incrementing `upperlimit` arbitrarily often. The correct version shown in b) does not exhibit this behavior, therefore a write past the end of the buffer `localbuf` cannot be achieved in this version.

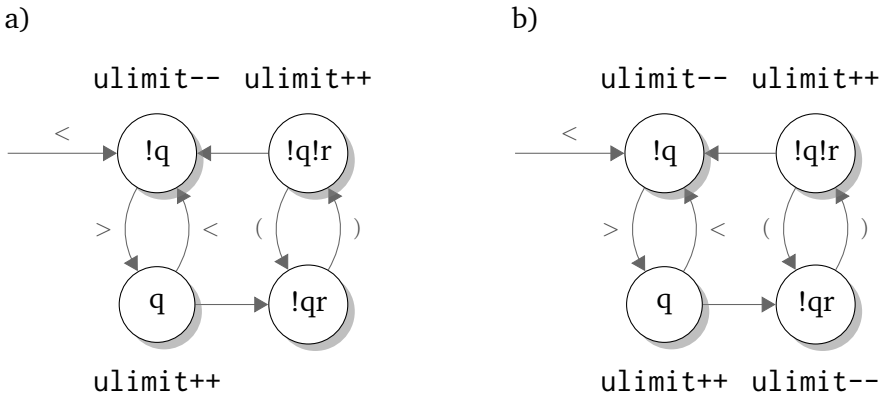


Figure 9.2: Parser automatons for code in Fig 9.1. Version a) contains the vulnerability and b) the corrected version

Analyzing the example is challenging on the one hand as verifiers based on the exploration of concrete states may run into an explosion of the states that need to be tracked [WKC13]. On the other hand, abstract interpretation based analyses lose precision due the join and widening and cannot separate the states in the state machine. Hence, the analysis flags both versions of the code, the vulnerable and the non-vulnerable as erroneous. Such an analysis warns about a buffer overflow and the overwriting of the return address but is not able to prove the correctness of the non-vulnerable version. Furthermore, code analyzers based on decision procedures require the manual specification of the loop invariant after which they are able to prove the correctness of the code [VHR12] by induction.

We will illustrate next that our abstract interpretation based analysis is precise enough to distinguish the correct from the vulnerable version.

9.2 Analysis

The challenges of a static analysis to show the correctness of the program in Fig. 9.3 are as follows: it must be shown that the variable `outputIndex` is bounded and smaller than the size of `localbuf`, that is given by `BUFFERSIZE = 200`; therefore it must be shown that $outputIndex < 200$ holds in lines 31, 36 and 40. Since the length of the input string, which is also the loop bound, is bounded by $length$ which we have to assume to be in $length \in [-\infty, +\infty]$. We cannot prove that the buffer `input` is accessed within bounds as we do not know the relation between the buffer and the variable `length`. We can however, analyze the accesses to `localbuf`. Here, the test at line 30, $outputIndex < upperlimit$ is used as widening threshold to bound the extrapolation of $outputIndex$ on widening. This, in turn, requires to show that the values of $upperlimit$ are smaller than `localbuf`, namely that $upperlimit < 200$ holds inside the loop.

step	line		intervals				affine
			o	u	q	r	
1	9		$[0,0]$	$[190,190]$	$[0,0]$	$[0,0]$	$u=190, q=0, r=0, o=0$
2	14		$[0,0]$	$[189,189]$	$[1,1]$	$[0,0]$	$u=189, q=1, r=0, o=0$
3	15	$\sqcup$	$[0,0]$	$[189,190]$	$[0,1]$	$[0,0]$	$u+q=190, r=0, o=0$
4	16		$[0,0]$	$[\mathbf{189},\mathbf{189}]$	$[1,1]$	$[0,0]$	$u+q=190, r=0, o=0$
5	18		$[0,0]$	$[190,190]$	$[0,0]$	$[0,0]$	$u=190, q=0, r=0, o=0$
6	19	$\sqcup$	$[0,0]$	$[189,190]$	$[0,1]$	$[0,0]$	$u+q=190, r=0, o=0$
7	20		$[0,0]$	$[\mathbf{190},\mathbf{190}]$	$[0,0]$	$[0,0]$	$u+q=190, r=0, o=0$
8	23		$[0,0]$	$[189,189]$	$[0,0]$	$[1,1]$	$u+q+1=190, r=1, o=0$
9	24	$\sqcup$	$[0,0]$	$[189,190]$	$[0,1]$	$[0,1]$	$u+q+r=190, o=0$
10	25		$[0,0]$	$[\mathbf{189},\mathbf{189}]$	$[0,0]$	$[1,1]$	$u+q+r=190, o=0$
11	27		$[0,0]$	$[190,190]$	$[0,0]$	$[0,0]$	$u+q=190, r=0, o=0$
12	28	$\sqcup$	$[0,0]$	$[189,190]$	$[0,1]$	$[0,1]$	$u+q+r=190, o=0$
13	33		$[1,1]$	$[189,190]$	$[0,1]$	$[0,1]$	$u+q+r=190, o=1$
14	34	$\sqcup$	$[0,1]$	$[189,190]$	$[0,1]$	$[0,1]$	$u+q+r=190$
15	9	$\sqcup$	$[0,1]$	$[189,190]$	$[0,1]$	$[0,1]$	$u+q+r=190$
16	12		$[0,1]$	$[\mathbf{189},\mathbf{190}]$	$[0,0]$	$[0,1]$	$u+q+r=190$
17	15	$\sqcup$	$[0,1]$	$[\mathbf{188},\mathbf{190}]$	$[0,1]$	$[0,1]$	$u+q+r=190$
18	20		$[0,1]$	$[\mathbf{190},\mathbf{190}]$	$[0,0]$	$[0,0]$	$u+q+r=190$
19	24	$\sqcup$	$[0,1]$	$[188,190]$	$[0,1]$	$[0,1]$	$u+q+r=190$
20	34	$\sqcup$	$[0,2]$	$[188,190]$	$[0,1]$	$[0,1]$	$u+q+r=190$
21	9	$\sqcup$	$[0,2]$	$[188,190]$	$[0,1]$	$[0,1]$	$u+q+r=190$
21'	9'	∇	$[0,190]$	$[188,190]$	$[0,1]$	$[0,1]$	$u+q+r=190$
22	31		$[\mathbf{0},\mathbf{189}]$	$[188,190]$	$[0,1]$	$[0,1]$	$u+q+r=190$
23	34	$\sqcup$	$[\mathbf{0},\mathbf{190}]$	$[188,190]$	$[0,1]$	$[0,1]$	$u+q+r=190$
24	9	$\sqsubseteq$	$[0,190]$	$[188,190]$	$[0,1]$	$[0,1]$	$u+q+r=190$

Figure 9.3: Analysis of the Sendmail code shown in Fig 9.1.

The table in Fig. 9.3 shows the analysis of the code in Fig. 9.1 using the domains stack $\mathcal{WP} \triangleright \mathcal{WD} \triangleright \mathcal{T} \triangleright \mathcal{A} \triangleright \mathcal{I}$. The domains $\mathcal{WP}$, $\mathcal{WD}$ and $\mathcal{T}$ denote our widening domains from Chapter 3 namely widening points inference, widening delay after constant assignments and widening thresholds. The remaining domains $\mathcal{A}$, $\mathcal{I}$ are the *affine* domain and the *interval* domain respectively. The analysis is performed on the non-vulnerable version,

proving that the fixed code does not exhibit a buffer overflow. For brevity, we skip most of the analysis steps that are irrelevant for the illustration of the analysis. Furthermore, we abbreviate the variables `outputIndex` and `upperlimit` to o and u. Analogously, the flag variables `quotation` and `roundquote` are abbreviated as q and r. The values of the remaining variables of the program are omitted due to lack of space. For the same reason, we omit the states for the widening domains. Note that the parameters to the function are tracked as $length \in [-\infty, +\infty]$ and $input \in [-\infty, +\infty]$, that is, the read access through the pointer `input` (line 10) will result in the value $\top$ being read, in particular $c \in [-\infty, +\infty]$. This means that the branch conditions performing a comparison of characters in lines 11, 15, 19, 24 result in both the **`then`** and the **`else`**-branches being considered. However, the branches also depend on the valuation of q and r at these program points.

After the variable initializations in step 1 the loop is analyzed and we consider each branch of the state machine in turn. In step 3 we infer an affine relation between u and the flag q which is used in the next step to refine the value tracked for u in the intervals. In the following steps each branch increments or decrements the value tracked for u and joins the result with the value of u propagated on the **`else`**-branch. The join infers affine equalities between the flags and variable u which allows us to apply reduction between affine and intervals and thereby track a precise value range for u.
In step 15 we have iterated the loop once and reanalyze the loop head. Widening is suppressed due to our $\mathcal{WD}$ domain, that tracks newly seen constant assignments from lines 12, 16, 20, 25, where the boolean flags are set. As no widening is performed we maintain the precise values for the flags and u. Next, the loop is analyzed again with a slightly modified state. Step 9 inferred the equality $u + q + r = 190$ which relates both boolean flags and u. As the guard in the first **`if`**-branch only restricts one flag q, with the other flag being $r \in [0, 1]$ the best refinement of the value of u is $u \in [189, 190]$. Executing the decrementation in line 13 and the join in step 17 leads to $u \in [188, 190]$ which describes a larger value range than in the previous iteration in step 3. The reduction in step 18 however restricts the value of both flags q and r, hence given the affine equality we are able to reduce the values of u to $u = 190$ inside the **`if`**-branch.
After incrementing o a second time in step 20 we reach the loop head again and perform widening in step 21'. Widening applies the threshold $o < u$ from line 30 that has been transformed in line 32 to $o - 1 < u$ and thereby we maintain a precise upper bound on o, namely $o \in [0, 190]$. The values of the flags q and r have not changed w.r.t the previous loop iteration, hence widening is a no-op for these variables. Note that due to the equality $u + q + r = 190$ we are able to refine the lower bound of variable u after widening would extrapolate this bound to $-\infty$. The loop is analyzed a last time in which the values of the variables do not change and we reached a stable state at the loop head. Note that due to the guard in line 30 the value of o is not incremented further than the upper bound inferred by widening, leading to a stable state at the loop head. Moreover this proves that the write in line 32 is not out of the bounds of `localbuf`.
In the last steps (not shown in the table) the analysis propagates $o \in [0, 190]$ outside of the loop and is able to show that the writes in lines 36 and 40 are safe.

In order to prove this example correct we employed a set of different domains, where each served a different purpose. Widening with thresholds is required to infer a precise upper bound for o inside and outside of the loop. Delaying widening until the values of the flag variables are stable maintains the precision and allows us to infer an affine equality $u + q + r = 190$ between the flags and u. The affine equality and the reduction with the interval domain is necessary to maintain precise values for u. The values for u are then used to restrict the upper bound of o during widening. Finally, inferring a precise value for o allows us to prove that the memory accesses are correct in this program.

The difference when analyzing the vulnerable version of the program is that due to the missing decrementation in line 22 we do not infer an equality between u and the flag variables. Hence, the values of u are not refined on entering the **`if`**-branches of the state automaton. A steady incrementation and decrementation then leads to widening extrapolating the values of u to $u \in [-\infty, +\infty]$. Although, we track and apply the widening threshold $o - 1 < u$ from line 30 it cannot restrict the upper bound of o during widening due to the precision loss for u. The analysis of the vulnerable example thus emits warnings for the writes to the buffer `localbuf`. It further emits a warning that the return address might be overwritten as the writes to `localbuf` are unbounded and might overwrite an arbitrary amount of memory cells on the stack.

10 | Conclusion

In this thesis we have presented a modular framework for the static analysis of executable programs. The framework performs the disassembling, the reconstruction of the control flow graph and analyzes the program to infer invariants for each program point. Furthermore, the framework implements a sound analysis able to prove the correctness of memory accesses. It improves upon previous work in that it provides different interfaces for inferring structural, bit-level and numeric information, thereby allowing the design of simple abstract domains at each level that are nevertheless able to communicate across these interfaces. Moreover, new abstract domains can be added to improve the precision of the analysis. This modularity was demonstrated by presenting several novel domains that implement widening heuristics, a domain for predicate abstraction and a domain that improves the analysis precision in case of uninitialized memory locations. Finally, we presented an extension to the framework that allows for combining the static analyzer with dynamic analyses.
We illustrated the practicality of our analysis by showing applications to challenging examples for which we discussed in detail the solutions inferred by our analyzer.

The implementation of the framework is available online at [Mih14a].

10.1 Contributions

In summary, the contributions to the field of static analysis of binaries are as follows:

- a platform independent intermediate language (RREIL) designed for simplicity and conciseness and full value semantics; the language abstracts the semantics of different platforms into a common representation

- an adaptable and sound static analysis framework for the analysis of machine code; a stack of numeric and symbolic domains designed to improve the analysis precision while maintaining scalability; configurable as each domain is defined in terms of a simple interface that allows adding domains as plugins

- a novel approach to implement widening heuristics as abstract domains, that is, without requiring modifications to the fixpoint engine; this approach allows testing

new widening heuristics and trading the precision of widening for scalability by adding or removing abstract domains for widening

- a novel abstract domain that tracks implications between predicates describing the precision losses in convex numeric domains; using modus ponens, the domain infers predicates that are applied to reduce the numeric child state, thereby recovering precision losses due to convex approximations
- an abstract domain that tracks initialized and uninitialized variables and improves precision whenever a state joins the values of such variables
- the combination of dynamic trace analysis with static analysis
- a user interface for the analyzer that improves the user interpretation of analysis results and thereby also improves the debugging and refining of analyses

10.2 Future Work

As mentioned in Chapter 2.5 the implementation of some domains in our hierarchy is not finished. Hence, future work shall address the implementation of the memory domains that summarize strings and arrays and should consider tracking the content of code segments as part of an abstract domain, thereby allowing the analysis of self-modifying code using similar techniques as in [Kin12].

As our domains stack is configurable, it is possible to integrate similar ideas as in [FL11] to improve the scalability of our analyzer. Fähndrich et. al commence an analysis with cheap abstract domains, thereby achieving a fast analysis. If a desired invariant cannot be proven they restart the analysis with more expensive domains. A similar iterative approach can easily be integrated in our framework. Furthermore, when using expensive relational domains, such as polyhedra future work will implement similar methods as in [Bla+03b; VB04], that splits the support-set of relational domains to improve the scalability.

We plan to fully integrate the GDSL disassembler frontend [KSS13a] with our analyzer and extend the RREIL language with concurrency primitives and the handling of endiannes.

Our approach to combining dynamic analysis with static analysis is still a proof of concept, that requires further work. Further afield is a backwards analysis that may infer a counterexample [GR06b] or a program input state which confirms the error given a warning emitted by the static analysis.

List of Figures

List of Tables

List of Code

Bibliography

[Ada93] S. Adams. "Implementing sets efficiently in a functional language." In: *Journal of Functional Programming* 3.04 (Nov. 1993), pp. 553–561. ISSN: 0956-7968. DOI: 10.1017/S0956796800000885 (cited on page 140)

[ARM10] ARM. *ARMv7-M Architecture Reference Manual*. Tech. rep. 2010 (cited on page 55)

[AS13] G. Amato and F. Scozzari. "Localizing widening and narrowing." In: *Static Analysis Symposium*. Ed. by F. Logozzo and M. Fähndrich. Vol. 7935. LNCS. Springer, 2013, pp. 25–42. DOI: 10.1007/978-3-642-38856-9_4 (cited on pages 19, 77, 93)

[ASV12] K. Apinis, H. Seidl, and V. Vojdani. "Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis." In: *Asian Symposium on Programming Languages and Systems*. Ed. by R. Jhala and A. Igarashi. Vol. 7705. LNCS. Springer Berlin Heidelberg, Dec. 2012, pp. 157–172. ISBN: 978-3-642-35181-5. DOI: 10.1007/978-3-642-35182-2_12 (cited on page 69)

[Bab+11] D. Babić, L. Martignoni, S. McCamant, and D. Song. "Statically-Directed Dynamic Automated Test Generation." In: *International Symposium on Software Testing and Analysis*. ISSTA '11. ACM, 2011, pp. 12–22. ISBN: 978-1-4503-0562-4. DOI: 10.1145/2001420.2001423 (cited on page 131)

[Bag+02] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. "Possibly Not Closed Convex Polyhedra and the Parma Polyhedra Library." In: *Static Analysis Symposium*. Ed. by M. V. Hermenegildo and G. Puebla. Vol. 2477. LNCS. Springer, Sept. 2002, pp. 213–229 (cited on page 52)

[Bag+05] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. "Precise Widening Operators for Convex Polyhedra." In: *Science of Computer Programming* 58.1–2 (2005), pp. 28–56 (cited on pages 75, 78, 92, 93)

[Bag+07] R. Bagnara, K. Dobson, P. M. Hill, M. Mundell, and E. Zaffanella. "Grids: A Domain for Analyzing the Distribution of Numerical Values." In: *Logic-Based Program Synthesis and Transformation*. Ed. by G. Puebla. Vol. 4407. LNCS. Springer Berlin Heidelberg, 2007, pp. 219–235. ISBN: 978-3-540-71409-5. DOI: 10.1007/978-3-540-71410-1_16 (cited on page 50)

[Bal+01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. "Automatic Predicate Abstraction of C Programs." In: *Programming Languages, Design and Implementation*. ACM, 2001, pp. 203–213 (cited on pages 96, 108)

[Bal+05] G. Balakrishnan, G. Grurian, T. Reps, and T. Teitelbaum. "CodeSurfer/x86 – A Platform for Analyzing x86 Executables." In: *Compiler Construction*. Vol. 3443. LNCS. Tool-Demonstration Paper. Springer, Apr. 2005, pp. 250–254 (cited on pages 68, 73, 75)

[Bal07] G. Balakrishnan. "WYSINWYX: What You See Is Not What You eXecute." PhD thesis. Winsconsin, Madison, 2007 (cited on pages 69, 92, 123)

[BDL06] S. Blazy, Z. Dargaye, and X. Leroy. "Formal Verification of a C Compiler Front-End." In: *Formal Methods*. Ed. by J. Misra, T. Nipkow, and E. Sekerinski. Vol. 4085. LNCS. Springer Berlin Heidelberg, Aug. 2006, pp. 460–475. ISBN: 978-3-540-37215-8. DOI: 10.1007/11813040_31 (cited on page 3)

[BDP12] G. Barthe, D. Demange, and D. Pichardie. "A Formally Verified SSA-Based Middle-End." In: *Programming Languages and Systems*. Ed. by H. Seidl. Vol. 7211. LNCS. Springer Berlin Heidelberg, Apr. 2012, pp. 47–66. ISBN: 978-3-642-28868-5. DOI: 10.1007/978-3-642-28869-2_3 (cited on page 3)

[Bel05] F. Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX Annual Technical Conference*. ATEC '05. USENIX Association, 2005, pp. 41–46 (cited on page 128)

[Ber+10] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. "Static Analysis and Verification of Aerospace Software by Abstract Interpretation." In: *AIAA Infotech@Aerospace 2010*. American Institute of Aeronautics and Astronautics, Apr. 2010, pp. 1–38. ISBN: 978-1-60086-963-1. DOI: 10.2514/6.2010-3385 (cited on pages 61, 63)

[BGA03] D. Bruening, T. Garnett, and S. Amarasinghe. "An Infrastructure for Adaptive Dynamic Optimization." In: *Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2003, pp. 265–275 (cited on page 128)

[BHT08] D. Beyer, T. Henzinger, and G. Théoduloz.
"Program analysis with dynamic precision adjustment."
In: *Automated Software Engineering*. 2008 (cited on page 108)

[BHV11] S. Bardin, P. Herrmann, and F. Védrine.
"Refinement-based CFG Reconstruction from Unstructured Programs."
In: *Verification, Model Checking, and Abstract Interpretation*.
Ed. by R. Jhala and D. Schmidt. Vol. 6538. LNCS.
Springer Berlin Heidelberg, 2011, pp. 54–69. ISBN: 978-3-642-18274-7.
DOI: 10.1007/978-3-642-18275-4_6 (cited on page 53)

[BHZ04] R. Bagnara, P. M. Hill, and E. Zaffanella.
"Widening Operators for Powerset Domains."
In: *Verification, Model Checking, and Abstract Interpretation*.
Ed. by B. Steffen and G. Levi. Vol. 2937. LNCS 4-5.
Springer Berlin Heidelberg, 2004, pp. 449–466. ISBN: 978-3-540-20803-7.
DOI: 10.1007/b94790 (cited on page 57)

[BHZ08] R. Bagnara, P. M. Hill, and E. Zaffanella. "The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems."
In: *Science of Computer Programming* 72.1-2 (2008), pp. 3–21.
DOI: 10.1016/j.scico.2007.08.001 (cited on pages 52, 55, 57)

[BK10] J. Brauer and A. King.
"Automatic Abstraction for Intervals using Boolean Formulae."
In: *Static Analysis Symposium*. Ed. by R. Cousot and M. Martel. LNCS.
Springer, Sept. 2010 (cited on pages 21, 22)

[Bla+02] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival.
"Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software."
In: *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*.
Ed. by T. Æ. Mogensen, D. A. Schmidt, and I. H. Sudborough. Vol. 2566.
LNCS. Springer, 2002, pp. 85–108 (cited on pages 73, 78, 92, 93, 140)

[Bla+03a] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival.
"A Static Analyzer for Large Safety-Critical Software."
In: *Programming Language Design and Implementation*. ACM, June 2003
(cited on pages 21, 27, 30, 44, 68, 73, 75, 78, 111, 120)

[Bla+03b] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival.
"A static analyzer for large safety-critical software."
In: *Programming Language Design and Implementation* 38.5 (2003). Ed. by

R. Cytron, pp. 196–207. ISSN: 03621340. DOI: 10.1145/780822.781153 (cited on pages 52, 69, 154)

[BLH12] S. Bygde, B. Lisper, and N. Holsti. "Fully Bounded Polyhedral Analysis of Integers with Wrapping." In: *Electronic Notes in Theoretical Computer Science* 288 (2012), pp. 3–13. ISSN: 15710661. DOI: 10.1016/j.entcs.2012.10.003 (cited on page 69)

[Boc09] D. R. Boccardo. "Context-sensitive analysis of x86 obfuscated executables." PhD Thesis. Universidade Estadual Paulista, 2009 (cited on page 69)

[Bou93] F. Bourdoncle. "Efficient Chaotic Iteration Strategies with Widenings." In: *Formal Methods in Programming and Their Applications*. Ed. by D. Bjørner, M. Broy, and I. V. Pottosin. Vol. 735. LNCS. Springer, June 1993, pp. 128–141 (cited on pages 16, 19, 73, 75–77, 94, 137)

[BR04] G. Balakrishnan and T. Reps. "Analyzing Memory Accesses in x86 Executables." In: *Compiler Construction*. Ed. by E. Duesterwald. Vol. 2985. LNCS. Springer Berlin Heidelberg, 2004, pp. 5–23. ISBN: 978-3-540-21297-3. DOI: 10.1007/978-3-540-24723-4_2 (cited on pages 24, 49)

[BR06] G. Balakrishnan and T. Reps. "Recency-Abstraction for Heap-Allocated Storage." In: *Static Analysis Symposium*. Ed. by K. Yi. Vol. 4134. LNCS. Springer, 2006, pp. 221–239 (cited on page 120)

[BR10] G. Balakrishnan and T. Reps. "WYSINWYX: What you see is not what you eXecute." In: *Transactions on Programming Languages and Systems* 32.6 (Aug. 2010), pp. 1–84. ISSN: 01640925. DOI: 10.1145/1749608.1749612 (cited on pages 3, 4)

[Bra+11] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina. "Exploit Programming: From Buffer Overflows to "Weird Machines" and Theory of Computation." In: *USENIX ;login:* 36.6 (Dec. 2011), pp. 13–21 (cited on page 147)

[Byg10] S. Bygde. "Static WCET Analysis based on Abstract Interpretation and Counting of Elements." Licentiate Thesis. Mälardalen University, 2010. ISBN: 9789186135553 (cited on page 41)

[CC04] R. Clarisó and J. Cortadella. "The Octahedron Abstract Domain." In: *Static Analysis Symposium*. Ed. by R. Giacobazzi. Vol. 3148. LNCS. Springer Berlin Heidelberg, Aug. 2004, pp. 115–139. ISBN: 0167-6423. DOI: 10.1016/j.scico.2006.03.009 (cited on page 52)

[CC10] P. Cousot and R. Cousot. “A gentle introduction to formal verification of computer systems by abstract interpretation.” In: *Logics and Languages for Reliability and Security*. Ed. by J. Esparza, O. Grumberg, and M. Broy. IOS Press, 2010, pp. 1–29 (cited on page 9)

[CC76] P. Cousot and R. Cousot. “Static Determination of Dynamic Properties of Programs.” In: *International Symposium on Programming*. Ed. by B. Robinet. Apr. 1976, pp. 106–130 (cited on pages 17, 18, 53, 54, 73)

[CC77] P. Cousot and R. Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.” In: *Principles of Programming Languages*. ACM, Jan. 1977, pp. 238–252 (cited on pages 10, 17, 18, 54, 73, 74)

[CC79a] P. Cousot and R. Cousot. “Systematic Design of Program Analysis Frameworks.” In: *Principles of Programming Languages*. ACM, Jan. 1979, pp. 269–282 (cited on pages 10, 55, 59, 64, 65, 96, 97, 108, 109)

[CC79b] P. Cousot and R. Cousot. “Constructive Versions of Tarski’s Fixed Point Theorems.” In: *Pacific Journal of Mathematics* 81.1 (1979), pp. 43–57 (cited on page 10)

[CC92a] P. Cousot and R. Cousot. “Abstract Interpretation and Application to Logic Programs.” In: *Journal of Logic Programming* 13.2–3 (1992), pp. 103–179 (cited on page 73)

[CC92b] P. Cousot and R. Cousot. “Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation.” In: *Programming Language Implementation and Logic Programming*. Ed. by M. Bruynooghe and M. Wirsing. Vol. 631. LNCS. Springer Berlin Heidelberg, Aug. 1992, pp. 269–295. ISBN: 978-3-540-55844-6. DOI: 10.1007/3-540-55844-6_101 (cited on pages 17, 18, 76)

[CCF13] A. Cortesi, G. Costantini, and P. Ferrara. “A Survey on Product Operators in Abstract Interpretation.” In: *Electronic Proceedings in Theoretical Computer Science* 129 (Sept. 2013), pp. 325–336. ISSN: 2075-2180. DOI: 10.4204/EPTCS.129.19. arXiv: arXiv:1309.5146 (cited on pages 59, 64)

[CCL11] P. Cousot, R. Cousot, and F. Logozzo. “A parametric segmentation functor for fully automatic and scalable array content analysis.” In: *Principles of Progamming Languages*. Ed. by T. Ball and M. Sagiv. Vol. 46. 1. ACM, Jan. 2011, pp. 105–118. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1925844.1926399 (cited on page 36)

[CCM11] P. Cousot, R. Cousot, and L. Mauborgne. "The Reduced Product of Abstract Domains and the Combination of Decision Procedures." In: *Foundations of Software Science and Computational Structures*. Ed. by H. Martin. Vol. 6604. LNCS. Springer Berlin Heidelberg, Mar. 2011, pp. 456–472. ISBN: 978-3-642-19804-5. DOI: 10.1007/978-3-642-19805-2_31 (cited on pages 59, 61)

[CD11] K. Coogan and S. Debray. "Equational Reasoning on x86 Assembly Code." In: *Source Code Analysis and Manipulation*. IEEE, Sept. 2011 (cited on page 69)

[CH78] P. Cousot and N. Halbwachs. "Automatic Discovery of Linear Constraints among Variables of a Program." In: *Principles of Programming Languages*. ACM, Jan. 1978, pp. 84–97 (cited on pages 17, 21, 24, 51, 75, 93, 99, 110, 118)

[CKC11] V. Chipounov, V. Kuznetsov, and G. Candea. "S2E: a platform for in-vivo multi-path analysis of software systems." In: *Architectural Support for Programming Languages and Operating Systems*. Ed. by R. Gupta and T. C. Mowry. ACM, Mar. 2011, pp. 265–278 (cited on page 131)

[CLV00] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. "Combinations of abstract domains for logic programming: open product and generic pattern construction." In: *Science of Computer Programming* 38.1-3 (Aug. 2000). Ed. by E. Astesiano and J. Bergstra, pp. 27–71. ISSN: 01676423. DOI: 10.1016/S0167-6423(99)00045-3 (cited on page 61)

[CN09] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Ed. by G. McGraw. 1st. Addison-Wesley Professional, Aug. 2009, p. 792. ISBN: 0321549252, 9780321549259 (cited on pages 3, 5)

[Coo+09] K. Coogan, S. Debray, T. Kaochar, and G. Townsend. "Automatic Static Unpacking of Malware Binaries." In: *Working Conference on Reverse Engineering*. IEEE, 2009, pp. 167–176 (cited on page 130)

[Cor08] A. Cortesi. "Widening Operators for Abstract Interpretation." In: *Software Engineering and Formal Methods*. Ed. by A. Cerone and S. Grune. IEEE Computer Society, Nov. 2008, pp. 31–40. ISBN: 978-0-7695-3437-4. DOI: 10.1109/SEFM.2008.20 (cited on pages 19, 76)

[Cou+06] P. Cousot, R. Cousot, J. Feret, A. Miné, L. Mauborgne, D. Monniaux, and X. Rival. "Combination of Abstractions in the ASTRÉE Static Analyzer." In: *Asian Computing Science Conference*. Ed. by M. Okada and I. Satoh.

Vol. 4435. LNCS. Springer, Dec. 2006, pp. 272–300
(cited on pages 32, 75, 78, 94, 113)

[Cou+07] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. "Combination of Abstractions in the ASTRÉE Static Analyzer." In: *Asian Computing Science Conference*. Ed. by M. Okada and I. Satoh. Vol. 4435. LNCS. Springer Berlin Heidelberg, 2007, pp. 1–29. ISBN: 978-3-540-77504-1. DOI: 10.1007/978-3-540-77505-8
(cited on page 61)

[Cou01] P. Cousot.
"Abstract Interpretation Based Formal Methods and Future Challenges."
In: *Informatics - 10 Years Back, 10 Years Ahead*. Ed. by R. Wilhelm.
LNCS. Vol. 2000. LNCS 562. Springer Berlin Heidelberg, Mar. 2001,
pp. 138–156. ISBN: 978-3-540-41635-7. DOI: 10.1007/3-540-44577-3_10
(cited on pages 9, 49)

[CS98] C. Cifuentes and S. Sendall.
"Specifying the Semantics of Machine Instructions."
In: *International Workshop on Program Comprehension*. 1998, pp. 126–133
(cited on page 21)

[CV04] L. Catuogno and I. Visconti.
"An Architecture for Kernel-Level Verification of Executables at Run Time."
In: *The Computer Journal* 47.5 (2004), pp. 511–526 (cited on page 123)

[DLS02] M. Das, S. Lerner, and M. Seigle.
"ESP: Path-Sensitive Program Verification in Polynomial Time."
In: *ACM SIGPLAN Notices* 37.5 (May 2002), p. 57 (cited on page 120)

[Dow03] M. Dowd. *Remote Sendmail Header Processing Vulnerability*. Mar. 2003.
URL: http://www.iss.net/threats/advise142.html
(cited on page 147)

[DP09] T. Dullien and S. Porst. *REIL: A platform-independent intermediate representation of disassembled code for static code analysis*.
CanSecWest Vancouver, Canada. 2009 (cited on pages 21–23, 25)

[DP10] S. Debray and J. Patel.
"Reverse Engineering Self-Modifying Code: Unpacker Extraction."
In: *Working Conference on Reverse Engineering*. IEEE, Oct. 2010, pp. 131–140
(cited on page 130)

[DRS07] B. Dufour, B. G. Ryder, and G. Sevitsky. "Blended analysis for performance understanding of framework-based applications."
In: *Proceedings of the 2007 international symposium on Software testing and analysis*. ISSTA '07. ACM, 2007, pp. 118–128 (cited on page 130)

[Dul11] T. Dullien. “Exploitation and State Machines - Programming the "weird machine" revisited.” In: *Infiltrate Security Conference*. Apr. 2011 (cited on page 147)

[Eld+11] M. Elder, J. Lim, T. Sharma, T. Andersen, and T. Reps. “Abstract Domains of Affine Relations.” In: *Static Analysis Symposium*. Ed. by E. Yahav. Vol. 6887. LNCS. Springer Berlin Heidelberg, Sept. 2011, pp. 198–215. ISBN: 9783642237010. DOI: 10.1007/978-3-642-23702-7_17 (cited on pages 25, 68)

[EMA10] N. Een, A. Mishchenko, and N. Amla. “A Single-Instance Incremental SAT Formulation of Proof- and Counterexample-Based Abstraction.” In: *Formal Methods in Computer-Aided Design*. ACM, Aug. 2010, pp. 181–188. arXiv: 1008.2021 (cited on page 40)

[EMS07] N. Een, A. Mishchenko, and N. Sörensson. “Applying Logic Synthesis for Speeding Up SAT.” In: *Theory and Applications of Satisfiability Testing*. Ed. by J. Marques-Silva and K. A. Sakallah. Vol. 4501. LNCS. Springer Berlin Heidelberg, May 2007, pp. 272–286. ISBN: 978-3-540-72787-3. DOI: 10.1007/978-3-540-72788-0_26 (cited on page 40)

[FJM05] J. Fischer, R. Jhala, and R. Majumdar. “Joining Dataflow with Predicates.” In: *European Software Engineering Conference*. Ed. by M. Wermelinger and H. Gall. Vol. 30. ACM, Sept. 2005, pp. 227–236 (cited on pages 104, 105, 109)

[FL09] M. Fähndrich and F. Logozzo. “Pentagons: A Weakly Relational Abstract Domain for the Efficient Validation of Array Accesses.” In: *Science of Computer Programming* (2009) (cited on page 51)

[FL11] M. Fähndrich and F. Logozzo. “Clousot: Static Contract Checking with Abstract Interpretation.” In: *Formal Verification of Object-Oriented Software*. Ed. by B. Beckert and C. Marché. Vol. 6528. LNCS. Springer Berlin Heidelberg, 2011, pp. 10–30. DOI: 10.1007/978-3-642-18070-5_2 (cited on pages 56, 61, 154)

[Fle+10] A. Flexeder, B. Mihaila, M. Petter, and H. Seidl. “Interprocedural control flow reconstruction.” In: *Asian Symposium on Program Languages and Systems*. Ed. by K. Ueda. Vol. 6461. LNCS. Springer, Nov. 2010, pp. 188–203. DOI: 10.1007/978-3-642-17164-2_14 (cited on page 4)

[GC10a] A. Gurfinkel and S. Chaki. “Boxes: A Symbolic Abstract Domain of Boxes.” In: *Static Analysis Symposium*. Ed. by R. Cousot and M. Martel. Vol. 6337. LNCS. Springer, 2010, pp. 287–303 (cited on pages 56, 57, 96, 108)

[GC10b] A. Gurfinkel and S. Chaki. "Combining Predicate and Numeric Abstraction for Software Model Checking."
In: *Software Tools for Technology Transfer* 12.6 (2010), pp. 409–427
(cited on pages 105, 107, 108)

[GGP09] K. Ghorbal, E. Goubault, and S. Putot.
"The Zonotope Abstract Domain Taylor1+." In: *Computer Aided Verification*.
Ed. by A. Bouajjani and O. Maler. Vol. 5643. LNCS.
Springer Berlin Heidelberg, 2009, pp. 627–633. ISBN: 3642026575.
DOI: 10.1007/978-3-642-02658-4_47 (cited on page 53)

[GLM12] P. Godefroid, M. Y. Levin, and D. Molnar.
"SAGE: Whitebox Fuzzing for Security Testing." In: 10.1 (Jan. 2012)
(cited on page 123)

[GMT08] S. Gulwani, B. McCloskey, and A. Tiwari.
"Lifting abstract interpreters to quantified logical domains."
In: *Principles of Progamming Languages*. ACM, Jan. 2008, pp. 235–246
(cited on page 120)

[God07] P. Godefroid. "Compositional dynamic test generation."
In: *Principles of Programming Languages*. ACM, 2007, pp. 47–54.
ISBN: 1-59593-575-4.
DOI: http://doi.acm.org/10.1145/1190216.1190226
(cited on page 124)

[Gop+04] D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv.
"Numeric Domains with Summarized Dimensions."
In: *Tools and Algorithms for the Construction and Analysis of Systems*.
Ed. by K. Jensen and A. Podelski. Vol. 2988. LNCS.
Springer Berlin Heidelberg, 2004, pp. 512–529. ISBN: 978-3-540-21299-7.
DOI: 10.1007/978-3-540-24730-2_38 (cited on page 36)

[GR06a] D. Gopan and T. Reps. "Lookahead Widening."
In: *Computer-Aided Verification*. Ed. by T. Ball and R. B. Jones. Vol. 4144.
LNCS. Springer, Aug. 2006 (cited on pages 73, 91, 92, 94)

[GR06b] B. S. Gulavani and S. K. Rajamani.
"Counterexample Driven Refinement for Abstract Interpretation."
In: *Tools and Algorithms for the Construction and Analysis of Systems*.
Ed. by H. Hermanns and J. Palsberg. Vol. 3920. LNCS. Springer, Mar. 2006,
pp. 474–488 (cited on pages 109, 154)

[GR07] D. Gopan and T. W. Reps. "Guided Static Analysis."
In: *Static Analysis Symposium*. Ed. by H. R. Nielson and G. Filé. Vol. 4634.
LNCS. Springer, Aug. 2007, pp. 349–365
(cited on pages 73, 75, 88, 94, 96, 108)

[GR99] R. Giacobazzi and F. Ranzato.
"The reduced relative power operation on abstract domains." In: *Theoretical Computer Science* 216.1-2 (Mar. 1999). Ed. by M. Nivat, pp. 159–211.
ISSN: 03043975. DOI: 10.1016/S0304-3975(98)00194-7
(cited on pages 64, 65)

[Gra89] P. Granger. "Static Analysis of Arithmetic Congruences." In: *International Journal of Computer Mathematics* 30 (3 & 4 1989), pp. 165–199
(cited on pages 48–50)

[Gra91] P. Granger. "Static Analysis of Linear Congruence Equalities among Variables of a Program." In: *Theory and Practice of Software Development*. Ed. by S. Abramsky and T. S. E. Maibaum. Vol. 493. LNCS.
Springer, Apr. 1991, pp. 169–192 (cited on page 50)

[Gra92] P. Granger. "Improving the Results of Static Analyses of Programs by Local Decreasing Iterations."
In: *Foundations of Software Technology and Theoretical Computer Science*. Ed. by R. Shyamasundar. Vol. 652. LNCS. Springer, 1992, pp. 68–79.
ISBN: 978-3-540-56287-0. DOI: 10.1007/3-540-56287-7_95
(cited on pages 61–63, 97)

[Gra97] P. Granger. "Static Analyses of Congruence Properties on Rational Numbers (Extended Abstract)." In: *Static Analysis Symposium*.
Ed. by P. Van Hentenryck. Springer, Sept. 1997, pp. 278–292
(cited on page 50)

[GS97] S. Graf and H. Saidi. "Construction of abstract state graphs with PVS."
In: *Computer Aided Verification*. Ed. by O. Grumberg. Vol. 1254. LNCS.
Springer, 1997, pp. 72–83. DOI: 10.1007/3-540-63166-6_10
(cited on page 108)

[Har77] W. H. Harrison. "Compiler Analysis of the Value Ranges for Variables."
In: *Transactions on Software Engineering* 3.3 (May 1977), pp. 243–250
(cited on pages 17, 21, 53)

[Hee11] S. Heelan. "Vulnerability Detection Systems: Think Cyborg, Not Robot."
In: *IEEE Security and Privacy* 9.June (2011), pp. 74–77. ISSN: 15407993.
DOI: 10.1109/MSP.2011.70 (cited on page 147)

[HH12] N. Halbwachs and J. Henry. "When the Decreasing Sequence Fails."
In: *Static Analysis Symposium*. Ed. by A. Miné and D. Schmidt. Vol. 7460.
LNCS. Springer, Sept. 2012, pp. 198–213.
DOI: 10.1007/978-3-642-33125-1_15
(cited on pages 16, 51, 73, 74, 85, 91–93)

[HHP13] M. Heizmann, J. Hoenicke, and A. Podelski.
"Software Model Checking for People Who Love Automata."
In: *Computer Aided Verification*. Ed. by N. Sharygina and H. Veith.
Vol. 8044. LNCS. Springer, July 2013, pp. 36–52 (cited on page 105)

[HMC94] J. Hollingsworth, B. Miller, and J. Cargille.
"Dynamic program instrumentation for scalable performance tools."
In: *Scalable High-Performance Computing Conference, 1994*. IEEE. 1994,
pp. 841–850 (cited on page 128)

[HPR97] N. Halbwachs, Y.-E. Proy, and P. Roumanoff.
"Verification of Real-Time Systems using Linear Relation Analysis."
In: *Formal Methods in System Design* 11.2 (Aug. 1997), pp. 157–185
(cited on pages 73, 75, 78, 92, 94)

[IBM03] IBM Internet Security Systems X-Force.
Sendmail mail header processing buffer overflow. Mar. 2003.
URL: http://xforce.iss.net/xforce/xfdb/10748 (cited on page 147)

[ISO05] ISO C Working Group. *ANSI C 98/99*. Tech. rep. 2005 (cited on page 55)

[Jaf+12] J. Jaffar, V. Murali, J. Navas, and S. Andrew.
"TRACER: A Symbolic Execution Tool for Verification."
In: *Computer Aided Verification*. Ed. by P. Madhusudan and S. A. Seshia.
Vol. 7358. LNCS. Springer, 2012, pp. 758–766 (cited on pages 123, 131)

[JM09] B. Jeannet and A. Miné.
"Apron: A Library of Numerical Abstract Domains for Static Analysis."
In: *Computer Aided Verification*. Ed. by A. Bouajjani and O. Maler.
Vol. 5643. LNCS. Springer, June 2009, pp. 661–667 (cited on pages 50, 92)

[JRM11] E. R. Jacobson, N. E. Rosenblum, and B. P. Miller.
"Labeling library functions in stripped binaries."
In: *Program analysis for software tools*. ACM, 2011, pp. 1–8
(cited on page 128)

[Kar76] M. Karr. "On affine relationships among variables of a program."
In: *Acta Informatica* 6.2 (1976), pp. 133–151 (cited on pages 21, 43, 75)

[Kin10] J. Kinder. "Static Analysis of x86 Executables."
PhD Thesis. Technische Universität Darmstadt, 2010 (cited on page 69)

[Kin12] J. Kinder. "Towards Static Analysis of Virtualization-Obfuscated Binaries."
In: *Working Conference on Reverse Engineering*.
Ed. by R. Oliveto, D. Poshyvanyk, J. Cordy, and T. Dean.
IEEE Computer Society, Oct. 2012, pp. 61–70. DOI: 10.1109/WCRE.2012.16
(cited on pages 5, 14, 154)

[Kir+12] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski.
"Frama-C A Software Analysis Perspective."
In: *Software Engineering and Formal Methods*.
Ed. by G. Eleftherakis, M. Hinchey, and M. Holcombe. Vol. 7504. LNCS.
Springer Berlin Heidelberg, 2012, pp. 233–247. ISBN: 978-3-642-33825-0.
DOI: 10.1007/978-3-642-33826-7_16 (cited on pages 73, 92, 139)

[KK08a] N. Kettle and A. King. "Bit-Precise Reasoning with Affine Functions." In: *Satisfiability Modulo Theories*. ACM, 2008, pp. 46–52. ISBN: 978-1-60558-440-9 (cited on page 21)

[KK08b] U. Khedker and B. Karkare. "Efficiency, Precision, Simplicity, and Generality in Interprocedural Data Flow Analysis: Resurrecting the Classical Call Strings Method." In: *Conference on Compiler Construction*. Ed. by L. Hendren. Vol. 4959. LNCS. Springer Berlin Heidelberg, Mar. 2008, pp. 213–228. ISBN: 3-540-78790-9, 978-3-540-78790-7. DOI: 10.1007/978-3-540-78791-4_15 (cited on page 69)

[KK12] J. Kinder and D. Kravchenko. "Alternating Control Flow Reconstruction." In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by V. Kuncak and A. Rybalchenko. Vol. 7148. LNCS. Springer Berlin Heidelberg, 2012, pp. 267–282. ISBN: 978-3-642-27939-3. DOI: 10.1007/978-3-642-27940-9 (cited on pages 130, 131)

[KS10] A. King and H. Søndergaard. "Automatic Abstraction for Congruences." In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by G. Barthe and M. Hermenegildo. LNCS 5944. Springer, Jan. 2010, pp. 197–213 (cited on pages 25, 68)

[KSS13a] J. Kranz, A. Sepp, and A. Simon. "GDSL: A Universal Toolkit for Giving Semantics to Machine Language." In: *Asian Symposium on Programming Languages and Systems*. Ed. by C. Shan. Springer, Dec. 2013 (cited on pages 38, 154)

[KSS13b] J. Kranz, A. Sepp, and A. Simon. "GDSL: A Universal Toolkit for Giving Semantics to Machine Language." In: *Asian Symposium on Program Languages and Systems*. Ed. by C.-c. Shan. Vol. 8301. LNCS. Springer, 2013, pp. 209–216. ISBN: 978-3-319-03541-3. DOI: 10.1007/978-3-319-03542-0 (cited on page 135)

[KVZ09] J. Kinder, H. Veith, and F. Zuleger. "An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries." In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by M. M.-O. N. D. Jones. Vol. 5403. LNCS. Springer, Jan. 2009, pp. 214–228 (cited on pages 4, 38, 123)

[Lak+11] A. Lakhotia, D. R. Boccardo, A. Singh, and A. Manacero. "Context-sensitive analysis without calling-context." In: *Higher-Order and Symbolic Computation* 23.3 (Nov. 2011). Ed. by O. Danvy and C. L. Talcott, pp. 275–313. ISSN: 1388-3690. DOI: 10.1007/s10990-011-9080-1 (cited on pages 33, 69)

[Ler06] X. Leroy. "Formal Certification of a Compiler Back-end."
In: *Principles of Programming Languages*.
Ed. by G. Morrisett and S. P. Jones. Vol. 41. ACM Press, Jan. 2006,
pp. 42–54. ISBN: 1595930272. DOI: 10.1145/1111037.1111042
(cited on page 3)

[Lev99] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., 1999.
ISBN: 1558604960 (cited on pages 32, 35)

[LF08] F. Logozzo and M. Fähndrich. "Pentagons: a weakly relational abstract
domain for the efficient validation of array accesses."
In: *ACM Symposium on Applied Computing*.
Ed. by R. L. Wainwright and H. M. Haddad. ACM Press, 2008, p. 184.
ISBN: 9781595937537. DOI: 10.1145/1363686.1363736
(cited on page 109)

[LJG11] L. Lakhdar-Chaouch, B. Jeannet, and A. Girault.
"Widening with Thresholds for Programs with Complex Control Graphs."
In: *Automated Technology for Verification and Analysis*.
Ed. by T. Bultan and P. Hsiung. Vol. 6996. LNCS. Springer, Oct. 2011,
pp. 492–502. DOI: 10.1007/978-3-642-24372-1_38
(cited on pages 53, 73, 75, 91–93)

[LL05] R. Leino and F. Logozzo. "Loop Invariants on Demand."
In: *Asian Symposium on Programming Languages and Systems*. Ed. by K. Yi.
Vol. 3780. LNCS. Springer, 2005, pp. 119–134 (cited on page 109)

[LL11] V. Laviron and F. Logozzo. "SubPolyhedra: a family of numerical abstract
domains for the (more) scalable inference of linear inequalities."
In: *International Journal on Software Tools for Technology Transfer* 13.6 (May
2011), pp. 585–601. ISSN: 1433-2779. DOI: 10.1007/s10009-011-0199-5
(cited on pages 73, 92, 99, 109)

[LSD03] LSD - Last Stage of Delirium.
Technical analysis of the remote sendmail vulnerability. Mar. 2003.
URL: http://www.ouah.org/LSDsendmail.html
(cited on pages 147, 149)

[Luk+05] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace,
V. J. Reddi, and K. Hazelwood. "Pin: building customized program analysis
tools with dynamic instrumentation."
In: *Programming language design and implementation*. PLDI.
ACM, June 2005, pp. 190–200 (cited on pages 128, 139)

[Mar99] F. Martin. "Experimental Comparison of Call String and Functional
Approaches to Interprocedural Analysis." In: *Compiler Construction*.
Ed. by S. Jähnichen. Vol. 1575. LNCS.
Springer Berlin Heidelberg, Mar. 1999, pp. 63–75. ISBN: 978-3-540-65717-0.
DOI: 10.1007/978-3-540-49051-7_5 (cited on pages 68, 69)

[McM11] K. L. McMillan. “Widening and Interpolation.”
In: *Static Analysis Symposium*. Ed. by E. Yahav. Vol. 6887. LNCS.
Springer Berlin Heidelberg, Sept. 2011, p. 1. ISBN: 978-3-642-23701-0.
DOI: 10.1007/978-3-642-23702-7_1 (cited on page 79)

[Mic99] Microsoft Corporation.
Microsoft Portable Executable and Common Object File Format Specification.
Feb. 1999.
URL: http://download.microsoft.com/download/e/b/a/eba1050f-a31d-436b-9281-92cdfeae4b45/pecoff.doc (cited on page 135)

[Mih09] B. Mihaila. “Control Flow Reconstruction from PowerPC Binaries.”
Diploma Thesis. Technical University of Munich, Nov. 2009
(cited on page 49)

[Mih14a] B. Mihaila. *BinDead - a static analysis tool for binaries*. Oct. 2014.
URL: https://bitbucket.org/mihaila/bindead
(cited on pages 136, 153)

[Mih14b] B. Mihaila. *BintTrace - a tool to record and dump traces of an executable program and its data*. Oct. 2014.
URL: https://bitbucket.org/mihaila/bintrace (cited on page 139)

[Mih14c] B. Mihaila. *p9 - the GUI for the BinDead binary analyzer*. Oct. 2014.
URL: https://bitbucket.org/mihaila/p9 (cited on page 143)

[Min01] A. Miné. “The Octagon Abstract Domain.”
In: *Conference on Reverse Engineering*. IEEE, Oct. 2001, pp. 310–319
(cited on page 52)

[Min06a] A. Miné. “Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics.”
In: *Languages, Compilers, and Tools for Embedded Systems*. ACM, June 2006, pp. 54–63 (cited on pages 24, 69)

[Min06b] A. Miné.
“Symbolic Methods to Enhance the Precision of Numerical Abstract Domains.”
In: *Verification, Model Checking, and Abstract Interpretation*.
Ed. by E. A. Emerson and K. S. Namjoshi. Vol. 3855. LNCS.
Springer Berlin Heidelberg, 2006, pp. 348–363.
DOI: 10.1007/11609773_23 (cited on page 45)

[Min06c] A. Miné. “The Octagon Abstract Domain.” In: *Higher-Order and Symbolic Computation* 19 (2006). Ed. by O. Danvy, pp. 31–100 (cited on page 52)

[Min12] A. Miné. “Abstract domains for bit-level machine integer and floating-point operations.” In: *International Workshop on Invariant Generation*.
Ed. by J. F. And, P. Höfner, A. McIver, and A. Smaill. Vol. 17. 2012,
pp. 55–70 (cited on page 41)

[MR05] L. Mauborgne and X. Rival.
"Trace Partitioning in Abstract Interpretation Based Static Analyzers."
In: *European Conference on Programming Languages and Systems*.
Ed. by M. Sagiv. Vol. 3444. LNCS. Springer Berlin Heidelberg, Apr. 2005,
pp. 5–20. ISBN: 978-3-540-25435-5. DOI: 10.1007/978-3-540-31987-0_2
(cited on pages 96, 103, 105, 107–109)

[MS05] M. Müller-Olm and H. Seidl. "Analysis of modular arithmetic."
In: *Programming Languages and Systems*. Ed. by M. Sagiv. Vol. 344. LNCS 5.
Springer Berlin Heidelberg, Apr. 2005, pp. 46–60.
DOI: 10.1145/1275497.1275504 (cited on page 41)

[MS14] B. Mihaila and A. Simon.
"Synthesizing Predicates from Abstract Domain Losses."
In: *NASA Formal Methods*. Ed. by J. Badger and K. Y. Rosier. Vol. 8430.
LNCS. Springer, Apr. 2014, pp. 328–342.
DOI: 10.1007/978-3-319-06200-6_28 (cited on pages 5, 6)

[MSS13] B. Mihaila, A. Sepp, and A. Simon. "Widening as Abstract Domain."
In: *NASA Formal Methods*. Ed. by G. Brat, N. Rungta, and A. Venet.
Vol. 7871. LNCS. Springer, May 2013, pp. 170–186.
DOI: 10.1007/978-3-642-38088-4_12 (cited on pages 5, 6, 108)

[Nav+12] J. Navas, P. Schachte, H. Søndergaard, and P. Stuckey.
"Signedness-Agnostic Program Analysis: Precise Integer Bounds for
Low-Level Code."
In: *Asian Symposium on Programming Languages and Systems*.
Springer Berlin Heidelberg, 2012, pp. 115–130 (cited on page 68)

[Nec97] G. C. Necula. "Proof-carrying code."
In: *Principles of Programming Languages*.
Ed. by N. D. Jones, F. Henglein, and P. Lee. Vol. 243. 65.
ACM Press, Jan. 1997, pp. 106–119. ISBN: 0897918533.
DOI: 10.1145/263699.263712 (cited on page 3)

[NS07] N. Nethercote and J. Seward.
"Valgrind: a framework for heavyweight dynamic binary instrumentation."
In: *Programming language design and implementation*. PLDI. ACM, 2007,
pp. 89–100 (cited on page 128)

[Orm06] T. Ormandy.
Heap-based buffer overflow in the JPEG decoder in the TIFF library. Aug. 2006.
URL:
http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3460
(cited on pages 124, 130)

[OS] J. L. Obes and J. Schuh. *Integer underflow bug in Google Chrome*.
URL: http://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html (cited on page 125)

[PH07] M. Péron and N. Halbwachs. “An Abstract Domain Extending Difference-Bound Matrices with Disequality Constraints.” In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by B. Cook and A. Podelski. Vol. 4349. LNCS. Springer, Jan. 2007, pp. 268–282 (cited on page 96)

[RBL06] T. Reps, G. Balakrishnan, and J. Lim. “Intermediate-representation recovery from low-level code.” In: *Partial evaluation and semantics-based program manipulation*. Ed. by J. Hatcliff and F. Tip. ACM Press, 2006, pp. 100–111. ISBN: 1595931961. DOI: 10.1145/1111542.1111560 (cited on page 49)

[Rey02] J. C. Reynolds. “Separation logic: A logic for shared mutable data structures.” In: *Logic in Computer Science*. IEEE, 2002, pp. 55–74 (cited on page 120)

[Riv03] X. Rival. “Abstract Interpretation-Based Certification of Assembly Code.” In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by L. D. Zuck, P. C. Attie, A. Cortesi, and S. Mukhopadhyay. Vol. 2575. LNCS. Springer Berlin Heidelberg, Dec. 2003, pp. 41–55. ISBN: 978-3-540-00348-9. DOI: 10.1007/3-540-36384-X_7 (cited on pages 3, 139)

[San+06] S. Sankaranarayanan, F. Ivančić, I. Shlyakhter, and A. Gupta. “Static Analysis in Disjunctive Numerical Domains.” In: *Static Analysis Symposium*. Ed. by K. Yi. Vol. 4134. LNCS. Springer, Aug. 2006, pp. 3–17 (cited on pages 96, 108)

[SC07] Y. Smaragdakis and C. Csallner. “Combining Static and Dynamic Reasoning for Bug Detection.” In: *Test and Proofs*. Ed. by Y. Gurevich and B. Meyer. Vol. 4454. LNCS. Springer, Feb. 2007, pp. 1–16 (cited on page 130)

[Sch+14] M. Schwarz, H. Seidl, V. Vojdani, and K. Apinis. “Precise Analysis of Value-Dependent Synchronization in Priority Scheduled Programs.” In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by K. L. McMillan and X. Rival. Vol. 8318. LNCS. Springer Berlin Heidelberg, 2014, pp. 21–38. ISBN: 978-3-642-54012-7. DOI: 10.1007/978-3-642-54013-4_2 (cited on page 110)

[SDA02] B. Schwarz, S. Debray, and G. Andrews. “Disassembly of Executable Code Revisited.” In: *Working Conference on Reverse Engineering*. IEEE Computer Society, 2002, pp. 45–54 (cited on page 4)

[Sha07] H. Shacham. “The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86).” In: *Computer and Communications Security*.

Ed. by S. De Capitani di Vimercati and P. Syverson. ACM Press, Oct. 2007, pp. 552–61 (cited on page 4)

[SIG07] S. Sankaranarayanan, F. Ivančić, and A. Gupta. "Program Analysis Using Symbolic Ranges." In: *Static Analysis Symposium*. Ed. by H. R. Nielson and G. Filé. Vol. 4634. LNCS. Springer Berlin Heidelberg, Aug. 2007, pp. 366–383. ISBN: 978-3-540-74060-5. DOI: 10.1007/978-3-540-74061-2 (cited on page 69)

[Sim08a] A. Simon. "Splitting the Control Flow with Boolean Flags." In: *Static Analysis Symposium*. Ed. by M. Alpuente and G. Vidal. Vol. 5079. LNCS. Springer, July 2008, pp. 315–331 (cited on pages 39, 67, 109)

[Sim08b] A. Simon. *Value-Range Analysis of C Programs*. Springer, Aug. 2008. ISBN: 978-1-84800-016-2. DOI: 10.1007/978-1-84800-017-9 (cited on pages 24, 25)

[SK02] A. Simon and A. King. "Analyzing String Buffers in C." In: *Algebraic Methodology and Software Technology*. Ed. by H. Kirchner and C. Ringeissen. Vol. 2422. LNCS. Springer, Sept. 2002, pp. 365–379 (cited on page 36)

[SK06] A. Simon and A. King. "Widening Polyhedra with Landmarks." In: *Asian Symposium on Programming Languages and Systems*. Ed. by N. Kobayashi. Vol. 4279. LNCS. Springer, Nov. 2006, pp. 166–182 (cited on pages 69, 73, 86, 93)

[SK07] A. Simon and A. King. "Taming the Wrapping of Integer Arithmetic." In: *Static Analysis Symposium*. Ed. by G. File and H. R. Nielson. Vol. 4634. LNCS. Springer, Aug. 2007, pp. 121–136 (cited on pages 40, 41)

[SKH03] A. Simon, A. King, and J. M. Howe. "Two Variables per Linear Inequality as an Abstract Domain." In: *Logic-Based Program Synthesis and Transformation*. Ed. by M. Leuschel. Vol. 2664. LNCS. Springer, Sept. 2003, pp. 71–89 (cited on page 51)

[SKS12] A. Sepp, J. Kranz, and A. Simon. "GDSL: A Generic Decoder Specification Language for Interpreting Machine Language." In: *Tools for Automatic Program Analysis*. Vol. 289. ENTCS. Springer, Sept. 2012, pp. 53–64. DOI: 10.1016/j.entcs.2012.11.006 (cited on page 135)

[SM03] A. Sabelfeld and A. Myers. "Language-based information-flow security." In: *Selected Areas in Communications* 21.1 (2003), pp. 5–19 (cited on page 131)

[SMA05] K. Sen, D. Marinov, and G. Agha.
"CUTE: A Concolic Unit Testing Engine for C." In: *European software engineering conference held jointly with Foundations of software engineering*. ACM, 2005, pp. 263–272 (cited on pages 123, 124, 131)

[SMS11] A. Sepp, B. Mihaila, and A. Simon.
"Precise Static Analysis of Binaries by Extracting Relational Information."
In: *Working Conference on Reverse Engineering*.
Ed. by M.Pinzger and D. Poshyvanyk. IEEE, Oct. 2011.
doi: 10.1109/WCRE.2011.50 (cited on pages 5, 75, 85, 90, 107, 135)

[SMS13] H. Siegel, B. Mihaila, and A. Simon. "The Undefined Domain: Precise Relational Information for Entities that Do Not Exist."
In: *Asian Symposium on Programming Languages and Systems*.
Ed. by C. Shan. LNCS. Springer, Dec. 2013.
doi: 10.1007/978-3-319-03542-0_6 (cited on pages 5, 6, 119)

[Son+08] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena.
"BitBlaze: A New Approach to Computer Security via Binary Analysis."
In: *International Conference on Information Systems Security*.
Ed. by R. Sekar and A. K. Pujari. Springer Berlin Heidelberg, Dec. 2008, pp. 1–25. doi: 10.1007/978-3-540-89862-7_1 (cited on page 25)

[SP81] M. Sharir and A. Pnueli.
"Two approaches to interprocedural data flow analysis." In:
Program Flow Analysis: Theory and Applications. Prentice-Hall, 1981.
Chap. 7, pp. 189–234 (cited on page 65)

[SRW02] M. Sagiv, T. Reps, and R. Wilhelm.
"Parametric Shape Analysis via 3-Valued Logic." In: *Transactions on Programming Languages and Systems* 24.3 (2002), pp. 217–298
(cited on page 120)

[SS13] H. Siegel and A. Simon. "FESA: Fold- and Expand-based Shape Analysis."
In: *Compiler Construction*. Vol. 7791. LNCS. Springer, Mar. 2013, pp. 82–101
(cited on pages 34, 118)

[SSM05] S. Sankaranarayanan, H. B. Sipma, and Z. Manna.
"Scalable Analysis of Linear Systems using Mathematical Programming."
In: *Verification, Model Checking, and Abstract Interpretation*.
Ed. by R. Cousot. Vol. 3385. LNCS. Springer Berlin Heidelberg, Jan. 2005, pp. 25–41. isbn: 978-3-540-24297-0. doi: 10.1007/978-3-540-30579-8_2
(cited on page 109)

[TA05] T. Terauchi and A. Aiken. "Secure Information Flow as a Safety Problem."
In: *Static Analysis Symposium*. Ed. by C. Hankin and I. Siveroni. Vol. 3672. LNCS. Springer, Sept. 2005, pp. 352–367 (cited on page 131)

[Tha+10] A. V. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. W. Reps.
"Directed Proof Generation for Machine Code."
In: *Computer Aided Verification*. Springer, 2010, pp. 288–305.
doi: 10.1007/978-3-642-14295-6_27 (cited on page 4)

[Too95] Tool Interface Standards Commitee (TIS).
Executable and Linkable Format (ELF) v1.2. Tech. rep. 1995
(cited on page 135)

[VB04] A. Venet and G. Brat. "Precise and efficient static array bound checking for large embedded C programs."
In: *Programming Language Design and Implementation*. Vol. 39. 6. June 2004, p. 231. isbn: 1-58113-807-5. doi: 10.1145/996893.996869
(cited on pages 52, 69, 154)

[Ven12] A. Venet.
"The Gauge Domain: Scalable Analysis of Linear Inequality Invariants."
In: *Computer Aided Verification*. Vol. 7358.
Lecture Notes in Computer Science. 2012, pp. 139–154.
isbn: 978-3-642-31423-0 (cited on page 48)

[Ven96] A. Venet. "Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs." In: *Static Analysis Symposium*. LNCS. Springer, 1996, pp. 366–382 (cited on pages 30, 73, 96, 109, 112, 113)

[VHR12] J. Vanegue, S. Heelan, and R. Rolles. "SMT Solvers for Software Security."
In: *Workshop on Offensive Technologies*. 2012 (cited on pages 147, 149)

[VL12] J. Vanegue and S. K. Lahiri.
"Modern static security checking of C / C++ programs."
In: *RECON: Reverse Engineering Conference*. June 2012 (cited on page 147)

[Wag00] D. Wagner. "Static analysis and computer security: New techniques for software assurance."
PhD thesis. University of California at Berkeley, Dec. 2000
(cited on page 131)

[War03] H. S. J. Warren. *Hacker's Delight*. 1st. Addison-Wesley, 2003.
isbn: 0-201-91465-4 (cited on pages 54, 55)

[WKC13] J. Wagner, V. Kuznetsov, and G. Candea.
"-OVERIFY: Optimizing Programs for Fast Verification."
In: *Workshop on Hot Topics in Operating Systems*. May.
USENIX Association, 2013 (cited on page 149)

[WZ91] M. N. Wegman and F. K. Zadeck.
"Constant Propagation with Conditional Branches." In: *ACM Transactions on Programming Languages and Systems* 13.2 (Apr. 1991), pp. 181–210.
doi: 10.1145/103135.103136 (cited on pages 44, 53)