Scalable Program Analysis: Abstract Interpretation Techniques and Practical Applications

by

Yusen Su

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2026

© Yusen Su 2026

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Antoine Miné
Full Professor
Computer Science
Sorbonne Université

Supervisor(s): Arie Gurfinkel
Professor
Department of Electrical and Computer Engineering
University of Waterloo

Internal Member: Patrick Lam
Associate Professor
Department of Electrical and Computer Engineering
University of Waterloo

Internal Member: Werner Dietl
Associate Professor
Department of Electrical and Computer Engineering
University of Waterloo

Internal-External Member: Richard Trefler
Associate Professor
David R. Cheriton School of Computer Science
University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

All work in this thesis was conducted in collaboration with my supervisor, Prof. Arie Gurfinkel, and with Dr. Jorge A. Navas (Certora Inc.). Chapter 3 was additionally carried out in collaboration with Dr. Isabel Garcia-Contreras.

Abstract

Program analysis aims to infer program behavior from formal semantics. It is a core technique for software verification, compiler optimization, and security analysis, where it is used to establish memory safety guarantees, infer numerical invariants, and detect unsafe information flows. A central challenge is the precision-scalability trade-off: coarse abstractions scale but generate many false alarms, while highly precise methods often struggle on large codebases. This thesis develops sound and automated static analyses in the context of abstract interpretation, to achieve sufficient precision for formal verification while maintaining scalability on practical verification tasks.

The thesis presents three contributions. First, for memory safety verification, which requires inferring invariants to prove the validity of spatial memory accesses (i.e., that accessed buffer offsets remain within the bounds of their allocated buffer), we build a new abstract domain for reasoning about object invariants, where memory objects sharing common properties are abstracted relationally (i.e., summarized into a single abstract representation). However, such summarization causes precision loss when the properties of the summarized objects are temporarily violated during updates. To mitigate this, we introduce a new memory abstraction that separates recently manipulated objects from unchanged objects (i.e., summary objects). The new domain follows this abstraction and is designed as a reduced product of subdomains to capture properties of objects and scalars modularly for scalability, and incorporates a reduction process to exchange information between subdomains for precision. Second, we propose a new numerical domain, Template Difference-Bounded Matrices (tDBM), to capture a useful subset of Two Variable Per Inequality (TVPI) constraints needed for bounds checking. By extending the matrix with additional dimensions via ghost variables that represent variables with scaled coefficients, tDBM provides a practical middle ground between lightweight weakly relational domains and expensive fully relational domains. Third, for information-flow security, we present a field-sensitive taint analysis that combines data-structure analysis and data-flow analysis, and further integrates abstract-interpretation-based value reasoning to prune infeasible flows and reduce false positives. To improve precision in heap reasoning, we also introduce a refined variant of data-structure analysis that better preserves field sensitivity while retaining scalability.

Overall, this thesis demonstrates that carefully designed abstract domains and their combinations can deliver analyses that remain sound, practical, and effective on real-world programs for memory safety and information-flow security tasks.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Arie Gurfinkel, for his patience, guidance, and unwavering support throughout my PhD journey. Thank you for your thoughtful mentorship and for guiding me, step by step, toward becoming a scholar. Without your guidance, I would not have developed the academic judgment, research skills, and confidence that shaped this dissertation and the work behind it.

I am also sincerely grateful to my other committee members, Werner Dietl, Patrick Lam, Richard Trefler, and Antoine Miné, for their time, careful reading of this thesis, constructive feedback, and valuable advice. Their insights have helped me improve both this dissertation and my research more broadly.

I would like to give special thanks to my collaborator, Jorge A. Navas. Without your work on the Crab library, I would not have been able to complete many of the projects in this thesis. I am deeply grateful for your generous support throughout my PhD, from answering my questions to supporting the theoretical development, implementation, and evaluation of each project.

I am also grateful to everyone I met at Waterloo. Each of you has been kind, supportive, and generous with your knowledge, creating a truly collaborative and welcoming research environment. I would first like to thank Isabel Garcia-Contreras, who helped me complete my first project and taught me how to write a solid research paper. I also thank Siddharth Priya for years of collaboration and for sharing both academic and engineering knowledge with me. Thank you to Joseph Elias Tafese for showing me what it means to approach research with passion and enthusiasm; to Hari Govind Vediramana Krishnan for helping me improve my communication skills during the early stages of my PhD; to Nham Le and Estifanos Getachew for broadening my perspective beyond my own research area; to Kevin Lee for helping me appreciate the many changes brought by AI; to Xiang Zhou for the joy of learning and collaborating together in the early stages of my PhD; to Yudi (Billy) Bai for generous support and help with mathematics; and to Zhengyang (John) Lü and Aosen Xiong for sharing knowledge, stories, and perspectives beyond my own research field.

Finally, I would like to thank my wife, Xizi (Lucy) Wang, for being by my side over the past ten years, from when we first met in an Advanced Programming Principles lecture to the life we share today. I am deeply grateful for your love, companionship, and support throughout this long journey. I also thank my parents for their unconditional love and for raising, supporting, and caring for me. Lastly, I thank myself for continuing to embrace challenges, to grow, and to move forward.

Dedication

Dedicated to my parents and my wife, for their unconditional love.

吾有知乎哉？无知也。有鄙夫问于我，空空如也，我叩其两端而竭焉。

---《论语·子罕》

“Confucius said, ‘Am I possessed of knowledge? I am not.
But if a layperson, who appears quite empty-like, asks anything of me,
I set it forth from one end to the other, and exhaust it.’ ”

The Analects, Book IX

Table of Contents

Examining Committee ii

Author's Declaration iii

Statement of Contributions iv

Abstract v

Acknowledgements vi

Dedication vii

List of Figures xi

List of Tables xiii

1 Introduction 1
- 1.1 Key Strategies for Improving Software Reliability 1
- 1.2 Motivation: Scalable Yet Precise Static Analysis 3
- 1.3 Contributions of the thesis 6
- 1.4 Overview of the thesis 7

2 Static Analysis by Abstract Interpretation 8
- 2.1 Notation and Basic Definitions 8
- 2.2 Programs and Semantics 12
- 2.3 Abstract Numerical Semantics 19
  - 2.3.1 Numerical Analysis by Interval Abstraction 20
  - 2.3.2 Improving Precision by Combining Abstractions 30
  - 2.3.3 Numerical Anlysis: Precision vs. Efficiency Trade-offs 34
- 2.4 Conclusion 35

3 A New Abstract Domain for Relational Object Invariants 37
3.1 Introduction . . . 38
3.2 Preliminaries . . . 40
3.2.1 `CrabIR`: An IR for Inferring Object Invariants . . . 41
3.2.2 Lightweight Equality Abstract Domain . . . 42
3.3 Recent-Use Memory Model . . . 46
3.4 MRUD: Object Invariant Abstract Domain . . . 51
3.5 Implementation . . . 59
3.6 Evaluation . . . 60
3.7 Related Works . . . 65
3.8 Conclusion . . . 67

4 Template DBM: A New Weakly Relational Domain 68
4.1 Introduction . . . 68
4.2 Difference Bound Matrice and Zones . . . 71
4.3 Template DBMs . . . 75
4.3.1 Saturation . . . 77
4.3.2 Incremental Saturation . . . 80
4.4 Template DBM Abstract Domain . . . 82
4.5 Implementation and Experimental Evaluation . . . 85
4.6 Related Work . . . 89
4.7 Conclusion . . . 90

5 Taint Analysis via Heap-Aware Propagation 91
5.1 Introduction . . . 92
5.2 Overview . . . 93
5.3 Language and Semantics . . . 95
5.4 Data Structure Analysis with Interval Offsets . . . 98
5.5 Taint Semantics . . . 106
5.6 Implementation . . . 110
5.7 Evaluation . . . 111
5.8 Related Work . . . 114
5.9 Conclusion . . . 115

6 Conclusion and Future Work 116
6.1 Summary . . . 116
6.2 Future Work . . . 118
6.2.1 "Cache" More . . . 119

6.2.2 Toward More Precise Template DBM Operations 120
6.2.3 Better DSA and DFA. 121

References **123**

List of Figures

1.1 An example of a memory safety issue in a C program. 4
1.2 An example of a data leak in a C program. 5

2.1 The Hasse diagrams for (a) $\langle \mathcal{P}(\{x, y, z\}), \subseteq \rangle$, and (b) $\langle \mathbb{N} \cup \{+\infty\}, \leq \rangle$. . . . 10
2.2 The syntax of an imperative language. 13
2.3 (a) A toy program, and (b) its CFG. 14
2.4 Concrete execution paths for Fig. 2.3a, with states shown from L2 onward. 16
2.5 Common numerical domains: expressiveness and operation costs. 34
2.6 Approximating the constraint set with interval, octagon, and polyhedra domains. 35

3.1 A simple C program. 39
3.2 Abstract state on line 3.1:17. 39
3.3 The syntax of `CrabIR`. 41
3.4 Hasse diagram for the `VarEq` over variable set $\mathcal{V} = \{s, t, x, y, z\}$. 44
3.5 The join and meet operations. 45
3.6 The `addEqual` and `equals` operations. 46
3.7 C memory model versus recent-use memory model. 47
3.8 (a) A program, and (b) an execution of line 4 under RUMM. 48
3.9 `CrabIR` statements operating under RUMM. 49
3.10 Cache operations. 50
3.11 (a) Concrete domain and (b) MRUD hierarchy. 51
3.12 Abstract semantic domains. 52
3.13 State at line 7, 2nd iteration. 52
3.14 Abstract transformers for memory operations. 53
3.15 Abstract cache operations. 54
3.16 (a) Generic algorithm for Lattice operations; (b) Cache-flush helper function. 55
3.17 Fixpoint computation for the entry state of the loop in Fig. 3.8a. 56
3.18 Domain reduction. 57
3.19 Scalability results. Summarization refers to $\mathcal{D}_{\mathcal{S}}$ and MRUD to $\mathcal{D}_{\mathcal{O}}$. 60

3.20 Another C program. . . . 61
3.21 The AI4BMC pipeline. . . . 63
3.22 AI4BMC vs. BMC. . . . 64

4.1 An example C program. . . . 69
4.2 (a) a DBM example m, its (b) potential graph, and (c) the normal form after applying closure. . . . 71
4.3 (a) Incremental DBM closure algorithm; (b) Graph-based visualization. . . . 74
4.4 (a) DBM-related constraints and (b) a tDBM $\bar{m}$. . . . 77
4.5 (a) Graph view of the tDBM update after adding $ax - by \leq c$ with new edges highlighted and the green dashed edge marking all implicit constraints between z and w; (b) The lattice operations of Template DBM. . . . 83
4.6 (a) Zones vs. Template DBM and (b) Polyhedra (PK) vs. Template DBM. . . . 87

5.1 (a) A program, and (b) its taint-flow graph over the data structure graph. . . . 94
5.2 The syntax of C-IR. . . . 96
5.3 Concrete collecting semantics (taint rules highlighted for Section 5.5). . . . 97
5.4 Effect of `unifyCells` on the same node: (a) before unify; (b) after unify. . . . 101
5.5 Unification rules (Part I). . . . 102
5.6 Unification rules (Part II). . . . 103
5.7 Effect of `unifyCells` on different nodes: (a) before unify; (b) after unify. . . . 104
5.8 The transfer functions of I-DSA. . . . 105
5.9 Abstract transfer functions $[\![s]\!]^\sharp_T$ for taint analysis. . . . 108
5.10 Overview of the taint-analysis workflow. . . . 110
5.11 Comparison of SeaDSA + DFA (OA) and I-DSA + DFA (IA). . . . 111
5.12 OA vs. IA: (a) without inlining; (b) with inlining. . . . 113

List of Tables

3.1 Precision results. 61
3.2 AI4BMC vs. BMC details. 65

4.1 Precision across Zones, Template DBM (tDBM), and Polyhedra (PK). . . . 88

Chapter 1

Introduction

Software appears everywhere in our lives, powering everything from mobile applications to the complex systems that run power grids, airplanes, and cloud infrastructure. As we depend increasingly on these systems, their reliability is crucial. While the nature of software has evolved over the decades, the fundamental need for reliability has remained unchanged. A single failure can lead to financial damage amounting to billions of dollars and, in some cases, can put human lives at risk. Remarkable examples include the 1988 Morris Worm, which caused the first major Internet outage, where a buffer overflow in the Unix `fingerd` program allowed a self-replicating worm to infect thousands of computers and cause millions of dollars in damage [39]; the 1996 Ariane 5 Flight 501 launch failure, where a floating-point conversion error caused the rocket to explode shortly after liftoff [72]; and the 2021 Meta data exposure, which involved 533 million users and was caused by the exploitation of an insecure contact-importing feature [74, 57]. From a safety perspective, detecting and preventing software failures is essential for the reliability of industrial systems and the security of end users.

1.1 Key Strategies for Improving Software Reliability

In response to these risks, checking whether a software implementation (i.e., a program) satisfies its specification has long been a central goal, ranging from lightweight methods such as software testing to rigorous methods such as formal verification. *Software testing* validates program behavior by executing a program on a set of inputs and comparing the outputs against expected results. Unit testing is one of the most common forms,

as it focuses on isolated components for functional validation. However, it requires substantial human effort to generate test inputs manually and to write test oracles. More automated testing techniques, such as fuzzing tools AFL++ [42] and LibFuzzer [73], mitigate this burden by generating inputs randomly. Property-based testing, pioneered by QuickCheck [18], further extends this approach by specifying preconditions and postconditions in test harnesses and leveraging fuzzers for automated testing. While testing with finite inputs can show the presence of bugs, as Edsger Dijkstra noted [36], it cannot demonstrate their absence. This means that even if a program passes all tests, testing still cannot guarantee that the program exactly satisfies its specification, that is, that the program is *correct*.

In contrast, *formal verification* proves or disproves the correctness of software with respect to its specification. It leverages *formal methods*, that is, mathematical techniques, to establish formally that a program satisfies its specification. Existing methods can be categorized as follows:

- **Deductive methods** aim to prove the functional correctness of a program given a formal specification written in a domain-specific language. By translating annotated programs into verification conditions (proof obligations), they use automated theorem provers such as Z3 [33] or interactive proof assistants such as Rocq [10]. Modern tools such as Dafny [71], Verus [69], and Frama-C [31] have made this approach more practical and more suitable for industrial use. However, such verification remains challenging for non-experts because it often requires expertise in providing auxiliary lemmas or proof guidance, even with the use of large language models (LLMs) for invariant synthesis [109] or agentic proof [102] to reduce this burden.
- **Model checking** [21, 20] is an automated verification technique that aims to determine whether specified properties hold for a program represented as a formal state model. If a property is violated, a model checker returns a counterexample showing how the violation occurs from an initial state to an error state. After decades of research and development, advanced tools for Bounded Model Checking (BMC), such as CBMC [22] and SeaBMC [89], have been widely used for software verification in C [17, 91] and Rust [107, 103]. Because model checking exhaustively explores the state space of the system, its main challenge is the state explosion problem, where the number of states grows exponentially with the size of the system. Therefore, termination on large systems may not be guaranteed.
- **Formal static analysis** is an automated technique that leverages mathematical

abstractions to prove that specific properties hold across all possible program executions. By analyzing code without executing it, the analysis over-approximates exact program behavior to guarantee that no bugs are missed, that is, there are no false negatives. However, the abstraction may report that a program is unsafe even when it is actually safe, that is, it may produce false positives. The choice of abstraction level determines the precision and scalability of the analysis. An abstract interpretation based analyzer such as ASTRÉE [11] achieves scalable analysis of large code bases, including Airbus flight control software, and can prove the absence of runtime errors with very few false alarms [34]. Similarly, symbolic execution provides a more precise analysis by automatically exploring program paths using symbolic inputs. Tools such as KLEE [12] and DART [48] have been widely used for systematic bug finding and test generation. However, while such precise analyses can be effective, they may also encounter state explosion due to exhaustive path exploration and therefore may fail to terminate.

In this thesis, we focus on abstract interpretation as a foundation for developing sound and automated formal methods for static analysis. Abstract interpretation [25] is a theory of sound approximation of program semantics. It provides a framework for developing static analyses that infer program properties, such as variable bounds, that are guaranteed to hold at specific program points across all possible executions, by mapping concrete program semantics to a simpler abstract model, namely an *abstract domain*. With abstract domains and a specialized operator, *widening*, the theory ensures termination by forcing fixedpoint computation to converge in finite time. In Chapter 2, we illustrate the concept of abstract interpretation with formal definitions and examples.

Abstract interpretation allows us to derive different levels of abstraction. A simple example is reasoning about the values of program variables: an analysis may either track value ranges independently (non-relationally) or express relations between variables (relationally). This flexibility allows abstractions to be tailored to specific verification requirements. However, different levels of abstraction introduce an inherent trade-off. With a coarser abstraction, a static analyzer may analyze a program efficiently but generate more false positives, that is, be less precise. With a more precise abstraction, the analysis may become inefficient and fail to scale. In this thesis, we study this trade-off in two important verification tasks: memory safety and taint tracking.

1.2 Motivation: Scalable Yet Precise Static Analysis

```c
void _libssh2_packet_add(unsigned char *data, size_t datalen) {
 // datalen: [1, DMAX]
 // where DMAX is the maximum packet size. We assume DMAX = 50.
 uint32_t len = 0;
 unsigned char want_reply = 0;
 if (datalen >= 9) {
   // datalen: [9, 50]
   len = read_u32(data + 5);
   // datalen: [9, 50], len: [0, UINT_MAX]
   if ((len + 9) < datalen)
     // datalen: [9, 50], len: [0, 40]
     want_reply = data[len + 9]; // safe access.
   // datalen: [9, 50], len: [0, UINT_MAX]
   read(data + 9, len); // OOB access when len > 40.
 }
}
```

Figure 1.1: An example of a memory safety issue in a C program.

Memory safety is one of the key verification tasks. Microsoft has reported that roughly 70% of vulnerabilities are rooted in memory-safety issues [76]. Consider a real-world C example[1], shown in Fig. 1.1, where an inconsistency involving `len` allows unconstrained values to be read from the `data` buffer, which can further cause an out-of-bounds read. Such failures are common in C programs, which provide no inherent memory safety guarantees. Approaches for detecting these vulnerabilities include manual code inspection, memory sanitizers that track uninitialized memory, and automated verification tools that report alarms or counterexamples to reproduce bugs. For formal verification, we first focus on:

- **Spatial Memory safety.** Build a sound static analysis that detects all potential out-of-bounds accesses, with a formal guarantee that no overflow is missed.
- **Analysis expectation.** As shown in the comments in Fig. 1.1, the analysis should infer variable ranges so that it can detect the potential out-of-bounds access in `read`.

Security vulnerabilities also arise from data dependencies, where untrusted inputs (i.e., tainted data) flow into sensitive operations. Such dependency flaws, as seen in the 2017 Equifax data breach [106], remain a critical security threat. In that case, attackers used malicious HTTP headers to send code disguised as text, tricking the server into executing

[1] Accessed from https://github.com/libssh2/libssh2/issues/1815.

```c
typedef struct GPLOT {
    char *cmdname;
} GPLOT;

void gplotMakeOutput(GPLOT *gplot) {
  char cmd[512];
  sprintf(cmd, 'gnuplot %s', gplot->cmdname); // field -> cmd
  system(cmd);                                // sink
}

int main(int argc, char **argv) {
  GPLOT gplot;
  if (argc < 2) return 1;
  gplot.cmdname = argv[1];                   // source -> field
  gplotMakeOutput(&gplot);
  return 0;
}
```

Figure 1.2: An example of a data leak in a C program.

arbitrary commands. A similar scenario appears in the code example shown in Fig. 1.2[2], where user-controlled input is stored in the `cmdname` field of a `GPLOT` struct and is later used in a system call. From a program analysis perspective, this is naturally a dataflow problem in which we track dependencies from sources to sinks. In this case, we study:

- **Taint tracking.** Build a sound static analysis that tracks the flow of untrusted data from sources to sensitive sinks, with a formal guarantee that no potential data leak is missed.
- **Analysis expectation.** Running the analysis on Fig. 1.2 should track taint propagation and detect the potential data leak.

These examples highlight the need for sound static analysis. Our goal is to design practical analyses based on abstract interpretation for memory safety and taint tracking while remaining scalable to large code bases. Thus, the final goal is:

- **Scalable yet precise analysis.** Design an abstract interpretation based analysis that strikes a balance between scalability and precision, enabling efficient analysis while reducing false positives.

[2]Publicly available at https://github.com/DanBloomberg/leptonica/issues/303.

- **Analysis expectation.** Achieve reasonable running time with a low false positive rate.

1.3 Contributions of the thesis

This thesis makes the following contributions:

- In Chapter 3, we introduce a new memory model that organizes memory objects into *banks*, each with a single slot most-recently-used (MRU) *cache*. This cache acts as a temporary buffer: when an object is pulled from a bank for modification, it is stored in the cache and then written back when the operations complete. This architecture enables a new abstraction: all objects within a bank are abstracted as a single *summary* object, while the cache retains a precise view of the MRU object. This abstraction improves **precision** by allowing the analysis to model object invariants on the MRU object concretely during updates, while maintaining summary object invariants abstractly.

- Building on this abstraction, we introduce a new abstract domain for reasoning about relational object invariants to support **memory safety** checks. The hierarchy of this domain follows the memory model, so it decomposes into subdomains that reason independently about object invariants in each memory bank. The analyzer can perform targeted manipulations on relevant subdomain elements while leaving the remaining elements unchanged, thereby improving **scalability**.

- In Chapter 4, we examine the use of Two Variable Per Inequality (TVPI) constraints, defined as $a * x - b * y \leq c$, to express **memory safety** properties. This form is well-suited for modeling array accesses; for example, $4 * x - y \leq -4$ ensures that an access at index x remains within the bounds of an integer array of size y. However, since full TVPI expressiveness is unnecessary for this goal, we leverage the Difference Bound Matrix (DBM) used in the Octagons domain. By extending the matrix dimensions with ghost variables (e.g., representing the scaled term $4 * x$ as a variable $4x$), we can efficiently encode memory safety properties without the overhead of general linear constraints.

- Following this idea, we introduce a Template DBM numerical domain for expressing a subset of TVPI constraints, which is more **efficient** than general TVPI or Polyhedra domains, while still being **precise** for memory safety checks.

- In Chapter 5, we present a **taint analysis** that maps how instructions access specific object fields. Our approach achieves field sensitivity by leveraging Data Structure Analysis (DSA) to infer which memory locations each instruction accesses, allowing our subsequent Data Flow Analysis (DFA) to track data propagation across memory locations. Because DSA is flow-insensitive and array-insensitive, it often loses field sensitivity when programs use C structs with flexible array members or pointers that may target multiple memory locations; this, in turn, makes DFA-based taint tracking imprecise. To mitigate this issue, we introduce a new DSA variant that improves field-sensitivity **precision** while maintaining the same level of **scalability** as the original analysis. Furthermore, we integrate DFA with an abstract interpretation based value analysis to further refine taint tracking, allowing the analysis to prune unreachable paths and reduce false positives.

1.4 Overview of the thesis

The remainder of this thesis is organized as follows. In Chapter 2, we introduce the foundations of abstract interpretation, including the mathematical preliminaries, core definitions, and examples used throughout the thesis. In Chapter 3, we present a new abstract domain for reasoning about relational object invariants based on a memory abstraction, and we evaluate its scalability and precision for memory-safety analysis. In Chapter 4, we introduce Template DBM, a new weakly relational numerical domain that captures a useful subset of TVPI constraints for memory-safety verification while remaining efficient in practice. In Chapter 5, we present a taint analysis that follows explicit data dependencies and develops a field-sensitive approach based on data-structure and data-flow analyses. Finally, Chapter 6 concludes the thesis and discusses directions for future work.

Chapter 2

Static Analysis by Abstract Interpretation

In this chapter, we introduce the foundations needed for the rest of the thesis. We begin in Section 2.1 with notation and basic mathematical concepts, including sets, relations, partial orders, lattices, and fixpoints. We then formalize a simple imperative language and its concrete semantics in Section 2.2, and use this formalization to motivate abstract interpretation. Then, in Section 2.3, we give abstract numerical semantics through interval abstraction (Section 2.3.1), together with soundness and termination arguments based on Galois connections and widening. Finally, we discuss how combining abstractions can improve precision in Section 2.3.2 and summarize precision-efficiency trade-offs among common numerical domains in Section 2.3.3.

2.1 Notation and Basic Definitions

We begin with the mathematical primitives needed to present the abstract interpretation framework. The definitions in this section provide the mathematical prerequisites for the remainder of the thesis.

A set X is a collection of elements. If an element x belongs to X, we write $x \in X$; otherwise, we write $x \notin X$. The number of elements in X (its cardinality) is denoted by $|X|$. An empty set has no elements, i.e., $|X| = 0$. A set is a singleton if it contains exactly one element. A set can be defined either by explicitly listing its elements, e.g., $\{x, y, \ldots\}$, or by a predicate, e.g., $\{x \mid P(x)\}$. A set X is a subset of Y, denoted by $X \subseteq Y$, if and

Example 2.1.1 (A variable set example).
Let $\mathcal{V} = \{x, y, z\}$ be a finite set of variables. Variables x and y belong to $\mathcal{V}$, while w does not. The set has $|\mathcal{V}| = 3$ elements. The set $\{x, z\}$ is a subset of $\mathcal{V}$, and $\mathcal{P}(\mathcal{V})$ contains all possible subsets, including $\emptyset$, $\{x\}$, $\{y\}$, $\{z\}$, $\{x, y\}$, $\{x, z\}$, $\{y, z\}$, and $\{x, y, z\}$. The union $\{x, y\} \cup \{y, z\}$ yields $\{x, y, z\}$, while $\{x, z\} \cap \{y\} = \emptyset$. Also, $\bigcup_{X \in \mathcal{P}(\mathcal{V})} X = \mathcal{V}$ and $\bigcap_{X \in \mathcal{P}(\mathcal{V})} X = \emptyset$.

Example 2.1.2 (Integer set examples).
For integer values, we use $\mathbb{Z}$ to denote the set of integers. Natural numbers are denoted by $\mathbb{N}$, with $\mathbb{N} \subseteq \mathbb{Z}$. We denote the sets of positive and negative numbers by $\mathbb{N}^+$ and $\mathbb{N}^-$, respectively.

only if $\forall x \in X.\ x \in Y$. The *powerset* of X is defined as $\mathcal{P}(X) \stackrel{\text{def}}{=} \{Y \mid Y \subseteq X\}$. The empty set is a subset of every set, i.e., $\emptyset \subseteq X$.

Sets support standard operations. Union combines elements from sets, while intersection keeps only common elements. We write these as $X \cup Y$ and $X \cap Y$, or, for a family of sets M, as $\bigcup_{X \in M} X$ and $\bigcap_{X \in M} X$. Set difference removes from one set the elements that appear in another and is defined as $X \setminus Y \stackrel{\text{def}}{=} \{x \mid x \in X \land x \notin Y\}$.

Multiple sets such as X, Y, and Z can be combined through ordered tuples: $X \times Y \times Z \stackrel{\text{def}}{=} \{(x, y, z) \mid x \in X \land y \in Y \land z \in Z\}$. This operation is called the *Cartesian product* of X, Y, and Z.

We define a binary relation R over a set X as a set of ordered pairs xRy where $x, y \in X$. Thus, $R \subseteq X \times X$. The following properties are standard:

1. **Reflexive:** $\forall x \in X$, we have xRx.
2. **Antisymmetric:** $\forall x, y \in X$, if xRy and yRx, then $x = y$.
3. **Transitive:** $\forall x, y, z \in X$, if xRy and yRz, then xRz.
4. **Symmetric:** $\forall x, y \in X$, xRy if and only if yRx.

We say that R is an equivalence relation if it is reflexive, transitive, and symmetric. We say that R is a partial order if it is reflexive, transitive, and antisymmetric.

Example 2.1.3 (Equivalence relation example).
Let $\mathcal{V} = \{x, y, z\}$ and a valuation map satisfy $\sigma(x) = 6$, $\sigma(y) = 9$, and $\sigma(z) = 6$. The equivalence relation induced by this valuation is $R_\sigma \subseteq \mathcal{V} \times \mathcal{V} = \{(x, x), (y, y), (z, z), (x, z)\}$. Note that $(x, z) \iff (z, x)$.

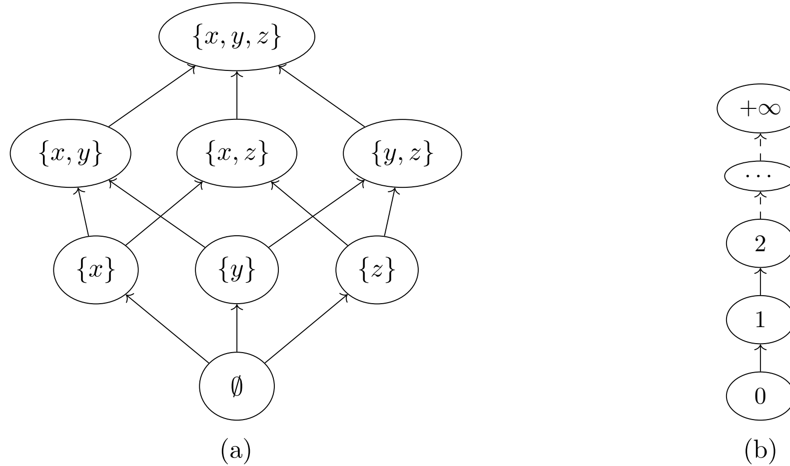


Figure 2.1: The Hasse diagrams for (a) $\langle \mathcal{P}(\{x, y, z\}), \subseteq \rangle$, and (b) $\langle \mathbb{N} \cup \{+\infty\}, \leq \rangle$.

As discussed above, a set X has a powerset $\mathcal{P}(X)$. Elements of the powerset are related by subset inclusion, which is a binary relation between sets. Specifically: (1) reflexive: $X \subseteq X$; (2) antisymmetric: if $X, Y \in \mathcal{P}(X)$ with $Y \subseteq X$ and $X \subseteq Y$, then $Y = X$; and (3) transitive: if $Y \subseteq Z$ and $Z \subseteq X$, then $Y \subseteq X$. Therefore, $\mathcal{P}(X)$ with $\subseteq$ forms a partially ordered set. We define a *partially ordered set* (or poset) formally as follows:

Definition 2.1.1 (Poset)**.** *Given a set D and a binary relation $\sqsubseteq$, a partially ordered set (poset) is an ordered pair $\langle D, \sqsubseteq \rangle$, where $\sqsubseteq$ meets Reflexive, Antisymmetric, and Transitive.*

The pair $\langle \mathcal{P}(X), \subseteq \rangle$ is one example of a poset. A poset can be visualized using a *Hasse diagram*, where lower nodes represent smaller elements and upper nodes represent larger elements.

A poset $\langle D, \sqsubseteq \rangle$ is a *lattice* $\langle D, \sqsubseteq, \sqcup, \sqcap \rangle$ when it is equipped with join $\sqcup$ and meet $\sqcap$, and for all $x, y \in D$, both $x \sqcup y$ and $x \sqcap y$ belong to D. The join operation $\sqcup$ computes the least upper bound of x and y, denoted by $x \sqcup y$ (equivalently $\sqcup\{x, y\}$). The meet operation $\sqcap$ computes the greatest lower bound.

Example 2.1.4 (Complete lattice examples).
A finite variable set $\mathcal{V} = \{x, y, z\}$ has powerset $\mathcal{P}(\mathcal{V})$, which forms a complete lattice under set inclusion $\subseteq$. The join operator is set union $\cup$, and the meet operator is set intersection $\cap$. The least element is $\emptyset$, and the greatest element is $\mathcal{V}$. The corresponding Hasse diagram is shown in Fig. 2.1a.
Another complete lattice is $\langle \mathbb{N} \cup \{+\infty\}, \leq, \max, \min, 0, +\infty \rangle$, shown in Fig. 2.1b. This poset extends the natural numbers $\mathbb{N}$ with a greatest element $+\infty$ and least element 0, ordered by $\leq$.

A lattice is a *complete lattice* $\langle D, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ if: (1) for every subset $X \in \mathcal{P}(D)$, the least upper bound $\sqcup X$ exists in D; and (2) there exist a greatest element $\top$ (supremum) and a least element $\bot$ (infimum).

A poset can serve as a *domain* whose elements represent pieces of information, while the order relation captures refinement: $d_1 \sqsubseteq d_2$ means that d_2 is at least as informative (precise) as d_1. A function that maps domain elements to domain elements is called a *transformer*; it models how information evolves through computation. Static program analysis follows this view: domain elements represent properties at program points, and transformers compute successive approximations of these properties. Formally, given a poset $\langle D, \sqsubseteq \rangle$, a transformer is a function $F : D \to D$. Repeated application is written as $F^n = \underbrace{F \circ F \circ \cdots \circ F}_{n \text{ times}}$. The transformer is required to be *monotone*: for all $d, d' \in D$, if $d \sqsubseteq d'$, then $F(d) \sqsubseteq F(d')$.

The computation aims to reach a stable result where applying F no longer changes the current element. Given a sequence $d_0, d_1, \ldots$ with $d_i \in D$ for all $i \geq 0$ and $d_{k+1} = F(d_k)$, we seek an index k such that $d_{k+1} = d_k$. Such an element is a *fixpoint*. In program analysis, a fixpoint represents an *invariant* that is stable under the transformer. Because a monotone function may have multiple fixpoints, the *least fixpoint*, denoted $\mathsf{lfp}\ F$, is the smallest one under $\sqsubseteq$ and represents the most precise information justified by F.

However, for an arbitrary poset and function, existence and computability of the least fixpoint are not automatic. Two classical results provide these guarantees. Tarski's Fixpoint Theorem (Theorem 2.1.1) states that a monotone function on a complete lattice has a least fixpoint, but it is non-constructive. Kleene's Fixpoint Theorem (Theorem 2.1.2) provides a constructive characterization by iterating F from the least element $\bot$.

Theorem 2.1.1 (Tarski Fixpoint Theorem [104])**.** *Let $\langle D, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ be a complete lattice and let $F : D \to D$ be monotone. Then the set of fixpoints of F is a non-empty*

Example 2.1.5 (Transformer example and its fixpoints).
Consider a complete lattice $\langle \mathcal{P}(\mathbb{N}), \subseteq, \emptyset, \mathbb{N}, \cup, \cap \rangle$ and a continuous function $F : \mathcal{P}(\mathbb{N}) \rightarrow \mathcal{P}(\mathbb{N})$ defined by:

$$F(X) = \{0\} \cup \{n + 1 \mid n \in X\}$$

Starting from the least element of the lattice, we obtain an ascending chain: $F(X_0) = X_1$, $F(X_1) = X_2$, and so on, where $X_0 = \emptyset$, $X_1 = \{0\}$, and $X_2 = \{0, 1\}$, with $X_0 \subseteq X_1 \subseteq X_2 \subseteq \cdots$. By induction, the n-th iterate is $F^n(\emptyset) = X_n = \{0, 1, \ldots, n-1\}$. According to Theorem 2.1.2, the least fixpoint is:

$$\mathsf{lfp}\ F = \bigcup_{n \geq 0} X_n = \{0, 1, 2, \ldots\} = \mathbb{N}$$

Since $\mathsf{lfp}\ F = \mathbb{N}$ (the top element of the lattice), $\mathbb{N}$ is the unique fixpoint for F.

complete lattice (under $\sqsubseteq$). In particular, there exists a least fixpoint $\mathsf{lfp}\ F$ *following:*

$$\mathsf{lfp}\ F = \bigsqcap \{x \in D \mid F(x) \sqsubseteq x\}.$$

Theorem 2.1.2 (Kleene Fixed-Point Theorem [65])**.** *Let $\langle D, \sqsubseteq, \sqcup, \bot \rangle$ be a chain-complete poset, which is a poset with a least element $\bot$ where every chain $C \subseteq D \stackrel{\text{def}}{=} \{d \in D \mid \forall x, y \in C : (x \sqsubseteq y) \vee (y \sqsubseteq x)\}$ has a least upper bound $\bigsqcup C \in D$. Let $F : D \rightarrow D$ be a Scott-continuous function (monotone and $F(\bigsqcup C) = \bigsqcup \{F(d) \mid d \in C\}$ for every ascending chain C). Then the least fixpoint of F exists and is given by:*

$$\mathsf{lfp}\ F = \bigsqcup_{i \in \mathbb{N}} F^i(\bot).$$

2.2 Programs and Semantics

In this chapter, we assume that the input program for analysis is written in a simple imperative language with no side effects, no functions, and only atomic statements. Its syntax is shown in Fig. 2.2. Any program in this language contains only integer variables, whose values range over $\mathbb{Z}$, and uses a fixed finite set of variables $\mathcal{V}$.

$$
\begin{array}{rlll}
\mathsf{P} \ni pg & ::= s^* & & \text{(Program)} \\
\mathsf{S} \ni s & ::= s;\ s & \mid\ i = a & \text{(Sequencing \& Int Arithmetic)} \\
& \quad \mathsf{if}(b)\ \mathsf{then}\ \{\ s\ \}\ \mathsf{else}\ \{\ s\ \} & \mid\ \mathsf{while}(b)\ \{\ s\ \} & \text{(Control Flow)} \\
\mathsf{A} \ni a & ::= c \in \mathsf{Const}\ \mid\ i \in \mathcal{V}\ \mid\ \mathsf{nd}(c_1, c_2) & \mid\ a\ \mathsf{op}_{int}\ a & \text{(Int Expressions)} \\
\mathsf{B} \ni b & ::= a\ \mathsf{op}_{cmp}\ a\ \mid\ \neg\ b & \mid\ b\ \mathsf{op}_{logic}\ b & \text{(Boolean Expressions)} \\
\mathsf{op}_{int} & ::= +\ \mid\ -\ \mid\ * & & \text{(Int Operators)} \\
\mathsf{op}_{cmp} & ::= \ <\ \mid\ <=\ \mid\ >\ \mid\ >=\ \mid\ ==\ \mid\ != & & \text{(Boolean Operators)} \\
\mathsf{op}_{logic} & ::= \wedge\ \mid\ \vee & & \text{(Logical Operators)}
\end{array}
$$

Figure 2.2: The syntax of an imperative language.

Example 2.2.1 (A toy program).
A small program is shown in Fig. 2.3a. The program initializes `x` to `0` and nondeterministically assigns `c` to either `0` or `1`. The `while` loop repeatedly increments `x` by either `2` or `4` and toggles `c` each time until `x` exceeds `7`. After the loop, `x` is expected to be either `8` or `10`. The corresponding control-flow graph (CFG) is shown in Fig. 2.3b.

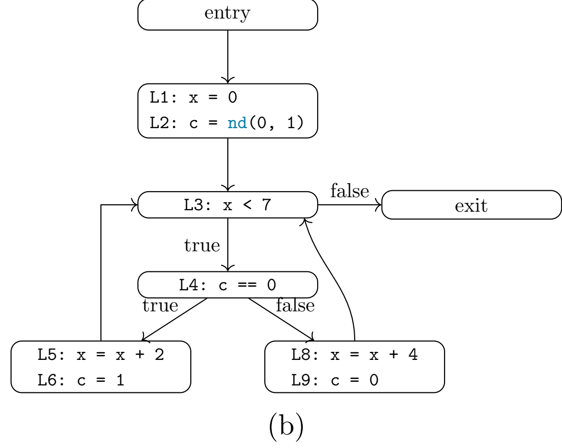
A program pg consists of a sequence of statements. Each statement is either an assignment, a sequential composition, a conditional statement of the form $\mathsf{if\text{-}then\text{-}else}$, or a while loop. An assignment statement assigns to an integer variable i the value of an expression a. Such an expression may be a constant c, another integer variable, a nondeterministic number generator nd with given lower and upper bounds, or an arithmetic expression built using an operator op_{int}. A Boolean condition b may compare two integer expressions using a comparison operator op_{cmp}, combine conditions using logical operators op_{logic}, or negate a condition.

An execution of a program is modeled as a sequence of program states. Intuitively, a state captures the runtime information available at a given point during execution. The semantics of a single computation step is given by a state transition relation, which describes how execution may proceed from one state to another.

To formalize program behavior, we define a state as an *environment map* from variables to integers. Formally, $\sigma \in \mathsf{State} : \mathcal{V} \to \mathbb{Z}$. Updating a state by assigning a value $val \in \mathbb{Z}$ to a variable i is written as $\sigma[i \mapsto val]$. For Boolean expressions, we do not introduce additional variables to store truth values; instead, we evaluate them directly as predicates over the program state. Before execution, we assume an initial state in which all variables

```
1  x = 0;
2  c = nd(0, 1);
3  while (x < 7) {
4    if (c == 0) {
5      x = x + 2;
6      c = 1;
7    } else {
8      x = x + 4;
9      c = 0;
10   }
11 }
12 // x is even after the loop
```

(a)



(b)

Figure 2.3: (a) A toy program, and (b) its CFG.

Example 2.2.2 (Program state).
Following Example 2.2.1, once the variables are initialized on line 2.3a:2, the program state σ_0 is $[x \mapsto 0, c \mapsto 0]$. Alternatively, depending on the non-deterministic assignment, σ_0 could also be $[x \mapsto 0, c \mapsto 1]$.

are undefined.

A single step of execution is modeled by a *state transformer*, that is, a function $[\![\cdot]\!] : \mathsf{S} \times \mathsf{State} \to \mathsf{State}$ that takes a statement s and an input state σ, and returns the resulting output state. Before defining this transformer, we first specify how integer and Boolean expressions are evaluated.

For an integer expression a, evaluation is defined by a function $[\![\cdot]\!]_{\mathsf{A}} : \mathsf{A} \times \mathsf{State} \to \mathbb{Z}$ that takes a state and returns an integer. Formally,

$$
\begin{aligned}
[\![c]\!]_{\mathsf{A}}(\sigma) &\stackrel{\text{def}}{=} c \\
[\![i]\!]_{\mathsf{A}}(\sigma) &\stackrel{\text{def}}{=} \sigma(i) \\
[\![\mathsf{nd}(c_1, c_2)]\!]_{\mathsf{A}}(\sigma) &\stackrel{\text{def}}{=} c \in \{x \mid c_1 \leq x \leq c_2\} \\
[\![a_1 \ \mathsf{op}_{int} \ a_2]\!]_{\mathsf{A}}(\sigma) &\stackrel{\text{def}}{=} [\![a_1]\!]_{\mathsf{A}}(\sigma) \ \mathsf{op}_{int} \ [\![a_2]\!]_{\mathsf{A}}(\sigma)
\end{aligned}
$$

Constants evaluate to themselves. Variables are evaluated by looking them up in the state. Compound arithmetic expressions are evaluated recursively by first evaluating their subexpressions and then applying the corresponding operator.

For each condition b, evaluation checks whether the given state satisfies that condition. A function $[\![\cdot]\!]_{\mathsf{B}} : \mathsf{B} \times \mathsf{State} \to \{\mathsf{true}, \mathsf{false}\}$ evaluates Boolean expressions. Formally,

$$
\begin{aligned}
[\![a_1 \ \mathsf{op}_{cmp} \ a_2]\!]_{\mathsf{B}}(\sigma) &\stackrel{\text{def}}{=} [\![a_1]\!]_{\mathsf{A}}(\sigma) \ \mathsf{op}_{cmp} \ [\![a_2]\!]_{\mathsf{A}}(\sigma) \\
[\![\neg b]\!]_{\mathsf{B}}(\sigma) &\stackrel{\text{def}}{=} \neg([\![b]\!]_{\mathsf{B}}(\sigma)) \\
[\![b_1 \ \mathsf{op}_{logic} \ b_2]\!]_{\mathsf{B}}(\sigma) &\stackrel{\text{def}}{=} [\![b_1]\!]_{\mathsf{B}}(\sigma) \ \mathsf{op}_{logic} \ [\![b_2]\!]_{\mathsf{B}}(\sigma)
\end{aligned}
$$

For integer comparisons, the operands are evaluated and the comparison is then checked. For negation, the inner condition is evaluated first and then negated. For compound conditions, the subconditions are evaluated and their Boolean results are combined using the corresponding logical operator.

The state transition for each statement is defined as follows:

$$
\begin{aligned}
[\![i = a]\!](\sigma) &\stackrel{\text{def}}{=} \sigma[i \mapsto [\![a]\!]_{\mathsf{A}}(\sigma)] \\
[\![s_1;\ s_2]\!](\sigma) &\stackrel{\text{def}}{=} [\![s_2]\!]([\![s_1]\!](\sigma)) \\
[\![\mathsf{if}(b) \ \mathsf{then} \ \{s_1\} \ \mathsf{else} \ \{s_2\}]\!](\sigma) &\stackrel{\text{def}}{=} \begin{cases} [\![s_1]\!](\sigma) & \text{if } [\![b]\!]_{\mathsf{B}}(\sigma) = \mathsf{true} \\ [\![s_2]\!](\sigma) & \text{if } [\![b]\!]_{\mathsf{B}}(\sigma) = \mathsf{false} \end{cases} \\
[\![\mathsf{while}(b)\{s\}]\!](\sigma) &\stackrel{\text{def}}{=} \begin{cases} [\![\mathsf{while}(b)\{s\}]\!]([\![s]\!](\sigma)) & \text{if } [\![b]\!]_{\mathsf{B}}(\sigma) = \mathsf{true} \\ \sigma & \text{if } [\![b]\!]_{\mathsf{B}}(\sigma) = \mathsf{false} \end{cases}
\end{aligned}
$$

For atomic statements, which here are only assignments, the transformer evaluates the arithmetic expression a, obtains an integer value, and updates the state σ by mapping the variable i to that result. For compound statements, evaluation is defined recursively. A sequential statement is evaluated as a composition: first S_1, then S_2. For a conditional statement, the transition depends on whether the current state satisfies b. If it does, then S_1 in the `then` branch is evaluated; otherwise, S_2 in the `else` branch is evaluated. For a while loop, the body statement s is repeatedly evaluated until the condition becomes false, and the final state is the one for which the condition no longer holds.

While state transitions describe how a program behaves in a single execution, they do not capture overall behavior of the program. Conditionals and loops create multiple execution paths, especially when the program interacts with external environments such as user input or file contents. To reason soundly about all possible behaviors, we lift individual state transitions to operate on sets of states. This leads to the *reachability semantics*, which captures all possible states reachable at each program point from a given set of initial states.

Example 2.2.3 (Program transition).
Following Example 2.2.1, suppose σ_0 is $[x \mapsto 0, c \mapsto 0]$ before entering the loop. When evaluating the loop condition at line 2.3a:3, the transition evaluates the arithmetic comparison and returns `true`; therefore, the loop body is executed. Since the body contains an if-then-else statement, the transformer evaluates the condition at line 2.3a:4, which returns `true` because $c = 0$. Execution then moves to line 2.3a:5 and updates the state through line 2.3a:6, yielding $\sigma_1 = [x \mapsto 2, c \mapsto 1]$. Next, we return to the loop condition and check whether `x < 7` still holds for σ_1. Since it does, we enter the loop again. This time, the condition at line 2.3a:4 evaluates to false because $c = 1$, so execution moves to line 2.3a:8 and updates the state at line 2.3a:9 to $\sigma_2 = [x \mapsto 6, c \mapsto 0]$. This process repeats until the loop condition is no longer satisfied, resulting in the final state $\sigma_3 = [x \mapsto 8, c \mapsto 1]$. In this example, there are two possible execution paths depending on the initial value of c. All execution paths are shown in Fig. 2.4.

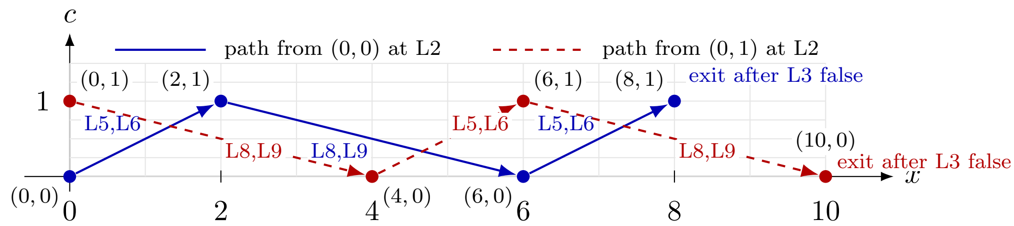


Figure 2.4: Concrete execution paths for Fig. 2.3a, with states shown from L2 onward.

During execution, a program may encounter runtime errors, such as division by zero or access to uninitialized variables. We do not explicitly represent such outcomes as error states. Instead, we focus on the collecting semantics over normal executions only, and exclude executions that fault. This keeps the presentation simple. In practical analyzers, potential errors are still checked, either through explicit assertions (e.g., proving $y \neq 0$ before x/y) or through dedicated safety checks, and are reported as warnings or errors.

The reachability semantics of a statement is defined as the set of all possible successor states produced by applying the transition relation to a given set of input states. Formally, we define a set transformer $[\![\cdot]\!] : \mathsf{S} \times \mathcal{P}(\mathsf{State}) \to \mathcal{P}(\mathsf{State})$ that takes a set of states X before the evaluation of a statement s and returns the set of states after execution. The semantics

Example 2.2.4 (If-then-else semantics).
Following Example 2.2.1, consider the set of initial states $X = \{\{x \mapsto 0, c \mapsto 0\}, \{x \mapsto 0, c \mapsto 1\}\}$ before evaluating the conditional at line 2.3a:4. The condition is $c = 0$.
For the first state $\sigma_a = \{x \mapsto 0, c \mapsto 0\}$, the condition evaluates to true, so the then-branch (lines 5–6) is taken: $x := x + 2; c := 1$, producing $\{x \mapsto 2, c \mapsto 1\}$.
For the second state $\sigma_b = \{x \mapsto 0, c \mapsto 1\}$, the condition evaluates to false, so the else-branch (lines 8–9) is taken: $x := x + 4; c := 0$, producing $\{x \mapsto 4, c \mapsto 0\}$.
Thus, the final reachable states are $\{\{x \mapsto 2, c \mapsto 1\}, \{x \mapsto 4, c \mapsto 0\}\}$.

of each statement is defined as follows:

$$
\begin{aligned}
[\![i = a]\!](X) &\stackrel{\text{def}}{=} \{\sigma' \mid \sigma \in X, \sigma' = [\![i = a]\!](\sigma)\} \\
[\![s_1;\ s_2]\!](X) &\stackrel{\text{def}}{=} [\![s_2]\!]([\![s_1]\!](X)) \\
[\![b]\!]_{\mathsf{B}}(X) &\stackrel{\text{def}}{=} \{\sigma \in X \mid [\![b]\!]_{\mathsf{B}}(\sigma) = \mathsf{true}\} \\
[\![\mathsf{if}(b)\mathsf{then}\{s_t\}\mathsf{else}\{s_f\}]\!](X) &\stackrel{\text{def}}{=} [\![s_t]\!]([\![b]\!]_{\mathsf{B}}(X)) \cup [\![s_f]\!]([\![\neg b]\!]_{\mathsf{B}}(X)) \\
[\![\mathsf{while}(b)\{s\}]\!](X) &\stackrel{\text{def}}{=} [\![\neg b]\!]_{\mathsf{B}}(\mathsf{lfp}\ F)\,,\ \text{where } F \stackrel{\text{def}}{=} (\lambda Y.\ X \cup [\![s]\!]([\![b]\!]_{\mathsf{B}}(Y)))
\end{aligned}
$$

As in the state-transition semantics, the transformer evaluates the corresponding statement for each state in the input set and collects the resulting states into a new set. For Boolean conditions, we define a filter function $[\![b]\!]_{\mathsf{B}}(X)$ that selects the subset of states in X satisfying the condition. For a conditional statement, because input states may satisfy either branch condition, we collect every possible post-state by taking the union of the states produced by the two branches.

Modeling the effect of a while loop, $[\![\mathsf{while}(b)\{s\}]\!](X)$, is more complex than modeling linear statements because its semantics must account for all possible numbers of loop iterations. The transformer must collect every possible post-state produced by the loop, which conceptually amounts to unrolling the loop and repeatedly evaluating its body an arbitrary number of times. To formalize this, we define a function F that takes a set of states Y reaching the loop head, filters them according to the loop condition, processes the filtered states through the loop body, and merges the result with the initial states X from the preceding program point.

F represents one iteration of the loop. Repeated application of F builds increasingly larger sets of reachable states, where each application corresponds to one more level of loop unrolling. Convergence is important because we seek the least fixpoint of F, namely the smallest set of states that is closed under one more loop iteration.

Example 2.2.5 (Loop semantics on an infinite loop).
Consider a loop statement s:

```
1 i = 0;
2 while (2 > 1) { i = 1; }
3 // dead end
```

Suppose $X = \{\{i \mapsto 0\}\}$. Then $\mathsf{lfp}\ F = \bigcup_{n\geq 0} F^n(\emptyset) = \{\{i \mapsto 0\}, \{i \mapsto 1\}\}$. Because the loop condition is always true, no state is reachable after the loop, so $[\![s]\!](X) = \emptyset$.

Example 2.2.6 (Loop semantics).
Returning to the running example in Fig. 2.3a, consider the set of possible input states before entering the loop: $X = \{\{x \mapsto 0, c \mapsto 0\}, \{x \mapsto 0, c \mapsto 1\}\}$. By the definition of F and Theorem 2.1.2, the Kleene iteration sequence is as follows:

- $F(\emptyset) = X$
- $F^2(\emptyset) = X \cup \{\{x \mapsto 2, c \mapsto 1\}, \{x \mapsto 4, c \mapsto 0\}\}$
- $F^3(\emptyset) = F^2(\emptyset) \cup \{\{x \mapsto 6, c \mapsto 0\}, \{x \mapsto 6, c \mapsto 1\}\}$
- $F^4(\emptyset) = F^3(\emptyset) \cup \{\{x \mapsto 8, c \mapsto 1\}, \{x \mapsto 10, c \mapsto 0\}\}$
- $\forall i >= 5 : F^i(\emptyset) = F^4(\emptyset)$.

$\mathsf{lfp}\ F$ computes all states reachable at the loop head after all possible iterations. We then filter this set using the loop exit condition, which yields $\{\{x \mapsto 8, c \mapsto 1\}, \{x \mapsto 10, c \mapsto 0\}\}$.

To compute $\mathsf{lfp}\ F$, we start with $Y = \emptyset$, so that $F(\emptyset) = X$, $F^2(\emptyset) = X \cup [\![s]\!]([\![b]\!]_{\mathsf{B}}(X))$, and so on. We write $F^n(\emptyset)$ for the result after $n-1$ iterations. By Theorem 2.1.2, the least fixpoint is exactly the limit of the sequence $F^n(\emptyset)$, namely $\bigcup_{n\geq 0} F^n(\emptyset)$. Therefore, the states reachable after the loop are those that do not satisfy the loop condition, which we compute as $[\![\neg b]\!]_{\mathsf{B}}(\bigcup_{n\geq 0} F^n(\emptyset))$.

Overall, for a whole program $pg := s$, we begin with a set of initial states $X_0 \in \mathcal{P}(\mathsf{State})$. The reachability semantics of the program is then computed as $[\![s]\!](X_0)$, which represents the transformation of X_0 into the set of all possible terminal states.

2.3 Abstract Numerical Semantics

To compute reachability semantics, we aim to capture the exact behavior of a program. However, although the semantics defined above is precise, the possibility of non-termination in programs with loops leads to an infinite state space. Consequently, computing the exact behavior is in general impossible; indeed, Rice's theorem [93] states that all non-trivial semantic properties of programs are undecidable. Instead, we seek an abstract semantics that is computable and soundly approximates the concrete semantics. Abstract interpretation provides the formal framework for such an abstraction.

To relate concrete semantics to abstract semantics, we introduce two *domains*. The *concrete domain*, $\langle D, \subseteq \rangle$ with $D = \mathcal{P}(\mathsf{State})$, formalizes program reachability semantics as the powerset of all possible program states. This domain captures the set of states reachable at each program point. We then introduce an *abstract domain* $\langle D^\sharp, \sqsubseteq^\sharp \rangle$, modeled as a poset that captures simplified abstract properties of program states. The connection between the two domains is established by an *abstraction function* α, which maps each concrete element to an abstract one, and a *concretization function* γ, which maps each abstract element back to a concrete one. This relationship is formalized as a *Galois connection* (Definition 2.3.1).

Definition 2.3.1 (Galois Connection)**.** *Let two posets $\langle D, \sqsubseteq \rangle$ and $\langle D^\sharp, \sqsubseteq^\sharp \rangle$ correspond to concrete semantic properties and abstract properties, respectively. A pair of functions $\alpha : D \to D^\sharp$ and $\gamma : D^\sharp \to D$ forms a Galois connection $(D, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (D^\sharp, \sqsubseteq^\sharp)$ if and only if:*

$$\forall d \in D, d^\sharp \in D^\sharp : \quad \alpha(d) \sqsubseteq^\sharp d^\sharp \iff d \subseteq \gamma(d^\sharp)$$

In our setting, given a set of states $d \in D$ and an abstract property $d^\sharp \in D^\sharp$, the relation $\alpha(d) \sqsubseteq^\sharp d^\sharp$ means that $d^\sharp$ is a valid *over-approximation* of the set of states. In other words, $d^\sharp$ is no more precise than the ideal abstraction $\alpha(d)$. Conversely, the relation $d \subseteq \gamma(d^\sharp)$ expresses the *concretization* of an abstract property, ensuring that the set of states represented by $d^\sharp$ includes the concrete states d. Together, these two relations ensure that the analysis performed in the abstract semantics is sound: no concrete behavior is lost. Formally, soundness is captured by $\forall d \in D.\ d \subseteq \gamma(\alpha(d))$.

Although establishing a full Galois connection between concrete and abstract semantics is ideal because it yields the best abstract approximation, many abstract domains do not satisfy this requirement. The abstract interpretation framework therefore allows a weaker formulation based only on a concretization function [27]. In this thesis, we follow that practice. It provides the flexibility needed for our analyses while still ensuring soundness.

Example 2.3.1 (Concretization and abstraction example).
Let $C = \{0, 1, 2, 3, 4, 5\}$ be a set of integers. We define the concrete domain as $\langle \mathcal{P}(C), \subseteq \rangle$, and an abstract domain as $\langle A, \sqsubseteq^\sharp \rangle$ for the set $A = \{\bot, \mathsf{Even}, \mathsf{Odd}, \top\}$, with order relation $\bot \sqsubseteq^\sharp \mathsf{Even} \sqsubseteq^\sharp \top$ and $\bot \sqsubseteq^\sharp \mathsf{Odd} \sqsubseteq^\sharp \top$.
The abstraction function $\alpha : \mathcal{P}(C) \to A$ is defined as follows:

$$\forall X \in \mathcal{P}(C), \alpha(X) = \begin{cases} \bot & \text{if } X = \emptyset, \\ \mathsf{Even} & \text{if } X \subseteq \{0, 2, 4\}, \\ \mathsf{Odd} & \text{if } X \subseteq \{1, 3, 5\}, \\ \top & \text{otherwise.} \end{cases}$$

The corresponding concretization function $\gamma : A \to \mathcal{P}(C)$ is defined as: $\gamma(\bot) = \emptyset, \gamma(\mathsf{Even}) = \{0, 2, 4\}, \gamma(\mathsf{Odd}) = \{1, 3, 5\}, \gamma(\top) = C$.
For the concrete set $X = \{3, 5\}$, we have $\alpha(X) = \mathsf{Odd}$. Conversely, $\gamma(\mathsf{Odd}) = \{1, 3, 5\}$, and therefore $X \subseteq \gamma(\alpha(X))$. It is also true that $X \subseteq \gamma(\top)$ implies $\alpha(X) \sqsubseteq^\sharp \top$.

Next, we introduce a simple abstract domain, Interval [23], to illustrate how abstraction and concretization are defined and how an abstract transformer is constructed to compute abstract properties that soundly approximate the concrete transformer.

2.3.1 Numerical Analysis by Interval Abstraction

An abstract domain, $\langle D^\sharp, \sqsubseteq^\sharp, \bot^\sharp, \top^\sharp, \sqcup^\sharp, \sqcap^\sharp \rangle$, is typically a poset equipped with a least element $\bot^\sharp$, a greatest element $\top^\sharp$, a least-upper-bound operator $\sqcup^\sharp$, and a greatest-lower-bound operator $\sqcap^\sharp$. We define an Interval domain, following [23], which is a *non-relational* abstract domain that captures the range of possible (integer) values for each variable independently.

An interval over the extended integers is denoted by:

$$I \in \mathcal{I} = \{[l, u] \mid l \in \mathbb{N} \cup \{-\infty\}, u \in \mathbb{N} \cup \{+\infty\}, l \leq u\}$$

An interval abstractly represents a range of integers $\{n \in \mathbb{N} \mid l \leq n \leq u\}$ with lower bound l and upper bound u. For example, the interval $[0, +\infty]$ represents all natural numbers. We define a complete lattice for intervals $\langle \mathcal{I} \cup \{\bot_\mathcal{I}\}, \sqsubseteq_\mathcal{I}, \bot_\mathcal{I}, \top_\mathcal{I}, \sqcup_\mathcal{I}, \sqcap_\mathcal{I} \rangle$, with a separate $\bot_\mathcal{I}$ as the least element and greatest element $\top_\mathcal{I} \stackrel{\text{def}}{=} [-\infty, +\infty]$. The domain operators are

defined as follows:

$$\begin{aligned} I_1 \sqsubseteq_{\mathcal{I}} I_2 &\overset{\text{def}}{\iff} l_1 \geq l_2 \wedge u_1 \leq u_2 \\ I_1 \sqcup_{\mathcal{I}} I_2 &\overset{\text{def}}{=} [\min(l_1, l_2), \max(u_1, u_2)] \\ I_1 \sqcap_{\mathcal{I}} I_2 &\overset{\text{def}}{=} [\max(l_1, l_2), \min(u_1, u_2)] \end{aligned}$$

For $\bot_{\mathcal{I}}$, $\bot_{\mathcal{I}} \sqsubseteq_{\mathcal{I}} I$, $\bot_{\mathcal{I}} \sqcup_{\mathcal{I}} I = I$, and $\bot_{\mathcal{I}} \sqcap_{\mathcal{I}} I = \bot_{\mathcal{I}}$.

We define the abstraction function $\alpha_{\mathcal{I}} : \mathcal{P}(\mathbb{N}) \to \mathcal{I}$ and concretization function $\gamma_{\mathcal{I}} : \mathcal{I} \to \mathcal{P}(\mathbb{N})$ for intervals as follows:

$$\begin{aligned} \alpha_{\mathcal{I}}(\emptyset) &\overset{\text{def}}{=} \bot_{\mathcal{I}} \\ \forall X \in \mathcal{P}(\mathbb{N}) \setminus \{\emptyset\}.\ \alpha_{\mathcal{I}}(X) &\overset{\text{def}}{=} [\min(X), \max(X)] \\ \gamma_{\mathcal{I}}(\bot_{\mathcal{I}}) &\overset{\text{def}}{=} \emptyset \\ \gamma_{\mathcal{I}}(I) &\overset{\text{def}}{=} \{c \in \mathbb{Z} \mid l \leq c \leq u\} \end{aligned}$$

Theorem 2.3.1 (Soundness of Interval Abstraction)**.** *The interval abstraction is sound by:*

$$\forall X \in \mathcal{P}(\mathbb{N}).\ X \subseteq \gamma_{\mathcal{I}}(\alpha_{\mathcal{I}}(X))$$

Proof. Let $X \in \mathcal{P}(\mathbb{N})$. If $X = \emptyset$, then by definition $\gamma_{\mathcal{I}}(\alpha_{\mathcal{I}}(X)) = \emptyset$, so $X \subseteq \gamma_{\mathcal{I}}(\alpha_{\mathcal{I}}(X))$ holds trivially. Otherwise, $X \neq \emptyset \Rightarrow \alpha_{\mathcal{I}}(X) = [\min(X), \max(X)]$. Take any $x \in X$, it will be in range of the minimum and maximum of X: $\min(X) \leq x \leq \max(X)$. By definition of $\gamma_{\mathcal{I}}$,

$$x \in \{c \in \mathbb{Z} \mid \min(X) \leq c \leq \max(X)\} = \gamma_{\mathcal{I}}(\alpha_{\mathcal{I}}(X)).$$

Since x was arbitrary, $X \subseteq \gamma_{\mathcal{I}}(\alpha_{\mathcal{I}}(X))$ holds. □

The intervals also support arithmetic operators. We define basic arithmetic operations shown in Fig. 2.2 as follows:

$$\begin{aligned} I_1 +_{\mathcal{I}} I_2 &\overset{\text{def}}{=} [l_1 + l_2, u_1 + u_2] \\ I_1 -_{\mathcal{I}} I_2 &\overset{\text{def}}{=} [l_1 - u_2, u_1 - l_2] \\ I_1 *_{\mathcal{I}} I_2 &\overset{\text{def}}{=} [\min(l_1 * l_2, l_1 * u_2, u_1 * l_2, u_1 * u_2), \max(l_1 * l_2, l_1 * u_2, u_1 * l_2, u_1 * u_2)] \end{aligned}$$

Specifically, for arithmetic involving infinities, we define:

$$\forall x \in \mathbb{N} \cup \{+\infty\}.\ x + (+\infty) = +\infty$$
$$\forall y \in \mathbb{N} \cup \{-\infty\}.\ (-\infty) + y = -\infty$$
$$\forall z \in \mathbb{N}.\ z * (\pm\infty) = \begin{cases} \pm\infty & \text{if } z > 0 \\ 0 & \text{if } z = 0 \\ \mp\infty & \text{if } z < 0 \end{cases}$$

In addition and subtraction, infinity absorbs any finite value. There is no undefined behavior involving infinities in these operations. However, we also define multiplication between 0 and ∞, even though it is not standard mathematically because it is an indeterminate form. For simplicity, we denote $I_1\ \mathsf{op}^\sharp_{int}\ I_2$ as the corresponding transfer function. For $\bot_\mathcal{I}$, we define $\bot_\mathcal{I}\ \mathsf{op}^\sharp_{int}\ I = \bot_\mathcal{I}$ for any I.

Building on the abstraction of integers into intervals, we define an abstract element (or abstract state) of the $\mathsf{Interval}$ domain as a mapping from variables to intervals, i.e., $d^\sharp \in D^\sharp = (\mathcal{V} \to \mathcal{I}) \cup \bot^\sharp$. The $\mathsf{Interval}$ domain is a complete lattice with explicit bottom element $\bot^\sharp$ and top element $\top^\sharp \stackrel{\text{def}}{=} \{v \mapsto \top_\mathcal{I} \mid v \in \mathcal{V}\}$. The domain operations are defined as follows:

$$d_1^\sharp \sqsubseteq^\sharp d_2^\sharp \stackrel{\text{def}}{\iff} \forall v \in \mathcal{V}.\ d_1^\sharp(v) \sqsubseteq_\mathcal{I} d_2^\sharp(v)$$
$$d_1^\sharp \sqcup^\sharp d_2^\sharp \stackrel{\text{def}}{=} \lambda v. d_1^\sharp(v) \sqcup_\mathcal{I} d_2^\sharp(v)$$
$$d_1^\sharp \sqcap^\sharp d_2^\sharp \stackrel{\text{def}}{=} \lambda v. d_1^\sharp(v) \sqcap_\mathcal{I} d_2^\sharp(v)$$

For $\bot^\sharp$ and any abstract element $d^\sharp$, $\bot^\sharp \sqsubseteq^\sharp d^\sharp$, $\bot^\sharp \sqcup^\sharp d^\sharp = d^\sharp$, and $\bot^\sharp \sqcap^\sharp d^\sharp = \bot^\sharp$.

The concretization function $\gamma : D^\sharp \to \mathcal{P}(\mathsf{State})$ and abstraction function $\alpha : \mathcal{P}(\mathsf{State}) \to D^\sharp$ are defined as follows:

$$\alpha(\emptyset) \stackrel{\text{def}}{=} \bot^\sharp$$
$$\forall X \in \mathcal{P}(\mathsf{State}) \setminus \{\emptyset\}.\ \alpha(X) \stackrel{\text{def}}{=} \lambda v. \alpha_\mathcal{I}(\{\sigma(v) \mid \sigma \in X\})$$
$$\gamma(\bot^\sharp) \stackrel{\text{def}}{=} \emptyset$$
$$\forall d^\sharp \in D^\sharp \setminus \{\bot^\sharp\}.\ \gamma(d^\sharp) \stackrel{\text{def}}{=} \{\sigma \in \mathsf{State} \mid \forall v \in \mathcal{V}.\ \sigma(v) \in \gamma_\mathcal{I}(d^\sharp(v))\}$$

We have already defined the concrete transformer over program states. The abstract transformer, in contrast, operates on numerical properties of those states. It is defined as a sound approximation of the concrete transformer, as formalized below:

Definition 2.3.1.2 (Sound Abstract Operator)**.** *Let two posets* $\langle D, \subseteq \rangle$ *and* $\langle D^\sharp, \sqsubseteq^\sharp \rangle$ *be given. A function* $f : D \to D$ *has a sound abstract operator* $f^\# : D^\sharp \to D^\sharp$ *if and only if:*

$$\forall d^\sharp \in D^\sharp.\ f(\gamma(d^\sharp)) \subseteq \gamma(f^\#(d^\sharp))$$

In the remainder of this section, we formalize the abstract semantics of the imperative language in Fig. 2.2 by constructing several transformers over the $\mathsf{Interval}$ domain. We proceed compositionally: first we define each abstract transformer, and then we prove a corresponding local soundness theorem relating it to the concrete transformer. Together, these results provide the inductive basis for the soundness of the whole analysis.

We define the abstract transformer for arithmetic expressions, denoted as $[\![\cdot]\!]^\sharp_{\mathsf{A}} : \mathsf{A} \times D^\sharp \to \mathcal{I} \cup \{\bot_\mathcal{I}\}$. The transformer is defined by inductively evaluating arithmetic expressions as follows:

$$\begin{aligned}
[\![c]\!]^\sharp_{\mathsf{A}}(d^\sharp) &\stackrel{\text{def}}{=} [c, c] \\
[\![i]\!]^\sharp_{\mathsf{A}}(d^\sharp) &\stackrel{\text{def}}{=} d^\sharp(i) \\
[\![\mathsf{nd}(c_1, c_2)]\!]^\sharp_{\mathsf{A}}(d^\sharp) &\stackrel{\text{def}}{=} [c_1, c_2] \\
[\![a_1\ \mathsf{op}_{int}\ a_2]\!]^\sharp_{\mathsf{A}}(d^\sharp) &\stackrel{\text{def}}{=} [\![a_1]\!]^\sharp_{\mathsf{A}}(d^\sharp)\ \mathsf{op}^\sharp_{int}\ [\![a_2]\!]^\sharp_{\mathsf{A}}(d^\sharp)
\end{aligned}$$

Theorem 2.3.2 (Local Soundness of Arithmetic Abstract Transformer)**.** *For every arithmetic expression* $a \in \mathsf{A}$ *and abstract state* $d^\sharp \in D^\sharp$*,*

$$\{[\![a]\!]_{\mathsf{A}}(\sigma) \mid \sigma \in \gamma(d^\sharp)\} \subseteq \gamma_\mathcal{I}([\![a]\!]^\sharp_{\mathsf{A}}(d^\sharp)).$$

Proof. We prove by structural induction on a.

For a constant c, the concrete semantics evaluates to c itself, while the abstract semantics evaluates to $[c, c]$. By the definition of $\gamma_\mathcal{I}$:

$$\{[\![c]\!]_{\mathsf{A}}(\sigma) \mid \sigma \in \gamma(d^\sharp)\} = \{c\} \subseteq \gamma_\mathcal{I}([c, c]).$$

For an integer variable i, since $\sigma \in \gamma(d^\sharp)$, by definition of γ, $\sigma(i) \in \gamma_\mathcal{I}(d^\sharp(i))$. Also, $[\![i]\!]^\sharp_{\mathsf{A}}(d^\sharp) = d^\sharp(i)$. Therefore,

$$\{[\![i]\!]_{\mathsf{A}}(\sigma) \mid \sigma \in \gamma(d^\sharp)\} \subseteq \gamma_\mathcal{I}([\![i]\!]^\sharp_{\mathsf{A}}(d^\sharp)).$$

For a nondeterministic expression $\mathsf{nd}(c_1, c_2)$, $[\![\mathsf{nd}(c_1, c_2)]\!]_{\mathsf{A}}(\sigma)$ returns a value in $\{x \mid c_1 \le x \le c_2\}$, while $[\![\mathsf{nd}(c_1, c_2)]\!]^\sharp_{\mathsf{A}}(d^\sharp) = [c_1, c_2]$. Following $\gamma_\mathcal{I}$, the inclusion holds directly.

For general arithmetic expression $a = a_1 \ \mathsf{op}_{int} \ a_2$, by induction hypothesis,

$$\{[\![a_1]\!]_{\mathsf{A}}(\sigma) \mid \sigma \in \gamma(d^\sharp)\} \subseteq \gamma_{\mathcal{I}}([\![a_1]\!]^\sharp_{\mathsf{A}}(d^\sharp)),$$
$$\{[\![a_2]\!]_{\mathsf{A}}(\sigma) \mid \sigma \in \gamma(d^\sharp)\} \subseteq \gamma_{\mathcal{I}}([\![a_2]\!]^\sharp_{\mathsf{A}}(d^\sharp)).$$

Let $x = [\![a_1]\!]_{\mathsf{A}}(\sigma)$ and $y = [\![a_2]\!]_{\mathsf{A}}(\sigma)$ for some $\sigma \in \gamma(d^\sharp)$. Then $x \in \gamma_{\mathcal{I}}([\![a_1]\!]^\sharp_{\mathsf{A}}(d^\sharp))$ and $y \in \gamma_{\mathcal{I}}([\![a_2]\!]^\sharp_{\mathsf{A}}(d^\sharp))$. By soundness of interval arithmetic operators

$$x \ \mathsf{op}_{int} \ y \in \gamma_{\mathcal{I}}\Big([\![a_1]\!]^\sharp_{\mathsf{A}}(d^\sharp) \ \mathsf{op}^\sharp_{int} \ [\![a_2]\!]^\sharp_{\mathsf{A}}(d^\sharp)\Big),$$

the required inclusion follows.

Therefore, the theorem holds for all arithmetic expressions. □

For condition abstraction, we define the abstract transformer $[\![\cdot]\!]^\sharp_{\mathsf{B}} : \mathsf{B} \times D^\sharp \to D^\sharp$, which refines variable intervals in an abstract state according to a Boolean condition. It is defined as follows:

$$\begin{aligned}
[\![a_1 \ \mathsf{op}_{cmp} \ a_2]\!]^\sharp_{\mathsf{B}}(d^\sharp) &\stackrel{\text{def}}{=} \mathsf{refine}[\![a_1 \ \mathsf{op}_{cmp} \ a_2]\!]^\sharp(d^\sharp) \\
[\![\neg b]\!]^\sharp_{\mathsf{B}}(d^\sharp) &\stackrel{\text{def}}{=} [\![\mathtt{dual}(b)]\!]^\sharp_{\mathsf{B}}(d^\sharp) \\
[\![b_1 \ \wedge \ b_2]\!]^\sharp_{\mathsf{B}}(d^\sharp) &\stackrel{\text{def}}{=} [\![b_1]\!]^\sharp_{\mathsf{B}}(d^\sharp) \ \sqcap^\sharp_{\mathcal{I}} \ [\![b_2]\!]^\sharp_{\mathsf{B}}(d^\sharp) \\
[\![b_1 \ \vee \ b_2]\!]^\sharp_{\mathsf{B}}(d^\sharp) &\stackrel{\text{def}}{=} [\![b_1]\!]^\sharp_{\mathsf{B}}(d^\sharp) \ \sqcup^\sharp_{\mathcal{I}} \ [\![b_2]\!]^\sharp_{\mathsf{B}}(d^\sharp)
\end{aligned}$$

For complex expressions such as the one shown in Example 2.3.1.2, refinement requires recursive propagation through the abstract state until the abstract values are tightened. The algorithm can be implemented as described in Section 4.6 of [81]. In this thesis, we model such refinement abstractly as $\mathsf{refine}[\![\cdot]\!]^\sharp : \mathsf{B} \times D^\sharp \to D^\sharp$ when arithmetic comparisons are evaluated. For negation $\neg b$, we use a function $\mathtt{dual} : \mathsf{B} \to \mathsf{B}$ based on De Morgan's laws and relational inversions to convert the Boolean expression internally.

Theorem 2.3.3 (Local Soundness of Condition Abstract Transformer)**.** *For every boolean condition $b \in \mathsf{B}$ and abstract state $d^\sharp \in D^\sharp$,*

$$[\![b]\!]_{\mathsf{B}}(\gamma(d^\sharp)) \subseteq \gamma([\![b]\!]^\sharp_{\mathsf{B}}(d^\sharp)).$$

Proof. We prove by structural induction on b.

For arithmetic comparison $a_1 \ \mathsf{op}_{cmp} \ a_2$, by definition, $[\![b]\!]_{\mathsf{B}}(X)$ filters X to states where the comparison holds, and $[\![b]\!]^\sharp_{\mathsf{B}}(d^\sharp)$ applies refinement through $\mathsf{refine}[\![a_1 \ \mathsf{op}_{cmp} \ a_2]\!]^\sharp(d^\sharp)$. We assume $\mathsf{refine}[\![\cdot]\!]^\sharp$ is a sound operator. Thus the inclusion holds.

Example 2.3.1.2 (Abstract Conditional Refiner Example).
Consider abstract state $d^\sharp = \{x \mapsto [0, 10], y \mapsto [0, 10], z \mapsto [0, 2]\}$ and condition $b : x + y \leq z + 3$. Let $a = x + y$ and $b = z + 3$ to linearize the expression. Refinement proceeds as follows:

Forward: Compute initial abstract values of sub-expressions:

$$b \mapsto d^\sharp(z) +^\sharp [3, 3] = [0, 2] + [3, 3] = [3, 5]$$
$$a \mapsto d^\sharp(x) +^\sharp d^\sharp(y) = [0, 10] + [0, 10] = [0, 20]$$

Backward: Refine sub-expressions under $a \leq b$ using $\sqcap_{\mathcal{I}}$ and inverse arithmetic:

$$a \mapsto d^\sharp(a) \sqcap_{\mathcal{I}} [-\infty, d^\sharp(b).u] = [0, 20] \sqcap_{\mathcal{I}} [-\infty, 5] = [0, 5]$$
$$b \mapsto d^\sharp(b) \sqcap_{\mathcal{I}} [d^\sharp(a).l, +\infty] = [3, 5] \sqcap_{\mathcal{I}} [0, +\infty] = [3, 5]$$

Then refine x, y, and z:

$$z \mapsto d^\sharp(z) \sqcap_{\mathcal{I}} (d^\sharp(b) -^\sharp [3, 3]) = [0, 2] \sqcap_{\mathcal{I}} [0, 2] = [0, 2]$$
$$x \mapsto d^\sharp(x) \sqcap_{\mathcal{I}} (d^\sharp(a) -^\sharp d^\sharp(y)) = [0, 10] \sqcap_{\mathcal{I}} ([0, 5] -^\sharp [0, 10]) = [0, 5]$$
$$y \mapsto d^\sharp(y) \sqcap_{\mathcal{I}} (d^\sharp(a) -^\sharp d^\sharp(x)) = [0, 10] \sqcap_{\mathcal{I}} ([0, 5] -^\sharp [0, 5]) = [0, 5]$$

The refined abstract state is $d^\sharp_{\text{new}} = \{x \mapsto [0, 5], y \mapsto [0, 5], z \mapsto [0, 2]\}$. A second forward-backward pass yields no further change, so a fixpoint is reached.

For negation $\neg b_1$, $[\![\neg b_1]\!]_{\mathsf{B}}(X) = X \setminus [\![b_1]\!]_{\mathsf{B}}(X)$. By abstract semantics, since `dual` applies negation of b_1. By induction hypothesis on b_1:

$$[\![b_1]\!]_{\mathsf{B}}(\gamma(d^\sharp)) \subseteq \gamma([\![b_1]\!]^\sharp_{\mathsf{B}}(d^\sharp))$$

and soundness of `dual`, the inclusion holds.

For logical conjunction $b_1 \wedge b_2$ and disjunction $b_1 \vee b_2$. By induction hypothesis,

$$[\![b_1]\!]_{\mathsf{B}}(\gamma(d^\sharp)) \subseteq \gamma([\![b_1]\!]^\sharp_{\mathsf{B}}(d^\sharp)),$$
$$[\![b_2]\!]_{\mathsf{B}}(\gamma(d^\sharp)) \subseteq \gamma([\![b_2]\!]^\sharp_{\mathsf{B}}(d^\sharp)).$$

For logical conjunction, the filtered states must satisfy both b_1 and b_2, so conjunction corresponds to $\cap$ and disjunction corresponds to $\cup$. By soundness of the meet $\sqcap^\sharp$ and join $\sqcup^\sharp$ operators of the $\mathsf{Interval}$ domain, the concrete states obtained by concrete meet $\cap$ and

join $\cup$ are contained in the corresponding concretizations. Formally,

$$\llbracket b_1 \rrbracket_{\mathsf{B}}(\gamma(d^\sharp)) \cap \llbracket b_2 \rrbracket_{\mathsf{B}}(\gamma(d^\sharp)) \subseteq \gamma(\llbracket b_1 \rrbracket^\sharp_{\mathsf{B}}(d^\sharp) \sqcap^\sharp \llbracket b_2 \rrbracket^\sharp_{\mathsf{B}}(d^\sharp)),$$
$$\llbracket b_1 \rrbracket_{\mathsf{B}}(\gamma(d^\sharp)) \cup \llbracket b_2 \rrbracket_{\mathsf{B}}(\gamma(d^\sharp)) \subseteq \gamma(\llbracket b_1 \rrbracket^\sharp_{\mathsf{B}}(d^\sharp) \sqcup^\sharp \llbracket b_2 \rrbracket^\sharp_{\mathsf{B}}(d^\sharp)).$$

Thus the inclusion holds.

Therefore, the theorem holds for all boolean expressions. □

For each statement, the abstract transformer $\llbracket \cdot \rrbracket^\sharp : \mathsf{S} \times D^\sharp \to D^\sharp$ is defined as follows:

$$\begin{aligned}
\llbracket i = a \rrbracket^\sharp(d^\sharp) &\stackrel{\text{def}}{=} \begin{cases} d^\sharp(i) = \llbracket a \rrbracket^\sharp_{\mathsf{A}}(d^\sharp) & \text{if } \llbracket a \rrbracket^\sharp_{\mathsf{A}}(d^\sharp) \neq \bot_{\mathcal{I}} \\ \bot^\sharp & \text{otherwise} \end{cases} \\
\llbracket s_1;\ s_2 \rrbracket^\sharp(d^\sharp) &\stackrel{\text{def}}{=} \llbracket s_2 \rrbracket^\sharp(\llbracket s_1 \rrbracket^\sharp(d^\sharp)) \\
\llbracket \mathsf{if}(b)\mathsf{then}\{s_t\}\mathsf{else}\{s_f\} \rrbracket^\sharp(d^\sharp) &\stackrel{\text{def}}{=} \llbracket s_t \rrbracket^\sharp(\llbracket b \rrbracket^\sharp{}_{\mathsf{B}}(d^\sharp)) \sqcup^\sharp \llbracket s_f \rrbracket^\sharp(\llbracket \neg b \rrbracket^\sharp{}_{\mathsf{B}}(d^\sharp)) \\
\llbracket \mathsf{while}(b)\{s\} \rrbracket^\sharp(d^\sharp) &\stackrel{\text{def}}{=} \llbracket \neg b \rrbracket^\sharp_{\mathsf{B}}\big(\mathsf{lfp}\ F^\sharp\big)\ , \text{ where } F^\sharp \stackrel{\text{def}}{=} \big(\lambda a^\sharp.\ d^\sharp \sqcup^\sharp \llbracket s \rrbracket^\sharp(\llbracket b \rrbracket^\sharp_{\mathsf{B}}(a^\sharp))\big)
\end{aligned}$$

For integer assignment, we first check whether the abstract value of the right-hand side is $\bot_{\mathcal{I}}$. If so, the post-state is set to $\bot^\sharp$, indicating that the assignment is infeasible. For sequencing, we use the same composition structure as in the concrete semantics. For an if-then-else statement, we constrain the abstract state with the condition in each branch, apply the corresponding statement transformer, and then join the two resulting states. This final join is an over-approximation by design, since the post-state of the conditional must approximate both branches. For a while loop, we similarly define an abstract function $F^\sharp$ to compute loop invariants and take $\mathsf{lfp}\ F^\sharp$ as the abstract state after the loop.

However, computing the least fixed point $\mathsf{lfp}\ F^\sharp$ is generally not feasible directly. By Theorem 2.1.2, the presence of infinite ascending chains in the abstract domain can cause the computation not to terminate. This situation is common in many abstract domains, such as the Interval domain, whose lattice contains infinitely many elements. For example, one can construct infinite ascending chains such as $[0, 0] \sqsubseteq^\sharp [0, 1] \sqsubseteq^\sharp [0, 2] \sqsubseteq^\sharp \ldots$. The following example illustrates this source of non-termination. To force convergence in finite time, an abstract domain provides a *widening* operator $\nabla : D^\sharp \times D^\sharp \to D^\sharp$, defined as follows:

Definition 2.3.1.3 (Widening on a Poset)**.** *Let $\langle D^\sharp, \sqsubseteq^\sharp \rangle$ be a poset. A binary operator $\nabla : D^\sharp \times D^\sharp \to D^\sharp$ is a widening if:*

- *it computes upper bounds: $\forall d_1^\sharp, d_2^\sharp \in D^\sharp.\ d_1^\sharp \sqsubseteq^\sharp (d_1^\sharp \nabla d_2^\sharp)\ \wedge\ d_2^\sharp \sqsubseteq^\sharp (d_1^\sharp \nabla d_2^\sharp)$;*

Example 2.3.1.3 (If-then-else interval analysis).
Based on Fig. 2.3a, consider the state at line 2.3a:4 as $d_0^\sharp = \{x \mapsto [0,0],\ c \mapsto [0,1]\}$. For condition `c == 0`, refinement gives $c \mapsto [0,0]$, and executing lines 5–6 yields

$$d_t^\sharp = \{x \mapsto [2,2],\ c \mapsto [1,1]\}.$$

For the other branch, refinement gives $c \mapsto [1,1]$, and executing lines 8–9 yields

$$d_f^\sharp = \{x \mapsto [4,4],\ c \mapsto [0,0]\}.$$

Joining both branches gives

$$d_t^\sharp \sqcup^\sharp d_f^\sharp = \{x \mapsto [2,4],\ c \mapsto [0,1]\}.$$

This over-approximates the concrete post-states shown in Example 2.2.4.

- *it ensures convergence: for every ascending chain $d_0^\sharp \sqsubseteq^\sharp d_1^\sharp \sqsubseteq^\sharp d_2^\sharp \sqsubseteq^\sharp \cdots$, there is a sequence $w_0 \sqsubseteq^\sharp w_1 \sqsubseteq^\sharp w_2 \sqsubseteq^\sharp \cdots$ defined by $w_0 = d_0^\sharp$ and $w_{i+1} = w_i \triangledown d_{i+1}^\sharp$ is ultimately stationary with in a finite iteration step $\exists n \geq 0.\ w_n = w_{n+1}$.*

Following this definition, we define widening over intervals as follows:

$$\bot_{\mathcal{I}} \triangledown_{\mathcal{I}} I = I \triangledown_{\mathcal{I}} \bot_{\mathcal{I}} \stackrel{\text{def}}{=} I,$$

$$I_1 \triangledown_{\mathcal{I}} I_2 \stackrel{\text{def}}{=} [l,u],\ \text{where } l = \begin{cases} l_1 & \text{if } l_1 \leq l_2 \\ -\infty & \text{otherwise} \end{cases} \text{ and } u = \begin{cases} u_1 & \text{if } u_1 \geq u_2 \\ +\infty & \text{otherwise} \end{cases}$$

The interval widening operator forces convergence by comparing two intervals and accelerating their bounds to the corresponding extremes. Extending this pointwise yields the widening operator for the Interval domain, $\triangledown : D^\sharp \times D^\sharp \to D^\sharp$, defined as follows:

$$\bot^\sharp \triangledown d^\sharp = d^\sharp \triangledown \bot^\sharp \stackrel{\text{def}}{=} d^\sharp,$$
$$d_1^\sharp \triangledown d_2^\sharp \stackrel{\text{def}}{=} \lambda v.\ d_1^\sharp(v) \triangledown_{\mathcal{I}} d_2^\sharp(v)$$

We now show how widening can be used to compute an approximation of a concrete loop invariant through the following fixpoint approximation theorem.

Theorem 2.3.4 (Fixpoint Approximation by Widening)**.** *Let $\langle D, \sqsubseteq \rangle$ be the concrete domain and $\langle D^\sharp, \sqsubseteq^\sharp \rangle$ the abstract domain. Let $F : D \to D$ be a monotone operator, and let*

Example 2.3.1.4 (Abstract loop semantics on an infinite loop).
Consider a loop statement s:

```
1 i = 0;
2 x = nd(0, 1);
3 while (x == 1) { i = i + 1; }
```

Suppose the abstract state before entering the loop is $\{i \mapsto [0,0], x \mapsto [0,1]\}$. Following the definition of $F^\sharp$ and lfp, we start with $\bot^\sharp$ and iteratively apply $F^\sharp$ (omitting x):

$$F^\sharp(\bot^\sharp) = \{i \mapsto [0,0]\} \quad (F^\sharp)^2(\bot^\sharp) = \{i \mapsto [0,1]\} \quad \ldots \quad (F^\sharp)^n(\bot^\sharp) = \{i \mapsto [0,n-1]\}$$

The above computation never reaches a fixpoint in finite time. Even though mathematically $\mathsf{lfp}\ F^\sharp = \{i \mapsto [0,+\infty], x \mapsto [0,1]\}$, the iterative computation itself does not terminate.

$F^\sharp : D^\sharp \to D^\sharp$ be a sound abstract operator for F, as in Definition 2.3.1.2. Consider the sequence of abstract elements $d_0^\sharp, d_1^\sharp, d_2^\sharp, \ldots$ defined by:

$$d_0^\sharp \stackrel{\text{def}}{=} \bot^\sharp$$
$$d_{i+1}^\sharp \stackrel{\text{def}}{=} d_i^\sharp \nabla F^\sharp(d_i^\sharp)$$

Then the following properties hold:

Termination: *The sequence is ultimately stationary, i.e.,* $\exists n \geq 0.\ d_n^\sharp = d_{n+1}^\sharp$.

Soundness: *Its limit $d_\infty^\sharp$ is a sound approximation of* $\mathsf{lfp}\ F$*, that is,* $\mathsf{lfp}\ F \subseteq \gamma(d_\infty^\sharp)$.

Following this theorem, we can provide a definition for the abstract semantics of while loop as follows:

$$[\![\mathsf{while}(b)\{s\}]\!]^\sharp(d^\sharp) \stackrel{\text{def}}{=} [\![\neg b]\!]^\sharp_{\mathsf{B}}\left(\mathsf{lim}\ F^\sharp\right), \text{ where } F^\sharp \stackrel{\text{def}}{=} \left(\lambda a^\sharp.\ a^\sharp \nabla (d^\sharp \sqcup^\sharp [\![s]\!]^\sharp([\![b]\!]^\sharp_{\mathsf{B}}(a^\sharp)))\right)$$

Theorem 2.3.5 (Local Soundness of Statement Abstract Transformer)**.** *For every statement $s \in \mathsf{S}$ and abstract state $d^\sharp \in D^\sharp$,*

$$[\![s]\!](\gamma(d^\sharp)) \subseteq \gamma([\![s]\!]^\sharp(d^\sharp)).$$

Proof. We prove by structural induction on s.

Example 2.3.1.5 (Abstract loop semantics on an infinite loop with widening).
Consider again the example in Example 2.3.1.4. We now apply widening to compute the abstract semantics of the loop. Suppose the abstract state before entering the loop is $\{i \mapsto [0, 0], x \mapsto [0, 1]\}$. Following the definition of $F^\sharp$ and lfp, we start with $\bot^\sharp$ and iteratively apply $F^\sharp$ (omitting x):

$$F^\sharp(\bot^\sharp) = \{i \mapsto [0, 0]\},$$
$$(F^\sharp)^2(\bot^\sharp) = \{i \mapsto [0, 0]\}\nabla\{i \mapsto [0, 1]\} = \{i \mapsto [0, +\infty]\}.$$

The iteration quickly reaches a stable limit, so $\mathsf{lim}\ F^\sharp$ becomes $\{i \mapsto [0, +\infty], x \mapsto [0, 1]\}$. A further iteration still returns the same result. After filtering states to satisfy loop exit condition, we compute the final state as $\{i \mapsto [0, +\infty], x \mapsto [0, 0]\}$.

For assignment $i = a$, the claim follows from Theorem 2.3.2: abstract evaluation of a approximates the concrete evaluation, so assignment of i to the abstract value of a soundly approximates the concrete assignment.

For sequencing $s_1; s_2$, apply the induction hypothesis to s_1 and then to s_2, which is exactly the desired inclusion.

For conditional statement $\mathsf{if}(b)\mathsf{then}\{s_t\}\mathsf{else}\{s_f\}$, by Theorem 2.3.3, filtering with b and $\neg b$ is soundly approximated by $[\![\cdot]\!]^\sharp_{\mathsf{B}}$. Then apply the induction hypothesis to each branch and use soundness of abstract join to conclude soundness of the union of two branches.

For while statement $\mathsf{while}(b)\{s\}$, let F and $F^\sharp$ be the concrete and abstract loop functionals. By Theorem 2.3.4, the abstract iteration with widening terminates and its limit soundly over-approximates $\mathsf{lfp}\ F$. Applying soundness of exit filtering by $\neg b$ (Theorem 2.3.3) gives the desired inclusion.

Therefore, the theorem holds for all statements. □

To conclude, we build the abstract transformer for the complete program $pg := s$. We define an initial abstract state $d^\sharp_{\text{init}}$ such that $X_0 \subseteq \gamma(d^\sharp_{\text{init}})$. Typically, this state assigns the top element ($\top_{\mathcal{I}}$) to all variables, representing a state of no prior knowledge. The value $[\![s]\!]^\sharp(d^\sharp_{\text{init}})$ then computes a final state that over-approximates all concrete states that may be reached at the exit point of the program.

Theorem 2.3.6 (Soundness of the Whole Analysis)**.** *For the input program* $pg := s$*, let* $[\![pg]\!] : \mathcal{P}(\mathsf{State}) \to \mathcal{P}(\mathsf{State})$ *be the concrete transformer and* $[\![pg]\!]^\sharp : D^\sharp \to D^\sharp$ *be the*

Example 2.3.1.6 (Loop interval analysis with widening).
To compare with the concrete loop example in the previous section, we analyze the loop in Fig. 2.3a from

$$d^\sharp_{\text{init}} = \{x \mapsto [0,0],\ c \mapsto [0,1]\}.$$

Using the widening-based loop functional, start from $a_0 = \bot^\sharp$:

$$\begin{aligned}
a_1 &= F^\sharp(a_0) = \{x \mapsto [0,0],\ c \mapsto [0,1]\},\\
a_2 &= F^\sharp(a_1) = a_1 \nabla \big(\{x \mapsto [0,0],\ c \mapsto [0,1]\} \sqcup^\sharp \{x \mapsto [2,4],\ c \mapsto [0,1]\}\big)\\
&= \{x \mapsto [0,+\infty],\ c \mapsto [0,1]\},\\
a_3 &= F^\sharp(a_2) = a_2.
\end{aligned}$$

Thus the iteration stabilizes at

$$\mathsf{lim}\ F^\sharp = \{x \mapsto [0,+\infty],\ c \mapsto [0,1]\}.$$

Applying the loop-exit filter $\neg(x < 7)$ yields $\{x \mapsto [7,+\infty],\ c \mapsto [0,1]\}$, which soundly approximates the concrete final states shown in Example 2.2.6.

abstract transformer. Let $\gamma : D^\sharp \to \mathcal{P}(\mathsf{State})$ be the concretization function. The abstract transformer is sound if, for any abstract state $d^\sharp \in D^\sharp$,

$$[\![pg]\!](\gamma(d^\sharp)) \subseteq \gamma([\![pg]\!]^\sharp(d^\sharp)).$$

Proof. Since the input program is $pg := s$, the soundness of analysis follows Theorem 2.3.5. □

2.3.2 Improving Precision by Combining Abstractions

As Example 2.3.1.6 shows, using the interval domain yields the final state $\{x \mapsto [7,+\infty],\ c \mapsto [0,1]\}$, which is a sound approximation:

$$\{\{x \mapsto 8, c \mapsto 1\}, \{x \mapsto 10, c \mapsto 0\}\} \subseteq \gamma(\{x \mapsto [7,+\infty],\ c \mapsto [0,1]\}).$$

However, this abstraction introduces imprecision, which may lead to false positives during verification. Specifically, the interval $[7,+\infty]$ implies that x could be 7, even though that value is unreachable. To improve precision, one might instead use a Parity domain to

track whether an integer is even or odd. However, parity alone cannot capture numerical properties in the same way as the Interval domain.

Abstract interpretation provides a systematic way to combine multiple domains through a *Cartesian product*, allowing the abstract state to track different properties of each variable simultaneously. Moreover, the subdomains can communicate and refine one another through a process called *reduction*. For example, the combination $x \mapsto [7, +\infty] \land \texttt{even}$ can be reduced to $x \mapsto [8, +\infty]$. Following this idea, we introduce a *reduced product domain* that combines Interval and Parity.

We first introduce the parity value domain and the parity state domain in Definition 2.3.2.4 and Definition 2.3.2.5.

Definition 2.3.2.4 (Parity Value Domain)**.** *The parity value domain is defined as*

$$\mathcal{P} \stackrel{\text{def}}{=} \{\texttt{odd}, \texttt{even}, \top_{\mathcal{P}}, \bot_{\mathcal{P}}\}.$$

Its lattice operations are:

$$\begin{aligned}
P_1 \sqsubseteq_{\mathcal{P}} P_2 &\stackrel{\text{def}}{\iff} P_1 = \bot_{\mathcal{P}} \ \lor\ P_2 = \top_{\mathcal{P}} \ \lor\ P_1 = P_2 \\
P_1 \sqcup_{\mathcal{P}} P_2 &\stackrel{\text{def}}{=} \begin{cases} P_2 & \text{if } P_1 = \bot_{\mathcal{P}} \\ P_1 & \text{if } P_2 = \bot_{\mathcal{P}} \\ P_1 & \text{if } P_1 = P_2 \\ \top_{\mathcal{P}} & \text{otherwise} \end{cases} \\
P_1 \sqcap_{\mathcal{P}} P_2 &\stackrel{\text{def}}{=} \begin{cases} P_2 & \text{if } P_1 = \top_{\mathcal{P}} \\ P_1 & \text{if } P_2 = \top_{\mathcal{P}} \\ P_1 & \text{if } P_1 = P_2 \\ \bot_{\mathcal{P}} & \text{otherwise} \end{cases} \\
P_1 \nabla_{\mathcal{P}} P_2 &\stackrel{\text{def}}{=} P_1 \sqcup_{\mathcal{P}} P_2.
\end{aligned}$$

Since $\mathcal{P}$ is finite, widening can be defined as join.

Its arithmetic operators are:

$$P_1 +_{\mathcal{P}} P_2 \stackrel{\text{def}}{=} \begin{cases} \bot_{\mathcal{P}} & \textit{if } P_1 = \bot_{\mathcal{P}} \vee P_2 = \bot_{\mathcal{P}} \\ \top_{\mathcal{P}} & \textit{if } P_1 = \top_{\mathcal{P}} \vee P_2 = \top_{\mathcal{P}} \\ \texttt{even} & \textit{if } (P_1, P_2) \in \{(\texttt{even}, \texttt{even}), (\texttt{odd}, \texttt{odd})\} \\ \texttt{odd} & \textit{otherwise} \end{cases}$$

$$P_1 -_{\mathcal{P}} P_2 \stackrel{\text{def}}{=} P_1 +_{\mathcal{P}} P_2$$

$$P_1 *_{\mathcal{P}} P_2 \stackrel{\text{def}}{=} \begin{cases} \bot_{\mathcal{P}} & \textit{if } P_1 = \bot_{\mathcal{P}} \vee P_2 = \bot_{\mathcal{P}} \\ \texttt{even} & \textit{if } P_1 = \texttt{even} \vee P_2 = \texttt{even} \\ \texttt{odd} & \textit{if } P_1 = \texttt{odd} \wedge P_2 = \texttt{odd} \\ \top_{\mathcal{P}} & \textit{otherwise} \end{cases}$$

The abstraction function $\alpha_{\mathcal{P}} : \mathcal{P}(\mathbb{N}) \to \mathcal{P}$ *and concretization function* $\gamma_{\mathcal{P}} : \mathcal{P} \to \mathcal{P}(\mathbb{N})$ *are defined as follows:*

$$\alpha_{\mathcal{P}}(\emptyset) \stackrel{\text{def}}{=} \bot_{\mathcal{P}}$$

$$\forall X \in \mathcal{P}(\mathbb{N}) \setminus \{\emptyset\}.\ \alpha_{\mathcal{P}}(X) \stackrel{\text{def}}{=} \begin{cases} \texttt{even} & \textit{if } X \subseteq \{n \in \mathbb{N} \mid n \bmod 2 = 0\} \\ \texttt{odd} & \textit{if } X \subseteq \{n \in \mathbb{N} \mid n \bmod 2 = 1\} \\ \top_{\mathcal{P}} & \textit{otherwise} \end{cases}$$

$$\gamma_{\mathcal{P}}(\bot_{\mathcal{P}}) \stackrel{\text{def}}{=} \emptyset$$

$$\gamma_{\mathcal{P}}(\texttt{even}) \stackrel{\text{def}}{=} \{n \in \mathbb{N} \mid n \bmod 2 = 0\}$$

$$\gamma_{\mathcal{P}}(\texttt{odd}) \stackrel{\text{def}}{=} \{n \in \mathbb{N} \mid n \bmod 2 = 1\}$$

$$\gamma_{\mathcal{P}}(\top_{\mathcal{P}}) \stackrel{\text{def}}{=} \mathbb{N}$$

Definition 2.3.2.5 (Parity State Domain)**.** *The parity state domain maps each variable to a parity abstract value. It is defined as* $D^\sharp = (\mathcal{V} \to \mathcal{P}) \cup \{\bot^\sharp\}$*, with top element* $\top^\sharp \stackrel{\text{def}}{=} \{v \mapsto \top_{\mathcal{P}} \mid v \in \mathcal{V}\}$*. Its operations are:*

$$d_1^\sharp \sqsubseteq^\sharp d_2^\sharp \stackrel{\text{def}}{\iff} \forall v \in \mathcal{V}.\ d_1^\sharp(v) \sqsubseteq_{\mathcal{P}} d_2^\sharp(v)$$

$$d_1^\sharp \sqcup^\sharp d_2^\sharp \stackrel{\text{def}}{=} \lambda v.\ d_1^\sharp(v) \sqcup_{\mathcal{P}} d_2^\sharp(v)$$

$$d_1^\sharp \sqcap^\sharp d_2^\sharp \stackrel{\text{def}}{=} \lambda v.\ d_1^\sharp(v) \sqcap_{\mathcal{P}} d_2^\sharp(v)$$

$$d_1^\sharp \nabla d_2^\sharp \stackrel{\text{def}}{=} \lambda v.\ d_1^\sharp(v) \nabla_{\mathcal{P}} d_2^\sharp(v)$$

For $\bot^\sharp$ *and any* $d^\sharp$*, we define* $\bot^\sharp \sqsubseteq^\sharp d^\sharp$*,* $\bot^\sharp \sqcup^\sharp d^\sharp = d^\sharp$*,* $\bot^\sharp \sqcap^\sharp d^\sharp = \bot^\sharp$*, and* $\bot^\sharp \nabla d^\sharp = d^\sharp$*.*

The concretization function $\gamma : D^\sharp \to \mathcal{P}(\mathsf{State})$ and abstraction function $\alpha : \mathcal{P}(\mathsf{State}) \to D^\sharp$ are defined as:

$$
\begin{aligned}
\alpha(\emptyset) &\stackrel{\text{def}}{=} \bot^\sharp \\
\forall X \in \mathcal{P}(\mathsf{State}) \setminus \{\emptyset\}.\ \alpha(X) &\stackrel{\text{def}}{=} \lambda v.\ \alpha_{\mathcal{P}}(\{\sigma(v) \mid \sigma \in X\}) \\
\gamma(\bot^\sharp) &\stackrel{\text{def}}{=} \emptyset \\
\forall d^\sharp \in D^\sharp \setminus \{\bot^\sharp\}.\ \gamma(d^\sharp) &\stackrel{\text{def}}{=} \{\sigma \in \mathsf{State} \mid \forall v \in \mathcal{V}.\ \sigma(v) \in \gamma_{\mathcal{P}}(d^\sharp(v))\}
\end{aligned}
$$

Next, we define a reduction operator $\rho : (\mathcal{I} \times \mathcal{P}) \to (\mathcal{I} \times \mathcal{P})$ between intervals and parities:

$$
\rho(I, P) \stackrel{\text{def}}{=} \begin{cases} (\bot_{\mathcal{I}}, \bot_{\mathcal{P}}) & \text{if } I = \bot_{\mathcal{I}} \vee P = \bot_{\mathcal{P}} \\ ([l_1, u_1], P_1) & \text{otherwise} \end{cases}
$$

$$
\text{with } l_1 = \begin{cases} l+1 & \text{if } P = \texttt{even} \wedge l \text{ is odd} \\ l+1 & \text{if } P = \texttt{odd} \wedge l \text{ is even} \\ u & \text{otherwise} \end{cases}
$$

$$
u_1 = \begin{cases} u-1 & \text{if } P = \texttt{even} \wedge u \text{ is odd} \\ u-1 & \text{if } P = \texttt{odd} \wedge u \text{ is even} \\ l & \text{otherwise} \end{cases}
$$

$$
\text{and } P_1 = \begin{cases} \texttt{even} & \text{if } l = u \wedge l \text{ is even} \\ \texttt{odd} & \text{if } l = u \wedge l \text{ is odd} \\ P & \text{otherwise} \end{cases}
$$

Definition 2.3.2.6 (Reductivity and Soundness of Reduction)**.** *Let $\langle D, \subseteq \rangle$ and $\langle D^\sharp, \sqsubseteq^\sharp \rangle$ be two posets and concretization function $\gamma : D^\sharp \to D$. A reduction operator $\rho : D^\sharp \to D^\sharp$ is: (1) sound if $\forall d^\sharp \in D^\sharp.\ \gamma(\rho(d^\sharp)) = \gamma(d^\sharp)$, and (2) reductive if $\forall d^\sharp \in D^\sharp.\ \rho(d^\sharp) \sqsubseteq^\sharp d^\sharp$.*

In particular, the product $\langle D_1^\sharp \times D_2^\sharp, \sqsubseteq_{12}^\sharp \rangle$ with γ_{12} is defined following $\langle x, y \rangle \sqsubseteq_{12}^\sharp \langle x', y' \rangle \stackrel{\text{def}}{=} (x \sqsubseteq_1^\sharp x') \wedge (y \sqsubseteq_2^\sharp y')$ and $\gamma_{12}(\langle x, y \rangle) \stackrel{\text{def}}{=} \gamma_1(x) \cap \gamma_2(y)$.

Finally, we update the abstract transformer for this new abstraction. We denote the $\mathsf{Interval}$ domain by $D_1^\sharp$ and the Parity domain by $D_2^\sharp$. The transformer is now $[\![\cdot]\!]^\sharp : (D_1^\sharp \times D_2^\sharp) \to (D_1^\sharp \times D_2^\sharp)$. The main change is the transfer function for arithmetic computation:

$$
[\![a]\!]_{\mathsf{A}}^\sharp((d_1^\sharp, d_2^\sharp)) \stackrel{\text{def}}{=} \rho([\![a]\!]_{\mathsf{A}}^\sharp(d_1^\sharp), [\![a]\!]_{\mathsf{A}}^\sharp(d_2^\sharp))
$$

Domain	Constraint form	Example	Join/Meet/Leq	Memory
Interval [23]	$l_i \leq x_i \leq u_i$	$0 \leq x \leq 10$	$O(n)$	$O(n)$
Zones [77]	$x_i - x_j \leq c$	$x - y \leq 3$	$O(n^3)$	$O(n^2)$
Octagons [78]	$\pm x_i \pm x_j \leq c$	$x + y \leq 7$	$O(n^3)$	$O(n^2)$
Polyhedra [30]	$\sum_k a_k x_k \leq c$	$2x - y \leq 10$	exponential	exponential

Figure 2.5: Common numerical domains: expressiveness and operation costs.

Example 2.3.3.7 (Precision hierarchy of numerical domains).
To illustrate the precision hierarchy, consider the concrete feasible region defined by the constraint set:

$$4x - y \leq -4 \qquad 0 \leq x \qquad y \leq 40$$

The shaded polygon in Fig. 2.6 represents the concrete feasible set. The interval domain approximates it as $\{x \mapsto [0, 9], y \mapsto [4, 40]\}$, and the octagon domain refines this box with the constraint $x - y \leq -4$. In this case, the polyhedra domain represents the original linear system exactly.

2.3.3 Numerical Anlysis: Precision vs. Efficiency Trade-offs

Numerical abstract domains, such as Interval and Parity, differ mainly in how they represent constraints. Non-relational domains are limited to constraints on individual variables, whereas relational domains can express relationships between variables.

Let n be the number of variables. Section 2.3.2 summarizes the numerical domains used in this thesis. The Interval domain typically requires $O(n)$ memory to represent one abstract state, and its join/meet/inclusion operations are computed pointwise in $O(n)$ time. In contrast, Zones and Octagons are relational domains that capture constraints over at most two variables. Both are commonly represented using a difference-bound matrix (DBM), which requires $O(n^2)$ memory, and their core operations are typically dominated by closure steps with $O(n^3)$ worst-case time. In Section 4.2, we present Zones in detail. Polyhedra is the most expressive domain, representing arbitrary linear constraints. However, its operations are computationally expensive, with worst-case exponential time and memory complexity.

Although a more expressive domain usually improves precision, its operations also become more expensive as the underlying representation becomes richer. In practice, one therefore chooses the least expensive domain that can still express the invariants required for the verification goal.

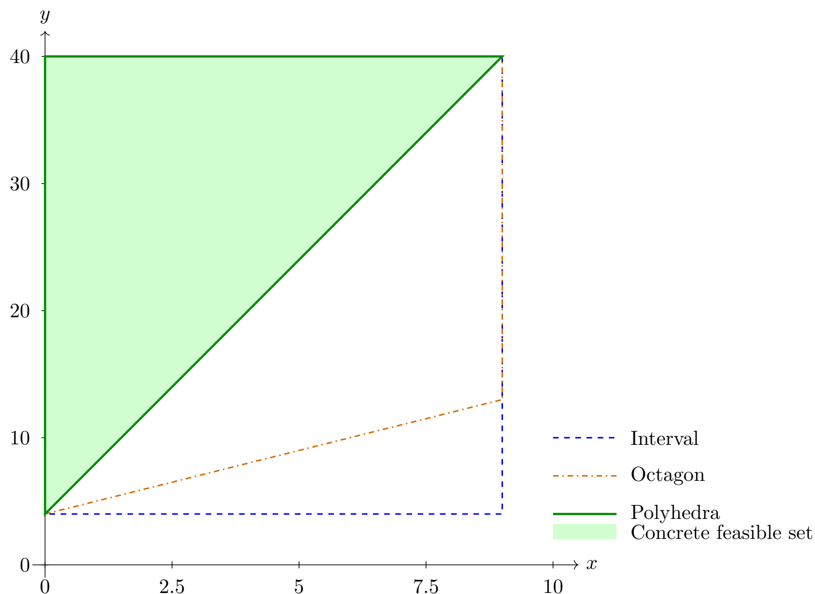


Figure 2.6: Approximating the constraint set with interval, octagon, and polyhedra domains.

There are two common engineering techniques for reducing the cost of domain operations while still preserving much of their expressiveness. First, one can build a reduced product domain, in which each subdomain captures a tighter class of constraints and uses *reduction* to recover precision that no single domain could infer alone. One example is SubPolyhedra [70], which combines LinEq (a linear equality domain with constraints of the form $\sum_k a_k x_k = c$) and Interval. Second, *variable packing* [11] partitions variables into smaller related groups and treats each group independently, reducing both asymptotic and constant factors while preserving important relations within each pack.

2.4 Conclusion

In this chapter, we explored the theory of lattices and Galois connections to provide a formal basis for Abstract Interpretation. These concepts ensure that a program analysis designed following abstract semantics remains a sound approximation of concrete program behavior. We further introduced the concept of abstract domains, whose internal representations and operators form the computational core of the analysis. Through the example of interval domain, we illustrated how numerical invariants are tracked across program points. We

also demonstrated the approximation of loop semantics and the use of widening operator to guarantee termination. Finally, we demonstrated how to combine multiple domains to improve the overall precision of the analysis while preserving the fundamental soundness of the framework.

Abstract Interpretation offers various abstract domains. As demonstrated through the Interval, Octagons, and Polyhedra domains, these numerical abstractions range from computationally efficient but non-relational approximations to highly precise relational domains, albeit at the cost of increased computational overhead. The choice of domain for analysis fundamentally strikes a balance between the precision required to prove properties and eliminate false positives, and the efficiency necessary to remain scalable.

Chapter 3

A New Abstract Domain for Relational Object Invariants

Relational object invariants (or representation invariants) are relational properties held by the fields of a (memory) object throughout its lifetime. For example, the length of a buffer never exceeds its capacity. Automatic inference of these invariants is particularly challenging because they are often broken temporarily during field updates.

In this chapter, we present an Abstract Interpretation-based solution to infer object invariants. Our key insight is a new object abstraction for memory objects, where memory is divided into multiple *memory banks*, each containing several objects. Within each bank, objects are abstracted by separating the *most recently used* (MRU) object, represented precisely with strong updates, while the rest are summarized. For an effective implementation of this approach, we introduce a new composite abstract domain, which forms a reduced product of numerical and equality sub-domains. This design efficiently expresses relationships between a small number of variables (e.g., fields of the same abstract object).

We implement the new domain in the CRAB abstract interpreter and evaluate it on several benchmarks for memory safety. We show that our approach is significantly more scalable for relational properties than the existing implementation of CRAB. To evaluate precision, we have integrated our analysis as a pre-processing step to SEABMC bounded model checker, and show that it is effective at both discharging assertions during pre-processing, and significantly improving the run-time of SEABMC.

3.1 Introduction

Program invariants are crucial to capture properties that persist during runtime. Verifying programs with classes or data structures requires determining *representation invariants* [75] that express *consistency* properties (e.g., the length of a vector never exceeds its capacity) of those data types. For memory objects, representation invariants as *object invariants* describe relational properties among object fields that hold across all program states where these objects are alive. These invariants are essential for proving memory safety and functional correctness of a program. However, the invariants become imprecise when the static analyzer is uncertain about which memory objects are affected by field updates, typically represented as *weak* updates.

Consider a C program in Fig. 3.1 that uses a `byte_buf` to represent a resizable byte buffer with length and capacity. The program keeps an array `ary` of byte buffers. Each initialized element of `ary` satisfies an invariant: `len <= cap`. Discovering this invariant is crucial for establishing memory safety (e.g., proving safe access on line 21), yet, notoriously hard for abstract interpreters. Note that *recency* [5] does not help here because all memory stores after the `for` loop are modeled as weak updates. For instance, Mopsa [82] with recency does not prove the assertion on line 20, since the inferred invariant is $\mathit{len} > 0 \land \mathit{cap} > 1$.

In this chapter, we present a new technique for inferring object invariants. We capture field updates *strongly* in a separate temporary object abstraction and join it with previously established invariants only when necessary. While preserving soundness, our approach produces more precise analysis results by not weakening inferred invariants with intermediate object states between updates.

First, we introduce a new concrete memory model that organizes memory as a collection of *memory banks*, each containing certain memory objects. The partitioning is achieved by a parameterized function that assigns each memory object in the program a corresponding bank. Each bank has two components: *storage*, holding objects, and *cache*, storing the object being read from or written to. For example, all byte buffers in Fig. 3.1 are placed into the storage of the same bank. The field updates on line 12 require loading the byte buffer referred by pointer `p` into the cache before updates. The cache singles out the object being modified. For brevity, we specify this usage pattern with a size of one as *most recently used* (MRU) and denote the object in the cache as the MRU object.

Second, we follow a standard summarization-based abstraction with a single summary object with its invariants representing properties common to all the objects stored in each bank. Similar to the concrete model, all memory updates are handled through the MRU object. This avoids temporarily breaking the invariants of the (abstract) summary object,

```c
#define N 100
struct byte_buf {
    int len;
    int cap;
    char *buf;
};
int main() {
    struct byte_buf *ary[N];
    for (int i = 0; i < N; ++i) {
        struct byte_buf *p =
            malloc(sizeof(struct byte_buf));
        int sz = i + 1;
        p->len = i; p->cap = sz;
        p->buf = malloc(sz);
        ary[i] = p;
    }
    char *new_buf = malloc(20);
    ary[0]->len = 15;
    ary[0]->cap = 20;
    ary[0]->buf = new_buf;
    assert(ary[0]->len <= ary[0]->cap);
    ary[0]->buf[ary[0]->len] = '\0';
}
```

Figure 3.1: A simple C program.

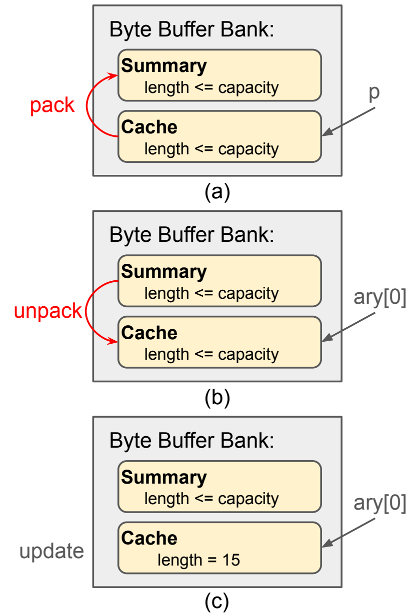


Figure 3.2: Abstract state on line 3.1:17.

as changes to the MRU object do not impact the summarized invariants until it is merged back. Fig. 3.2 presents the changes in the abstract memory state at line 17. The memory bank for byte buffers includes one MRU object and one summary object. Before evaluating line 17, as shown in Fig. 3.2(a), `p` refers to the MRU object, since the last two field updates (line 12) happened on this object. Following the initialization loop, `len <= cap` is kept for both MRU and summary objects.

The cache may *miss* if the cached object is no longer the MRU. For example, the field update, `ary[0]->len = 15`, on line 17 requires access to the byte buffer referenced by `ary[0]`, while the cache still holds the object referred by `p`. In this case, the cached object is *packed* back to the summary (see Fig. 3.2(a)) and the new MRU object is *unpacked* from the summary (Fig. 3.2(b)). We track pointer alias information to decide when to pack and unpack. Before each memory access, if the dereferenced pointer does not alias with the pointer accessed to the MRU object, packing and unpacking occur. In this example, after the loop computation, `p` does not alias `ary[0]`.

After the cache is replaced, the field update, `ary[0]->len = 15`, breaks the invariant `len <= cap`, but our solution (Fig. 3.2(c)) ensures that we update the content of the MRU object properly without affecting the invariants in the summary object. Then, the invariant is restored at line 18, thus proving the assertion on line 20 and memory safety on line 21

through our invariants in the cache.

Third, we introduce a new abstract domain, called *MRUD*, that infers automatically object invariants based on our new memory model. This domain requires combining heap (memory abstraction), must alias (flow-sensitive points-to information) and value (numerical relational invariants) analyses. Using a monolithic numerical domain is highly inefficient because of the large number of dimensions required to model all program variables and their ghost versions that keep track of base addresses, offsets, etc. However, a key insight is that each transfer function typically affects a small subset of variables (e.g., reading a field only updates the corresponding integer/pointer value). Based on this observation, *MRUD* is designed as a composite abstract domain where each memory bank is modeled separately and the propagation of facts between them is carefully limited to a small set of shared variables. This modular design is what makes MRUD both scalable for large code bases and capable of preserving precise object invariants.

We implemented MRUD in the Crab analyzer [55] and evaluated both its scalability and precision. For scalability, we compare it to the summarization-based abstract domain implemented in Crab. Our approach shows improved scalability, with 75X faster performance than the state-of-the-art. For precision, we compare it to the recency domain implemented on Mopsa using a small set of benchmarks. The results show that our approach successfully proves all assertions in the programs and achieves better precision by preserving object invariants. Additionally, we use MRUD in a case study with the bounded model checker SeaBMC, where it effectively proves and discharges memory safety checks to reduce the verification cost of SeaBMC.

In summary, the contributions of this chapter are: (1) We introduce a new memory model designed for object abstraction as an alternative to the C memory model, and describe the concrete semantics of an intermediate representation based on the new model (Section 3.3); (2) We describe the MRUD and corresponding abstract transfer functions, and introduce a domain reduction for invariant refinement (Section 3.4); (3) We detail our implementation (Section 3.5) and evaluate it in the Crab analyzer (Section 3.6).

3.2 Preliminaries

In this section, we first describe an intermediate representation (IR) used for anlysis, `CrabIR` [55]. Next, we introduce a lightweight abstract domain for representing variable equavalences, which is used as a sub-domain in our implementation.

$$
\begin{aligned}
\mathsf{P} &::= \mathsf{F}^{+} \\
\mathsf{F} &::= \mathsf{declare}\ \mathit{fun}(v^{*})\{\ \mathsf{BB}^{+}\ \} \\
\mathsf{BB} &::= l : \mathsf{S}^{*}\ \mathtt{goto}\ l^{+} \mid \\
&\quad\ \ l : \mathsf{S}^{*}\ \mathtt{return}\ v^{*} \\
\mathsf{S} &::= \mathtt{assert}(\mathsf{E}_{\mathsf{cond}}) \mid \mathtt{assume}(\mathsf{E}_{\mathsf{cond}}) \mid \\
&\quad\ \ \mathsf{num} := \mathsf{E}_{\mathsf{int}} \mid \mathsf{S}_{\mathsf{ptr}}
\end{aligned}
\qquad
\begin{aligned}
\mathsf{S}_{\mathsf{ptr}} &::= \mathsf{ptr} := \mathsf{alloc}(\mathsf{fld}, \mathsf{num}) \mid \\
&\quad\ \ \mathsf{ptr2}, \mathsf{fld2} := \mathsf{gep}(\mathsf{ptr1}, \mathsf{fld1}, \mathsf{num}) \mid \\
&\quad\ \ \mathsf{scl} := \mathsf{load}(\mathsf{ptr}, \mathsf{fld}) \mid \mathsf{store}(\mathsf{ptr}, \mathsf{fld}, \mathsf{scl}) \\
\mathsf{E}_{\mathsf{int}} &::= \mathsf{Const} \mid \mathsf{num} \mid \mathsf{E}_{\mathsf{int}}\ \mathsf{op}_{int}\ \mathsf{E}_{\mathsf{int}} \\
\mathsf{E}_{\mathsf{cond}} &::= \mathsf{E}_{\mathsf{int}}\ \mathsf{op}_{cmp}\ \mathsf{E}_{\mathsf{int}}
\end{aligned}
$$

Figure 3.3: The syntax of `CrabIR`.

3.2.1 `CrabIR`: An IR for Inferring Object Invariants

The syntax of `CrabIR` is detailed in Fig. 3.3. For presentation, we assume variables in a program are either integers or pointers. The program operates on memory objects composed of integer and pointer fields. A pointer is a pair of a base address and an offset, where an offset is given by a number num and associated with an optional field name fld. All named fields have fixed offsets. That is, field names are redundant – they are use to simplify the abstraction function in the abstract semantics. In our implementation, the field names are automatically discovered by a whole-program pointer analysis during compilation from the source language to `CrabIR`.

`CrabIR` is a strongly typed IR with explicit type declarations for variables. For brevity, we write $\mathcal{V}$ for the set of all program variables, partitioned into integer variables $\mathcal{V}_{int}$, pointer variables $\mathcal{V}_{ptr}$, and fields $\mathcal{V}_{fld}$. We refer to $\mathcal{V}_{int} \cup \mathcal{V}_{ptr}$ as *scalars*.

As usual, we model a program P as a control-flow graph (CFG) whose basic blocks BB are composed of statements S. Statements in `CrabIR` include gotos, assumptions, assertions, and arithmetic and memory operations. Structured control flow (e.g., `if-then-else` blocks and `for` loops) is lowered to a uniform form using `assume` and `goto` statements. Memory allocation is performed by alloc. Pointer arithmetic is expressed via gep, which computes a target address from a base pointer and an integer offset. Memory reads and writes are performed by load and store, respectively. Note that, `CrabIR` also supports C-like aggregate objects (e.g., arrays) without requiring them to be partitioned into fields. We handle such objects as in prior work [55]. For simplicity, we omit them from the theoretical exposition in this chapter, but our implementation follows [55].

Example 3.2.2.1 (Union-find data structure examples).
Consider two union-find data structures over variable set $\mathcal{V} = \{s, t, x, y, z\}$:

$$\mathbf{m} : \{e_1 : \{s, t\}, e_2 : \{x, y, z\}\} \qquad\qquad \mathbf{n} : \{e_1 : \{x, y\}\}$$

$\mathbf{m}$ represents equivalent relations $s \approx t \wedge x \approx y \approx z$. $\mathbf{n}$[a] only expresses $x \approx y$.

[a] The full representation for $\mathbf{n}$ is $\{e_1 : \{x, y\}, e_2 : \{z\}, e_3 : \{s\}, e_4 : \{t\}\}$

3.2.2 Lightweight Equality Abstract Domain

Our analysis requires an equality domain over variable sets $\mathcal{V}$ to express equivalence relations such as $x \approx y$. Although the equality domain can be implemented using weakly relational numerical domains (e.g., [64, 77, 78]), we present a lightweight equality domain that is more efficient for our purposes.

A lightweight equality domain VarEq tracks variable equalities of the form $x \approx y$. We represent these equalities using a union-find (disjoint-set) data structure, where each set corresponds to an equivalence class of variables. Formally, let $\mathcal{V}$ be a finite set of variables. A union-find value $\mathbf{m}, \mathbf{n} \in \prod(\mathcal{V})$[1] induces a partition $\mathcal{V}_1, \ldots, \mathcal{V}_n$ of $\mathcal{V}$. For each variable subset $\mathcal{V}_i$, we designate a canonical *representative* symbol e_i. Two variables $x, y \in \mathcal{V}_i$ satisfy $x \approx y$, and they refer to the same representative e_i. Example 3.2.2.1 shows an example.

We define the following basic operations for a union-find data structure: (1) $\mathit{make}(v)$, which creates a new equivalence class for a fresh variable v; (2) $e := \mathit{find}(v)$, which returns the equivalence class representative e of variable v; (3) $\mathit{union}(x, y)$, which merges the equivalence classes containing x and y. We additionally provide (4) $s := \mathit{vars}(e)$, which returns the set s of variables in the equivalence class represented by e.

The purpose of VarEq (implemented by union-find) is to approximate *must*-equivalence relations between variables. Formally, we define a set of all equivalence relations over $\mathcal{V}$ as $\mathit{Eq}(\mathcal{V}) = \{R \subseteq \mathcal{V} \times \mathcal{V} \mid R \text{ is an equivalence relation}\}$, and each R is reflexive, symmetric, and transitive. We denote $x \sim_R y$ or an ordered pair $(x, y) \in R$ for two variables x and y follows the equivalence relation R. Example 3.2.2.2 shows a value equivalence relation between variables. The concrete domain is the powerset lattice $\mathcal{R} = (\mathcal{P}(\mathit{Eq}(\mathcal{V})), \subseteq, \cup, \cap, \emptyset, \mathit{Eq}(\mathcal{V}))$, where a concrete element $X \in \mathcal{P}(\mathit{Eq}(\mathcal{V}))$ is a set of (possible) equivalence relations. The infimum (bottom element) is $\emptyset$, denoting that no equivalence relation is possible, and the

[1] $\prod(\mathcal{V})$ is the set of all partitions of $\mathcal{V}$.

Example 3.2.2.2 (Instantiating $Eq(\mathcal{V})$ with value equivalence).
Let $\mathcal{V} = \{x, y, z\}$ and a concrete program state be a valuation map $\sigma : \mathcal{V} \to \mathbb{Z}$. A straightforward equivalence relation is value equivalence between variables. That is, for each state σ, an equivalence relation $R_\sigma \in Eq(\mathcal{V})$ is defined as:

$$R_\sigma = \{(u, v) \in \mathcal{V} \times \mathcal{V} \mid \sigma(u) = \sigma(v)\}.$$

For example, let

$$\sigma(x) = 3, \quad \sigma(y) = 3, \quad \sigma(z) = 5.$$

Then the induced equivalence relation is

$$R_\sigma \;=\; \{(x, x), (y, y), (z, z), (x, y), (y, x)\}.$$

supremum (top element) is $Eq(\mathcal{V})$, denoting that any equivalence relation being possible. All lattice operations follow set operations.

VarEq is a complete lattice $\mathcal{R}^\sharp = (\prod(\mathcal{V}), \sqsubseteq^{Eq}, \sqcup^{Eq}, \sqcap^{Eq}, \bot^{Eq}, \top^{Eq})$. The top element is $\top^{Eq} = \{e_1 : \{v_1\}, e_2 : \{v_2\}, \ldots\}$, representing no equalities are known. We explicitly add a distinguished bottom element $\bot^{Eq}$ to represent an unreachable (inconsistent) abstract state. A Hasse diagram example for $\mathcal{R}^\sharp$ is presented in Fig. 3.4. The domain follows concretization function γ^{Eq} defined as follows:

Definition 3.2.2.1 (Concretization)**.** *The concretization function $\gamma^{Eq} : \mathcal{R}^\sharp \to \mathcal{R}$ is defined as:*

$$\mathbf{m} \in \mathcal{R}^\sharp.\ \gamma^{Eq}(\mathbf{m}) = \begin{cases} \emptyset & \textit{if } \mathbf{m} = \bot^{Eq} \\ \{R \in Eq(\mathcal{V}) \mid \forall x, y \in \mathcal{V}.\ x \approx y \in \mathbf{m} \implies x \sim_R y\} & \textit{otherwise} \end{cases}$$

Our abstract domain lattice operations are defined based on operations of union-find data structure. The following paragraphs describe these.

Partial Order $\sqsubseteq^{Eq}$. Inclusion follows the property: a union-find object $\mathbf{m}$ is a *refined* partition of $\mathbf{n}$ if and only if $\forall x, y \in \mathcal{V}.\ x \approx y$ entailed by $\mathbf{n}$ is also satisfied in $\mathbf{m}$. Thus, $\mathbf{m} \sqsubseteq^{Eq} \mathbf{n} \iff (\forall x, y \in \mathcal{V} \cdot\ \mathbf{n}.\mathit{find}(x) = \mathbf{n}.\mathit{find}(y) \implies \mathbf{m}.\mathit{find}(x) = \mathbf{m}.\mathit{find}(y))$. As seen in Example 3.2.2.1, $\mathbf{m} \sqsubseteq^{Eq} \mathbf{n}$.

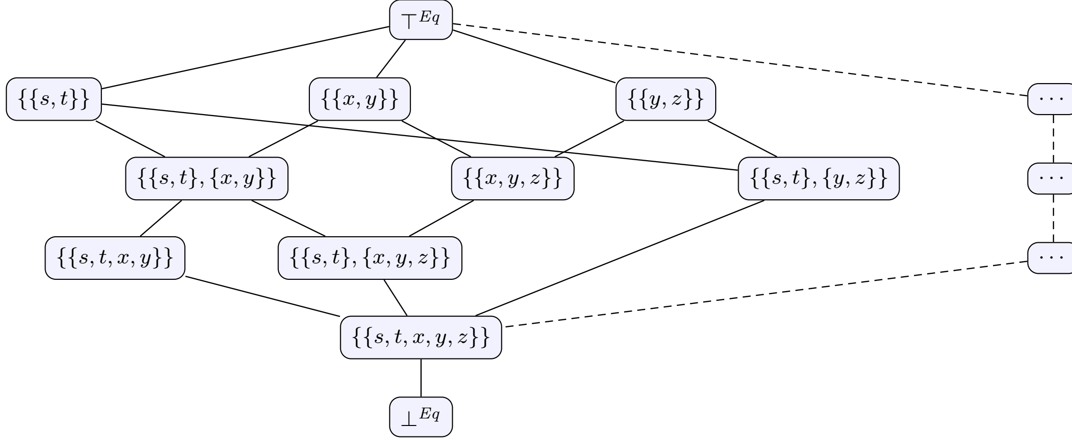


Figure 3.4: Hasse diagram for the VarEq over variable set $\mathcal{V} = \{s, t, x, y, z\}$.

Example 3.2.2.3 (Concretization).
Following Example 3.2.2.1, by definition of γ^{Eq},

$$\gamma^{Eq}(\mathbf{m}) = \{\, R, Q \,\},$$

where $R = \{s \sim_R t, x \sim_R y, y \sim_R z\}$ and $Q = \{s \sim_Q t, t \sim_Q x, x \sim_Q y, y \sim_Q z\}$.
For $\mathbf{n}$, since it infers only $x \approx y$, $\gamma^{Eq}(\mathbf{n})$ contains every equivalence relation R such that $x \sim_R y$ holds.

Join $\sqcup^{Eq}$. We define the join following the join of congruence closures algorithm described in [52]. The result of join regarding union-find is the *least* common partition of both $\mathbf{m}, \mathbf{n}$. That is, $\forall x, y \in \mathcal{V}.\ x \approx y$ from the result union-find is satisfied in both. The join algorithm, as presented in Fig. 3.5, computes the intersection of two variable sets corresponding to equivalence classes in $\mathbf{m}, \mathbf{n}$ if these two classes have common variables. The result will be further added into $\mathbf{r}$ as a new equivalence class. Based on Example 3.2.2.1, $\mathbf{m} \sqcup^{Eq} \mathbf{n} = \mathbf{n}$.

Meet $\sqcap^{Eq}$. The meet computes the *greatest* common partition of $\mathbf{m}$ and $\mathbf{n}$, ensuring that $\forall x, y \in \mathcal{V}.\ x \approx y$ persisted in the result must also be valid in either $\mathbf{m}$ or $\mathbf{n}$. The meet algorithm is outlined in Fig. 3.5 and involves: (1) duplicating the union-find object $\mathbf{m}$ to serve as $\mathbf{r}$; (2) iterating over each pair of equivalence relation $x \approx y$ in each equivalence class $e_{\mathbf{n}}$ of $\mathbf{n}$ to further refine the classes in $\mathbf{r}$. Referring back to Example 3.2.2.1, $\mathbf{m} \sqcap^{Eq} \mathbf{n} = \mathbf{m}$.

VarEq has no infinite increasing (decreasing) chains, so the widening (narrowing) op-

```
procedure JOIN(m, n)
    r := {}
    for all x, y ∈ V_m ∪ V_n do
        if m.find(x) = m.find(y) ∧ n.find(x) = n.find(y) then
            e_m := m.find(x); e_n := n.find(x)
            S_m := m.vars(e_m); S_n := n.vars(e_n)
            S_r := S_m ∩ S_n
            if S_r ≠ ∅ then
                pick an element v ∈ S_r
                r.make(v)
                for all v' ∈ S_r ∧ v' ≠ v do
                    r.make(v'); r.union(v, v')
    return r

procedure MEET(m, n)
    r := m                                          ▷ r is copied from m
    for all x, y ∈ V_m ∪ V_n do
        if n.find(x) = n.find(y) then r.union(x, y)
    return r
```

Figure 3.5: The join and meet operations.

eration follows join (meet) strictly. Next, we provide auxiliary operations that are used in Section 3.4.

`addEqual`. To add a new equality $x \approx y$ to the DBM $\mathbf{m}$, merge y into the equivalence class containing x. If x is not already present in $\mathbf{m}$, first initialize a new equivalence class for x. The procedure is shown in Fig. 3.6.

`equals`. To check whether $x \approx y$ or not in value $\mathbf{m}$, we follow the algorithm shown in Fig. 3.6. Regarding union-find data structure, we test equality by checking whether or not two variables are in the same class.

`toCons`. We would like to convert a union-find data structure $\mathbf{m}$ into a conjunction of equalities over variables. The function iterates each equivalence class and pairs each variable with every other variable in the class as one equality (known as a linear constraint).

```
procedure addEqual(m, x, y)
    r := m                                   ▷ r is copied from m
    if x ∉ 𝒱_r then
        r.make(x)
    r.make(y)
    r.union(x, y)
    return r

procedure equals(m, x, y)
    return {x, y} ⊆ 𝒱_m ∧ m.find(x) = m.find(y)
```

Figure 3.6: The `addEqual` and `equals` operations.

As shown in Example 3.2.2.1, `toCons`$(\mathbf{m})$ will compute a system of equality constraints $\{x = y, y = z, x = z, s = t\}$.

In this section, we presented the syntax of `CrabIR` and our lightweight equality domain. The next sections introduce a new memory model use it to define the concrete semantics of `CrabIR`. With this foundation, we then present MRUD for inferring object invariants under the new memory model. Within MRUD, VarEq serves as a sub-domain: it tracks must-alias equivalences among pointer variables and captures value equalities between scalars and object fields.

3.3 Recent-Use Memory Model

A memory model defines how memory is structured and accessed in the operational semantics (i.e., execution) of the program. The standard C memory model (CMM) treats each allocation as a blob of bytes. Specifically, each memory object is a blob of bytes (logically sub-divided into fields). A pointer is a pair (b, o) of an object identifier b (a.k.a., the *base* address) and a numeric offset o within that object. At allocation, an object occupies a blob in memory at an address determined by the memory allocator. Each memory operation is performed through a pointer to access the object's content. In practice, CMM is typically implemented by a flat memory model of the underlying architecture. However, non-flat memory models with multiple address spaces are common, especially in embedded systems [59].

In this chapter, we introduce a new memory model, called *recent-use memory model* (RUMM), that differentiates between the most recently used (MRU) object and other

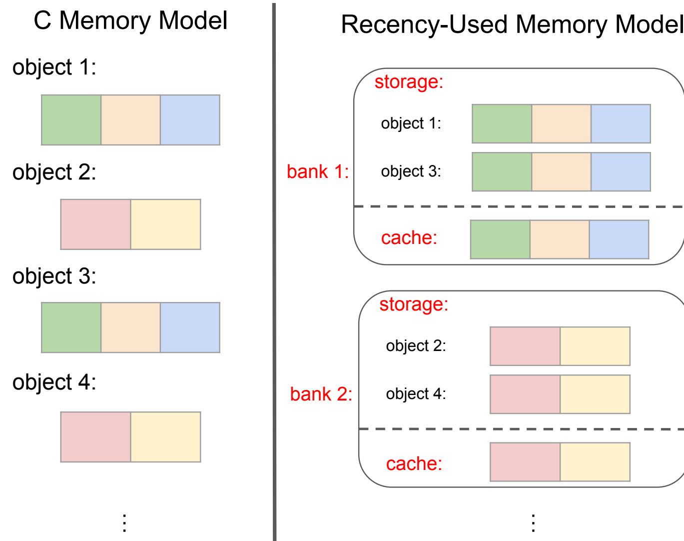


Figure 3.7: C memory model versus recent-use memory model.

memory objects. Fig. 3.7 contrasts CMM and RUMM. RUMM partitions memory into multiple *banks*, each with (a) a *storage* – a blob of bytes that permanently stores memory objects, and (b) a *cache* – a blob of bytes that temporarily holds the MRU object of that bank. The notion of objects and pointers in RUMM is exactly as in CMM. Furthermore, RUMM is parameterized by a function findmb that maps allocation sites to specific memory banks of RUMM. This is similar to a pool allocation, where objects are allocated in different pools [68]. Each object is allocated as a blob in the selected bank's storage, with each bank managing its allocations.

What makes RUMM special is its handling of read and write operations. To access an object x from a given bank, x is first loaded into the cache and then accessed from there. If a different object y currently occupies the cache, y is flushed back to its place in its memory bank before x is loaded. Thus, multiple read and write operations that work on the same object only use the cache, until the cache is flushed when a new object, from the same bank, is accessed.

Fig. 3.8a shows a `CrabIR` for the `for` loop in Fig. 3.1. Variables prefixed with `@` are the fields of `byte_buf`. The loop starts at the *entry block* and checks whether the counter `i` meets the enter/exit condition. In `CrabIR`, `assume` is used to enforce this condition. The loop initializes a memory object, increments the counter, and loops back to the loop entry. Fig. 3.8b illustrates the execution of line 4 during the *second* iteration of the loop. Fig. 3.8b(1) shows the state at line 4, where scalar variables map to their values as *scalar*

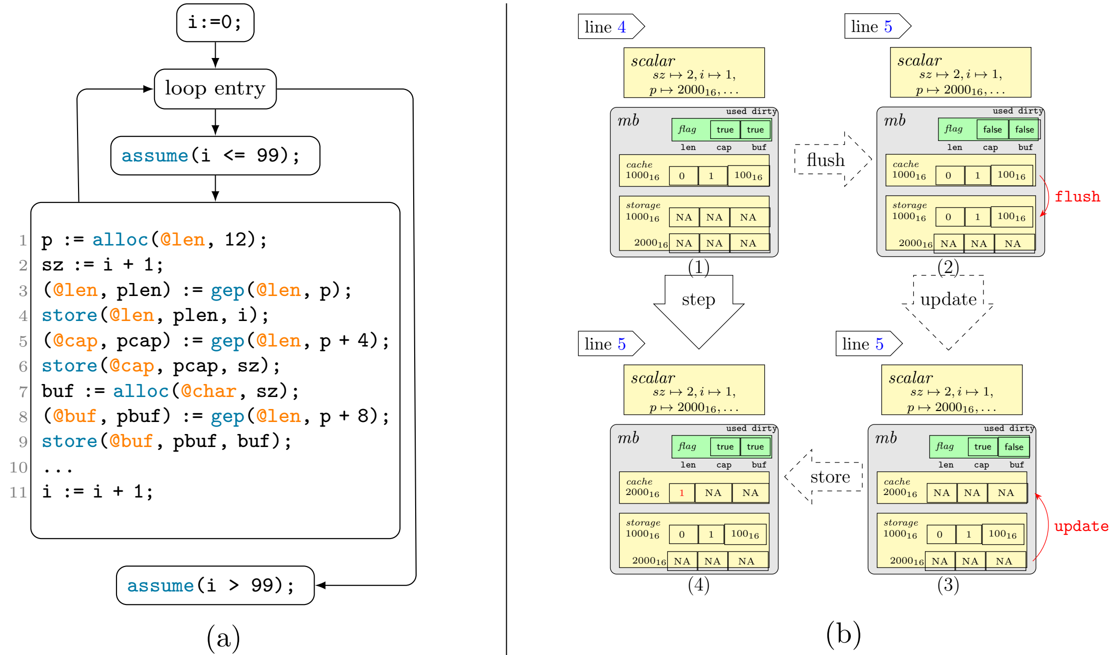


Figure 3.8: (a) A program, and (b) an execution of line 4 under RUMM.

and a memory bank *mb* is provided to store memory objects allocated at line 1. We assume the first two iterations allocate objects at addresses 1000_{16} and 2000_{16}, respectively. The fields of each object are visually represented as slots, with either concrete values or marked as not available (NA) if uninitialized. The storage keeps two uninitialized objects, while the cache holds the MRU object. The object at address 1000_{16} is the MRU since its last access is at line 9 during the first iteration. The cache status is indicated by two flags: *used*, indicating the cache is active, and *dirty*, meaning the cache value has been updated. When `store` at line 4 accesses the object with address 2000_{16}, the cache flushes the object (1000_{16}) back to the storage (Fig. 3.8b(2)) and updates with the uninitialized object from the storage (Fig. 3.8b(3)). The cache is then ready to write `@len` with a value of 1 (Fig. 3.8b(4)).

We argue that RUMM is compatible with CMM. This follows from: (1) **Spatial disjointness:** RUMM organizes memory objects into separate, non-overlapping memory banks, and (2) **Cache transparency:** the usage of cache is an extra step that does not invalidate the properties of each object. The semantics of `CrabIR` are the same under both memory models. In the following, we formalize the concrete semantics of `CrabIR` under RUMM.

$$
\begin{array}{l}
[\![\mathsf{ptr} := \mathsf{alloc}(\mathsf{fld}, \mathsf{num})]\!]^{\text{RUMM}}(\sigma) \stackrel{\text{def}}{=} \\
\quad \textbf{let } \langle \mathit{scalar}, \mathit{mem} \rangle = \sigma \textbf{ in} \\
\quad \textbf{let } \mathit{mb} = \mathsf{findmb}(\mathsf{fld}, \mathit{mem}) \textbf{ in} \\
\quad \textbf{let } \langle \mathit{cache}, \mathit{storage}, \mathit{flag} \rangle = \mathit{mb} \textbf{ in} \\
\quad \textbf{let } \langle _, \mathit{sz} \rangle = \mathit{scalar}[\mathsf{num}] \textbf{ in} \\
\quad \textbf{let } \langle \mathsf{ptr}^{\mathit{base}}, \mathit{storage}' \rangle = \\
\qquad \mathtt{allocator}_{\mathit{mb}}(\mathit{storage}, \mathit{sz}) \textbf{ in} \\
\quad \textbf{let } \mathit{scalar}' = \\
\qquad \mathit{scalar}[\mathsf{ptr} \mapsto \langle \mathsf{ptr}^{\mathit{base}}, 0 \rangle] \textbf{ in} \\
\quad \textbf{let } \mathit{mb}' = \langle \mathit{cache}, \mathit{storage}', \mathit{flag} \rangle \textbf{ in} \\
\quad \langle \mathit{scalar}', \mathit{mem} \setminus \{\mathit{mb}\} \cup \{\mathit{mb}'\} \rangle
\end{array}
$$

$$
\begin{array}{l}
[\![\mathsf{scl} := \mathsf{load}(\mathsf{ptr}, \mathsf{fld})]\!]^{\text{RUMM}}(\sigma) \stackrel{\text{def}}{=} \\
\quad \textbf{let } \langle \mathit{scalar}, \mathit{mem} \rangle = \sigma \textbf{ in} \\
\quad \textbf{let } \mathit{mb} = \mathsf{findmb}(\mathsf{fld}, \mathit{mem}) \textbf{ in} \\
\quad \textbf{let } \langle \mathsf{ptr}^{\mathit{base}}, _ \rangle = \mathit{scalar}[\mathsf{ptr}] \textbf{ in} \\
\quad \textbf{let } \mathit{mb}' = \mathtt{cacheSync}(\mathit{mb}, \mathsf{ptr}^{\mathit{base}}) \textbf{ in} \\
\quad \textbf{let } \langle \mathit{cache}, _, _ \rangle = \mathit{mb}' \textbf{ in} \\
\quad \textbf{let } \langle _, \mathit{fields} \rangle = \mathit{cache} \textbf{ in} \\
\quad \textbf{let } \mathit{scalar}' = \\
\qquad \mathit{scalar}[\mathsf{scl} \mapsto \mathit{fields}[\mathsf{fld}]] \textbf{ in} \\
\quad \langle \mathit{scalar}', \mathit{mem} \setminus \{\mathit{mb}\} \cup \{\mathit{mb}'\} \rangle
\end{array}
$$

$$
\begin{array}{l}
[\![\mathsf{ptr2}, \mathsf{fld2} := \mathsf{gep}(\mathsf{ptr1}, \mathsf{fld1}, \mathsf{num})]\!]^{\text{RUMM}}(\sigma) \stackrel{\text{def}}{=} \\
\quad \textbf{let } \langle \mathit{scalar}, \mathit{mem} \rangle = \sigma \textbf{ in} \\
\quad \textbf{let } \langle \mathsf{ptr1}^{\mathit{base}}, \mathit{offset} \rangle = \mathit{scalar}[\mathsf{ptr}] \textbf{ in} \\
\quad \textbf{let } \langle _, \mathit{val} \rangle = \mathit{scalar}[\mathsf{num}] \textbf{ in} \\
\quad \textbf{let } \mathit{offset}' = \mathit{offset} + \mathit{val} \textbf{ in} \\
\quad \textbf{let } \mathit{scalar}' = \\
\quad\; \mathit{scalar}[\mathsf{ptr2} \mapsto \langle \mathsf{ptr1}^{\mathit{base}}, \mathit{offset}' \rangle] \textbf{ in} \\
\quad \langle \mathit{scalar}', \mathit{mem} \rangle
\end{array}
$$

$$
\begin{array}{l}
[\![\mathsf{store}(\mathsf{ptr}, \mathsf{fld}, \mathsf{scl})]\!]^{\text{RUMM}}(\sigma) \stackrel{\text{def}}{=} \\
\quad \textbf{let } \langle \mathit{scalar}, \mathit{mem} \rangle = \sigma \textbf{ in} \\
\quad \textbf{let } \mathit{mb} = \mathsf{findmb}(\mathsf{fld}, \mathit{mem}) \textbf{ in} \\
\quad \textbf{let } \langle \mathsf{ptr}^{\mathit{base}}, _ \rangle = \mathit{scalar}[\mathsf{ptr}] \textbf{ in} \\
\quad \textbf{let } \mathit{mb}' = \mathtt{cacheSync}(\mathit{mb}, \mathsf{ptr}^{\mathit{base}}) \textbf{ in} \\
\quad \textbf{let } \langle \mathit{cache}, \mathit{storage}, _ \rangle = \mathit{mb}' \textbf{ in} \\
\quad \textbf{let } \langle \mathit{cache}^{\mathit{base}}, \mathit{fields} \rangle = \mathit{cache} \textbf{ in} \\
\quad \textbf{let } \mathit{cache}' = \langle \mathit{cache}^{\mathit{base}}, \\
\qquad \mathit{fields}[\mathsf{fld} \mapsto \mathit{scalar}[\mathsf{scl}]] \rangle \textbf{ in} \\
\quad \textbf{let } \mathit{mb}'' = \\
\qquad \langle \mathit{cache}', \mathit{storage}, \langle \mathsf{true}, \mathsf{true} \rangle \rangle \textbf{ in} \\
\quad \langle \mathit{scalar}, \mathit{mem} \setminus \{\mathit{mb}\} \cup \{\mathit{mb}''\} \rangle
\end{array}
$$

Figure 3.9: `CrabIR` statements operating under RUMM.

A `CrabIR` program has scalars (i.e., integers $\mathcal{V}_{int}$ and pointers $\mathcal{V}_{ptr}$) whose values are represented as *cells*. A cell, $\mathit{cell} \in \mathsf{Cell} : \mathbb{N} \times \mathbb{Z}$, represents either a pointer's base address and offset, denoted as $\langle \mathit{baddr}, \mathit{offset} \rangle$, or an integer value: $\langle 0, \mathit{val} \rangle$. Formally, a scalar state is $\mathit{scalar} \in \mathsf{Scalar} : \mathcal{V}_{scl} \mapsto \mathsf{Cell}$. To avoid redundancy, we explicitly associate the base address of a ptr with a ghost variable $\mathsf{ptr}^{\mathit{base}} \in \mathcal{V}_{ptr}^{base}$. For example, if a pointer `p` is $\langle 100_{16}, 8 \rangle$, then $\mathtt{p}^{\mathbf{base}}$ is 100_{16}.

The memory is modeled as a set of memory banks, $\mathit{mem} \in \mathsf{Memory} : \{\mathit{mb} \mid \mathit{mb} \in \mathsf{MB}\}$. Each bank, $\mathit{mb} \in \mathsf{MB} : \mathsf{Cache} \times \mathsf{Storage} \times \mathsf{Flag}$, holds memory values for cache, storage, and boolean flags. The cache, $\mathit{cache} \in \mathsf{Cache} : \mathbb{N} \times \mathsf{FldVal}$, includes the cached object's base address (as $\mathit{cache}^{\mathit{base}}$) and field values. The field values (as cells) are kept in an environment $\mathit{fields} \in \mathsf{FldVal} : \mathcal{V}_{\mathit{fld}} \mapsto \mathsf{Cell}$. The storage, $\mathit{storage} \in \mathsf{Storage} : \mathbb{N} \mapsto \mathsf{FldVal}$, maps base addresses of memory objects to the corresponding field environment. The cache boolean flags, *flag*, indicate if it is occupied (*used*) and overwritten (*dirty*). Overall, a

$$
\begin{aligned}
&\texttt{cacheSync}(mb, \mathsf{ptr}^{base}) \stackrel{\text{def}}{=}\\
&\quad \textbf{let } \langle cache, storage, \langle used, dirty\rangle\rangle = mb \textbf{ in}\\
&\quad \textbf{let } \langle cache^{base}, _\rangle = cache \textbf{ in}\\
&\quad \textbf{let } mb' =\\
&\qquad \textbf{if } \neg used \wedge \mathsf{ptr}^{base} \neq cache^{base} \textbf{ then}\\
&\qquad\quad \textbf{let } storage' = \textbf{if } dirty \textbf{ then}\\
&\qquad\quad\ \texttt{flush}(cache, storage) \textbf{ else } storage \textbf{ in}\\
&\qquad\quad \textbf{let } cache' = \texttt{refresh}(storage', \mathsf{ptr}^{base}) \textbf{ in}\\
&\qquad\quad \textbf{let } mb' =\\
&\qquad\quad\ \langle cache', storage', \langle \mathsf{true}, \mathsf{false}\rangle\rangle \textbf{ in}\\
&\qquad\quad mb'\\
&\qquad \textbf{else } mb\\
&\quad \textbf{in } mb'
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{flush}(cache, storage) \stackrel{\text{def}}{=}\\
&\quad \textbf{let } \langle cache^{base}, fields\rangle =\\
&\qquad cache \textbf{ in}\\
&\quad storage[cache^{base} \mapsto fields]
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{refresh}(storage, \mathsf{ptr}^{base}) \stackrel{\text{def}}{=}\\
&\quad \langle \mathsf{ptr}^{base}, storage[\mathsf{ptr}^{base}]\rangle
\end{aligned}
$$

Figure 3.10: Cache operations.

concrete program state $\sigma \in \mathsf{State}$ is a tuple: $\langle scalar, mem\rangle$. We assume findmb maps a field variable and memory state to a memory bank, indicating in which bank the field is stored.

Figs. 3.9 and 3.10 describe the changes to a program state at each memory and pointer arithmetic statements in `CrabIR`. The function $[\![\cdot]\!]^{\text{RUMM}}(\cdot)$ takes a statement and a program state and returns the computed state under RUMM. The initial state's *scalar* is an empty map. Each bank *mb* contains an empty *cache*, an empty map *storage*, and a $\langle \mathsf{false}, \mathsf{false}\rangle$ cache flags.

The alloc statement creates a new memory object of size num, assigns it to a specific bank's storage. The bank is determined by fld through findmb, and its `allocator` constructs the object and returns its base address assigned to ptr.

The gep computes a new pointer value for ptr2 by adding an offset num to the pointer value of ptr1. Earlier, we assume all pointer arithmetic stays inbounds, so the ptr2 and ptr1 have the same base address but (presumably) different offsets.

The load operation accesses the object pointed by ptr from the cache associated with the corresponding memory bank. To ensure the object is cached, we use the `cacheSync` function to check if the cache is missed. If so, we flush the cache back to the storage with `flush` if the cache is modified, and then load the new MRU object by calling `refresh`. The `flush` function moves the currently cached object into *storage*, while `refresh` refreshes the cache with the object pointed by ptr. After that, the object at ptr is in the cache, so the flag *used* is set to `true`. The value of scl in *scalar* gets updated by the cached field fld.

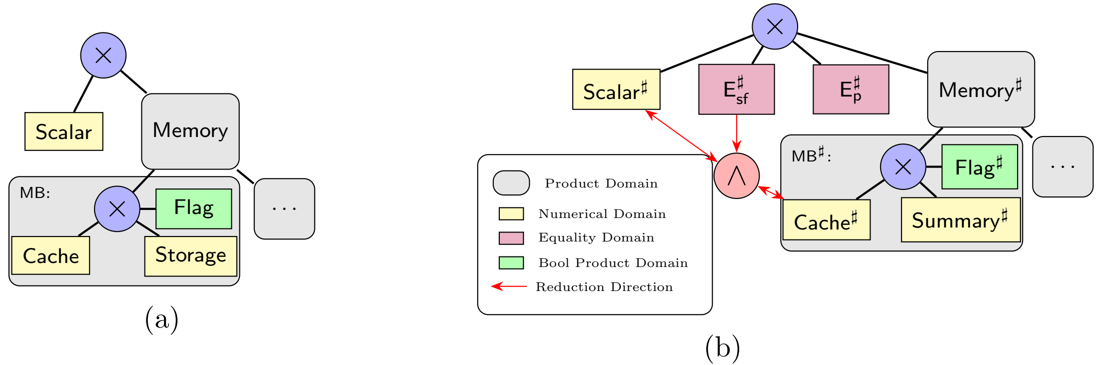


Figure 3.11: (a) Concrete domain and (b) MRUD hierarchy.

Similarly, `store` updates the field for the object, using `cacheSync` to ensure it is in the cache. The flag *dirty* is set to `true`, indicating the object has been modified.

Overall, RUMM offers a different way to organize C memory by partitioning it into multiple banks, with additional space (i.e., the cache) to temporarily hold a memory object for reads and writes. This setup is very convenient for two reasons: first, it allows strong updates on the cache; second, it provides a straightforward memory abstraction by summarizing all objects from the same bank into one and simplifies the design of MRUD, as described in Section 3.4.

3.4 MRUD: Object Invariant Abstract Domain

In this section, we introduce MRUD, a new abstract domain that is a (partially) reduced product of the domains for scalars, pointers, and objects. After setting up the domain, we detail key transfer functions and the reduction procedure.

Similar to the concrete domain in Fig. 3.11a, the MRUD is shown in Fig. 3.11b. It is a reduced product of four domains: (a) a numerical domain $\mathsf{Scalar}^\sharp$, (b) an equality domain $\mathsf{E}^\sharp_{\mathsf{p}}$, (c) an equality domain $\mathsf{E}^\sharp_{\mathsf{sf}}$, and (d) a collection of product domains $\mathsf{Memory}^\sharp : \{\mathsf{MB}^\sharp\}$. $\mathsf{MB}^\sharp$ is a product of two numerical domains, and three Boolean domains: $\mathsf{Cache}^\sharp \times \mathsf{Summary}^\sharp \times \mathsf{Flag}^\sharp$. Fig. 3.12 shows the abstract semantic domains where variables are mapped to unique *dimensions* of each abstract domain. Most domains correspond to those in concrete semantics, except for a few that provide additional information. Specifically, $\mathsf{E}^\sharp_{\mathsf{sf}}$ represents the value equivalence of fields and scalars, which enables information propagation between $\mathsf{Scalar}^\sharp$ and $\mathsf{Cache}^\sharp$ for domain reduction. $\mathsf{E}^\sharp_{\mathsf{p}}$ captures the aliasing properties of pointers, indicating which pointer refers to which object. The added

$$
\begin{aligned}
scalar \in \mathsf{Scalar}^\sharp &\stackrel{\text{def}}{=} \mathsf{Num}(\mathcal{V}_{scl}) \\
e_{sf} \in \mathsf{E}^\sharp_{\mathsf{sf}} &\stackrel{\text{def}}{=} \mathsf{Eq}(\mathcal{V}_{scl} \cup \mathcal{V}_{fld}) \\
e_p \in \mathsf{E}^\sharp_{\mathsf{p}} &\stackrel{\text{def}}{=} \mathsf{Eq}(\mathcal{V}^{base}_{ptr}) \\
\langle used, dirty, ispk \rangle \in \mathsf{Flag}^\sharp &\stackrel{\text{def}}{=} \mathsf{Bool}^\sharp \times \mathsf{Bool}^\sharp \times \mathsf{Bool}^\sharp \\
cache \in \mathsf{Cache}^\sharp &\stackrel{\text{def}}{=} \mathsf{Num}(\mathcal{V}_{fld}) \\
sum \in \mathsf{Summary}^\sharp &\stackrel{\text{def}}{=} \mathsf{Num}(\mathcal{V}_{fld}) \\
mb \in \mathsf{MB}^\sharp &\stackrel{\text{def}}{=} \mathsf{Cache}^\sharp \times \mathsf{Summary}^\sharp \times \mathsf{Flag}^\sharp \\
mem \in \mathsf{Memory}^\sharp &\stackrel{\text{def}}{=} \prod_i \{\mathsf{MB}^\sharp_i\} \\
\sigma^\sharp \in \mathsf{State}^\sharp &\stackrel{\text{def}}{=} \mathsf{Scalar}^\sharp \times \mathsf{MB}^\sharp \times \mathsf{E}^\sharp_{\mathsf{p}} \times \mathsf{E}^\sharp_{\mathsf{sf}}
\end{aligned}
$$

Figure 3.12: Abstract semantic domains.

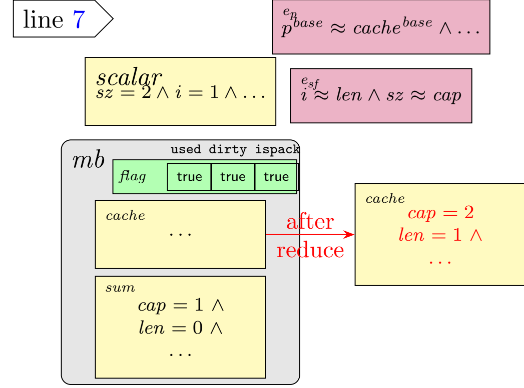


Figure 3.13: State at line 7, 2nd iteration.

Boolean domain in $\mathsf{Flag}^\sharp$ is a flag for later use. All domains are parameterized by relational abstract domains like Zones [77]. An abstract state $\sigma^\sharp$ is represented by lattice elements within the MRUD.

Fig. 3.13 shows the abstract state at line 7 during the second iteration of the `CrabIR` example from Fig. 3.8a. We assume that the Zones domain is used for numerical domains and VarEq is for equality domains. We only show the invariants for scalars `i` and `sz`, and fields `len` and `cap`. *scalar* shows invariants for the scalars `i` and `sz`. The sole memory bank *mb* represents the objects of type `byte_buf`. The *cache* shows the invariants for the MRU `byte_buf` object referenced by pointer `p`. This follows from the equality $p^{base} \approx cache^{base}$ in e_p. The *cache* does not have any explicit invariants for fields. However, the fields invariants are *implicitly* represented through the invariants in *scalar* and the equalities in e_{sf}, $i \approx len$ and $sz \approx cap$, that connect fields and scalars. These equalities are established during field writes. For instance, $i \approx len$ is there because instruction `store(@len, plen, i)` was used to update the field `len` with scalar `i`. Finally, *sum* shows the object invariants for the objects initialized at the first iteration. Specifically, the fields of that object satisfy `len <= cap`.

The most relevant transfer functions for inferring object invariants are shown in Fig. 3.14. For the initial state of analysis, we assign all subdomain elements with $\top$, except for *flag* in each memory bank as $\langle \mathsf{false}, \mathsf{false}, \mathsf{false} \rangle$. The third flag, *ispk*, is false to indicate the *sum* does not represent any concrete objects.

For alloc, the transformer assigns a ptr as not NULL in *scalar* indicating the valid address of the allocated object that ptr refers to. For gep, the transformer computes the address for ptr2 by addition in *scalar* and establishes an equivalence between ptr2 and ptr1 in e_p, denoting that the two pointers refer to the same memory object. For load/store, the transformer requires that the object referred by ptr is in the cache before it is accessed.

$$
\begin{aligned}
&[\![\mathsf{ptr} := \mathsf{alloc}(\mathsf{fld}, \mathsf{num})]\!]^{\text{RUMM}}(\sigma^\sharp) \stackrel{\text{def}}{=} \\
&\quad \textbf{let } \langle \mathit{scalar}, e_{\mathit{sf}}, e_p, \mathit{mem} \rangle = \sigma^\sharp \textbf{ in} \\
&\quad \textbf{let } \mathit{scalar}' = \mathsf{forget}(\mathit{scalar}, \mathsf{ptr}) \textbf{ in} \\
&\quad \textbf{let } \mathit{scalar}'' = \mathsf{addCons}(\\
&\qquad\qquad \mathit{scalar}', \mathsf{ptr} \neq 0) \textbf{ in} \\
&\quad \textbf{let } {e_p}' = \mathsf{forget}(e_p, \mathsf{ptr}^{\mathit{base}}) \textbf{ in} \\
&\quad \langle \mathit{scalar}'', e_{\mathit{sf}}, {e_p}', \mathit{mem} \rangle
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathsf{store}(\mathsf{ptr}, \mathsf{fld}, \mathsf{scl})]\!]^{\text{RUMM}}(\sigma^\sharp) \stackrel{\text{def}}{=} \\
&\quad \textbf{let } \langle \mathit{scalar}, e_{\mathit{sf}}, e_p, \mathit{mem} \rangle = \sigma^\sharp \textbf{ in} \\
&\quad \textbf{let } \mathit{mb} = \mathsf{findmb}^\sharp(\mathsf{fld}, \mathit{mem}) \textbf{ in} \\
&\quad \textbf{let } \langle {e_p}', \mathit{mb}' \rangle = \texttt{cacheSync}^\sharp(\\
&\qquad\qquad \mathit{mb}, e_p, \mathsf{ptr}) \textbf{ in} \\
&\quad \textbf{let } \langle \mathit{cache}, \mathit{sum}, \langle _, _, \mathit{ispk} \rangle \rangle = \mathit{mb}' \textbf{ in} \\
&\quad \textbf{let } \mathit{cache}' = \mathsf{forget}(\mathit{cache}, \mathsf{fld}) \textbf{ in} \\
&\quad \textbf{let } {e_{\mathit{sf}}}' = \mathsf{forget}(e_{\mathit{sf}}, \mathsf{fld}) \textbf{ in} \\
&\quad \textbf{let } {e_{\mathit{sf}}}'' = \texttt{addEqual}({e_{\mathit{sf}}}', \mathsf{scl}, \mathsf{fld}) \textbf{ in} \\
&\quad \textbf{let } \mathit{flag} = \langle \mathsf{true}, \mathsf{true}, \mathit{ispk} \rangle \textbf{ in} \\
&\quad \textbf{let } \mathit{mb}'' = \langle \mathit{cache}', \mathit{sum}, \mathit{flag} \rangle \textbf{ in} \\
&\quad \langle \mathit{scalar}, {e_{\mathit{sf}}}'', {e_p}', \\
&\qquad\qquad \mathit{mem} \setminus \{\mathit{mb}\} \cup \{\mathit{mb}''\} \rangle
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathsf{ptr2}, \mathsf{fld2} := \mathsf{gep}(\mathsf{ptr1}, \mathsf{fld1}, \mathsf{num})]\!]^{\text{RUMM}}(\sigma^\sharp) \stackrel{\text{def}}{=} \\
&\quad \textbf{let } \langle \mathit{scalar}, e_{\mathit{sf}}, e_p, \mathit{mem} \rangle = \sigma^\sharp \textbf{ in} \\
&\quad \textbf{let } \mathit{scalar}' = \mathsf{forget}(\mathit{scalar}, \mathsf{ptr2}) \textbf{ in} \\
&\quad \textbf{let } \mathit{scalar}'' = \mathsf{addCons}(\\
&\qquad \mathit{scalar}, \mathsf{ptr2} = \mathsf{ptr1} + \mathsf{num}) \textbf{ in} \\
&\quad \textbf{let } {e_p}' = \mathsf{forget}(e_p, \mathsf{ptr2}^{\mathit{base}}) \textbf{ in} \\
&\quad \textbf{let } {e_p}'' = \texttt{addEqual}(\\
&\qquad\qquad {e_p}', \mathsf{ptr1}^{\mathit{base}}, \mathsf{ptr2}^{\mathit{base}}) \textbf{ in} \\
&\quad \langle \mathit{scalar}'', e_{\mathit{sf}}, {e_p}'', \mathit{mem} \rangle
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathsf{scl} := \mathsf{load}(\mathsf{ptr}, \mathsf{fld})]\!]^{\text{RUMM}}(\sigma^\sharp) \stackrel{\text{def}}{=} \\
&\quad \textbf{let } \langle \mathit{scalar}, e_{\mathit{sf}}, e_p, \mathit{mem} \rangle = \sigma^\sharp \textbf{ in} \\
&\quad \textbf{let } \mathit{mb} = \mathsf{findmb}^\sharp(\mathsf{fld}, \mathit{mem}) \textbf{ in} \\
&\quad \textbf{let } \langle {e_p}', \mathit{mb}' \rangle = \texttt{cacheSync}^\sharp(\\
&\qquad\qquad \mathit{mb}, e_p, \mathsf{ptr}) \textbf{ in} \\
&\quad \textbf{let } \mathit{scalar}' = \mathsf{forget}(\mathit{scalar}, \mathsf{scl}) \textbf{ in} \\
&\quad \textbf{let } {e_{\mathit{sf}}}' = \mathsf{forget}(e_{\mathit{sf}}, \mathsf{scl}) \textbf{ in} \\
&\quad \textbf{let } {e_{\mathit{sf}}}'' = \texttt{addEqual}({e_{\mathit{sf}}}', \mathsf{fld}, \mathsf{scl}) \textbf{ in} \\
&\quad \langle \mathit{scalar}', {e_{\mathit{sf}}}'', {e_p}', \\
&\qquad\qquad \mathit{mem} \setminus \{\mathit{mb}\} \cup \{\mathit{mb}'\} \rangle
\end{aligned}
$$

Figure 3.14: Abstract transformers for memory operations.

The function $\texttt{cacheSync}^\sharp$ in Fig. 3.15 checks for a cache miss and handles operations when a miss happens. It tests whether ptr refers to the cached object by comparing $\mathsf{ptr}^{\mathit{base}}$ with $\mathit{cache}^{\mathit{base}}$ in e_p. When the cache is missed, the function performs $\texttt{pack}^\sharp$ and $\texttt{unpack}^\sharp$. The $\texttt{pack}^\sharp$ operation merges *cache* into *sum*. The invariants of the first cached object are copied to *sum* because, initially, *sum* does not represent any concrete objects. We change the flag *ispk* to true since the *sum* now holds the invariants for that object. Any subsequent packs use the join operation. The $\texttt{unpack}^\sharp$ is achieved by copying the *sum* as the new *cache*. The $\texttt{pack}^\sharp$ and $\texttt{unpack}^\sharp$ operations are similar to the *fold* and *expand* in [49] but simpler because *cache* and *sum* are two domain values underlying the same field dimensions. After unpacking, $\mathit{cache}^{\mathit{base}}$ equals $\mathsf{ptr}^{\mathit{base}}$, signifying the *cache* is for the new MRU object. The transformer then performs a strong read/update in *cache* without changing any invariant stored in *sum*. The read/update creates an equivalence relation between fld and scl in e_{sf} through $\texttt{addEqual}$. For field read, the transformer discards the information in scl before

$$
\begin{array}{l}
\texttt{cacheSync}^\sharp(mb, e_p, \mathsf{ptr}) \stackrel{\text{def}}{=} \\
\quad \textbf{let } \langle cache, sum, \langle used, dirty, ispk\rangle\rangle = mb \textbf{ in} \\
\quad \textbf{let } \langle e_p', mb'\rangle = \\
\quad\quad \textbf{if } \neg used \wedge \neg\texttt{equals}(e_p, \mathsf{ptr}^{base}, cache^{base}) \textbf{ then} \\
\quad\quad\quad \textbf{let } sum', ispk' = \textbf{if } dirty \textbf{ then } \texttt{pack}^\sharp(cache, sum, ispk) \textbf{ else } sum, ispk \textbf{ in} \\
\quad\quad\quad \textbf{let } cache' = \texttt{unpack}^\sharp(sum') \textbf{ in} \\
\quad\quad\quad \textbf{let } e_p' = \mathsf{forget}(e_p, cache^{base}) \textbf{ in} \\
\quad\quad\quad \textbf{let } e_p'' = \texttt{addEqual}(e_p', \mathsf{ptr}^{base}, cache^{base}) \textbf{ in} \\
\quad\quad\quad \langle e_p'', \langle cache', sum', \langle \mathsf{true}, \mathsf{false}, ispk'\rangle\rangle\rangle \\
\quad\quad \textbf{else } \langle e_p, mb\rangle \\
\quad \textbf{in } \langle e_p', mb'\rangle
\end{array}
$$

$$
\texttt{pack}^\sharp(cache, sum, ispk) \stackrel{\text{def}}{=} \textbf{if } \neg ispk \textbf{ then } \langle \texttt{copy}(cache), \mathsf{true}\rangle \textbf{ else } \langle sum \sqcup cache, ispk\rangle
$$

$$
\texttt{unpack}^\sharp(sum) \stackrel{\text{def}}{=} \texttt{copy}(sum)
$$

Figure 3.15: Abstract cache operations.

adding the equality. For field update, the transformer forgets information about fld ahead of setting the equality and sets *dirty* to true afterward.

Other abstract operators, including join $\sqcup^{MRU}$, meet $\sqcap^{MRU}$, widening $\triangledown^{MRU}$, and narrowing $\sqsubseteq^{MRU}$, are computed pointwise over respective subdomains. For instance, the $\sqcup^{MRU}$ is delegated to the specific join operator of each subdomain, such as $\sqcup^{Num}$ or $\sqcup^{Eq}$. Prior to this pointwise computation, an additional step is required: packing the dirty cache for each memory bank and resetting it as unused. The abstract operators $\star^\sharp \in \{\sqcup, \sqcap, \triangledown, \triangle\}$ is shown in Fig. 3.16a. The algorithm invokes a helper function in Fig. 3.16b to clear the cache in each memory bank. After this, the subdomains are ready for pairwise computations.

We argue that the abstract semantics is sound as it is systematically derived from the concrete semantics. At each program point, the scalar abstraction over-approximates the set of numeric values or addresses of each scalar variable. For memory objects, the abstraction collapses concrete objects in each memory bank into one summary (abstract) object, also as an over-approximation. The soundness argument follows from our design of abstraction and Galois connections. We define the concretization function as follows:

Definition 3.4.1 (MRUD Concretization)**.** *We assume that γ^{Num}, γ^{Eq}, and γ^{Bool} are predefined for the numerical, equality, and Boolean domains, respectively. The γ^{MRU} :*

$\star^\sharp(\sigma_1^\sharp, \sigma_2^\sharp) \stackrel{\text{def}}{=}$
  let $\langle scalar_1, e_{sf\,1}, e_{p\,1}, mem_1\rangle = \sigma_1^\sharp$ **in**
  let $\langle scalar_2, e_{sf\,2}, e_{p\,2}, mem_2\rangle = \sigma_2^\sharp$ **in**
  for all $mb_1 \in mem_1$ **do**
    $mb_1 := \mathsf{flushCache}^\sharp(mb_1)$ ▷ *Update mb_1 directly*
  for all $mb_2 \in mem_2$ **do**
    $mb_2 := \mathsf{flushCache}^\sharp(mb_2)$ ▷ *Update mb_2 directly*
  $\langle scalar_1 \;\star^\sharp\; scalar_2, e_{sf\,1} \;\star^\sharp\; e_{sf\,2}, e_{p\,1} \;\star^\sharp\; e_{p\,2}, mem_1 \;\star^\sharp\; mem_2\rangle$

(a)

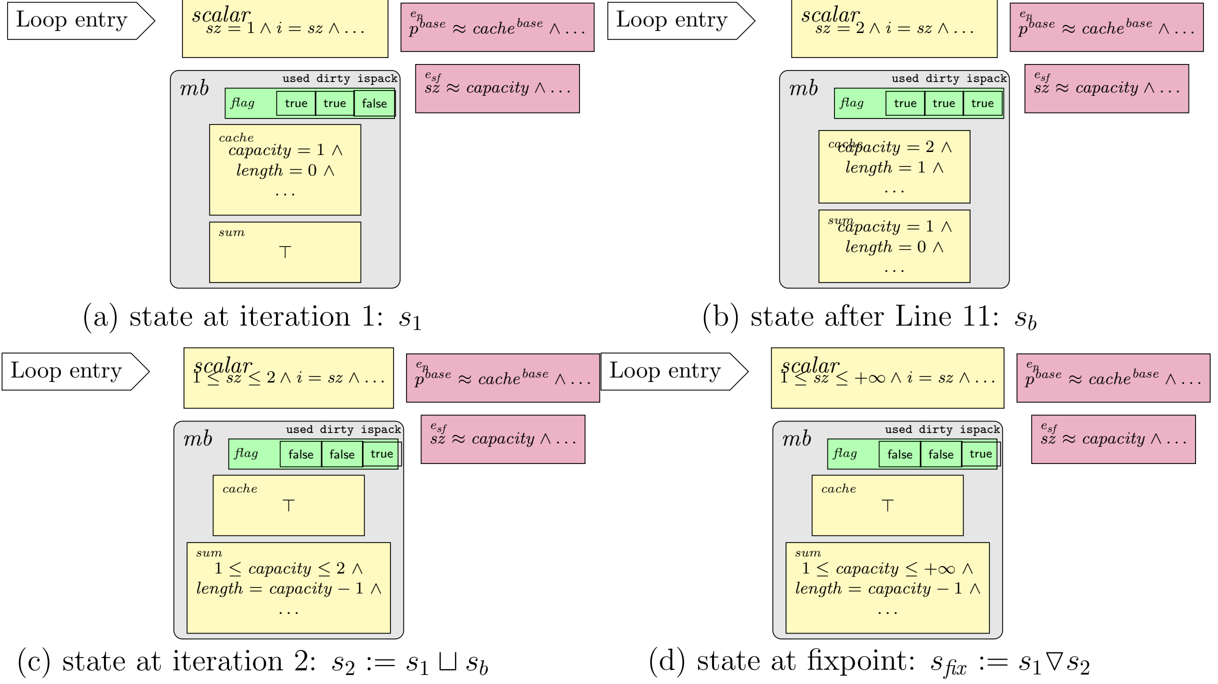
$\mathsf{flushCache}^\sharp(mb) \stackrel{\text{def}}{=}$
  if $\neg used$ **then**
    let $sum', ispk' =$ **if** $dirty$ **then** $\mathtt{pack}^\sharp(cache, sum, ispk)$ **else** $sum, ispk$ **in**
    $\langle cache, sum', \langle \mathsf{false}, \mathsf{false}, ispk'\rangle\rangle$
  else mb

(b)

Figure 3.16: (a) Generic algorithm for Lattice operations; (b) Cache-flush helper function.

$\mathsf{State}^\sharp \to \mathcal{P}(\mathsf{State})$ *is defined as:*

$$
\begin{aligned}
\sigma^\sharp \in \mathsf{State}^\sharp.\quad \gamma^{MRU}(\sigma^\sharp) = \{\sigma \in \mathsf{State} \mid\ & \\
&\sigma.scalar = [\mathsf{num} \mapsto \langle 0, val_1\rangle, \ldots, \\
&\quad \mathsf{ptr}_1 \mapsto \langle baddr_1, \mathit{offset}_1\rangle, \mathsf{ptr}_2 \mapsto \langle baddr_2, \mathit{offset}_2\rangle, \mathsf{ptr}_3 \mapsto \langle baddr_2, \mathit{offset}_3\rangle, \ldots] \wedge \\
&\sigma.mem = \{mb \mid \\
&\quad mb.cache = cache^{base} : [\mathsf{fld}_1 \mapsto \langle 0, val_1\rangle, \ldots], \\
&\quad mb.storage = \{\mathsf{ptr}_2^{base} : [\mathsf{fld}_1 \mapsto \langle 0, val_2\rangle, \ldots], \mathsf{ptr}_4^{base} : \ldots, \ldots\}, \\
&\quad mb.flag = \langle used, dirty\rangle \\
&\} \wedge \\
&[\mathsf{num} \mapsto val_1, \mathsf{ptr}_1^{base} \mapsto baddr_1, \mathsf{ptr}_1^{\mathit{offset}} \mapsto \mathit{offset}_1, \ldots] \in \gamma^{Num}(\sigma^\sharp.scalar) \wedge \\
&\langle used, dirty, _\rangle \in \gamma^{Bool}(\sigma^\sharp.flag) \wedge \\
&[(\mathsf{num}, \mathsf{fld}_1), \ldots] \in \gamma^{Eq}(\sigma^\sharp.e_{sf}) \wedge \\
&[(cache^{base}, \mathsf{ptr}_1^{base}), (\mathsf{ptr}_2^{base}, \mathsf{ptr}_3^{base}), \ldots] \in \gamma^{Eq}(\sigma^\sharp.e_p) \wedge \\
&[\mathsf{fld}_1 \mapsto val_1, \ldots] \in \gamma^{Num}(\sigma^\sharp.mb.cache) \wedge \\
&[\mathsf{fld}_1 \mapsto val_2, \ldots] \in \gamma^{Num}(\sigma^\sharp.mb.sum)\}
\end{aligned}
$$



(a) state at iteration 1: s_1

(b) state after Line 11: s_b

(c) state at iteration 2: $s_2 := s_1 \sqcup s_b$

(d) state at fixpoint: $s_{fix} := s_1 \nabla s_2$

Figure 3.17: Fixpoint computation for the entry state of the loop in Fig. 3.8a.

The function γ^{MRU} maps an abstract state $\sigma^\sharp$ to the set of all concrete states whose variable assignments satisfy the invariants represented by the subdomains of $\sigma^\sharp$. A concrete state σ belongs to $\gamma^{MRU}(\sigma^\sharp)$ if the values in $\sigma.scalar$ are admitted by $\gamma^{Num}(\sigma^\sharp.scalar)$, and similarly for each memory bank $mb \in \sigma.mem$. The equality components connect these concrete values: for example, $(\mathsf{num}, \mathsf{fld}_1)$ is justified by $\gamma^{Eq}(\sigma^\sharp.e_{sf})$ when num and fld_1 have the same value val_1, and $(\mathsf{ptr}_2^{base}, \mathsf{ptr}_3^{base})$ is established by $\gamma^{Eq}(\sigma^\sharp.e_p)$ when ptr_2^{base} and ptr_3^{base} denote the same base address $baddr_2$. Thus, $\gamma^{MRU}(\sigma^\sharp)$ collects the concrete states whose scalar values, memory contents, and equality relations are all consistent with $\sigma^\sharp$.

Fig. 3.17 illustrates the computation of abstract states at the loop entry of Fig. 3.8a. In Fig. 3.17a, state s_1 represents the an abstract state at the loop entry after the first iteration opf the loop. Since during the first iteration only one `byte_buf` object is initialized, the cache in s_1 has the invariants only of that object: $len = 0$ and $cap = 1$, while the summary has no objects (i.e., *ispk* flag is unset). The next abstract state is s_b (Fig. 3.17b) after line 11. During the second iteration, the cache is flushed for the new `byte_buf` object and the summary only maintains the invariants for the flushed object. Then, s_1 and s_b are joined at the loop entry, resulting in s_2 (Fig. 3.17c). The join is pairwise across subdomains

$\mathsf{reduce}(base_{src}, base_{dst}, e) \stackrel{\text{def}}{=}$
let $e_1 = \mathsf{project}(e, \mathcal{V}_{src} \cup \mathcal{V}_{dst})$ **and** $cons = \mathtt{toCons}(e_1)$ **in**
let $base_{dst2} = \mathsf{project}((base_{dst} \sqcap \mathsf{addCons}(base_{src}, cons)), \mathcal{V}_{dst})$ **in**
return $base_{dst2}$

$\mathsf{reduction}(\sigma^\sharp) \stackrel{\text{def}}{=}$
let $\langle scalar, e_{sf}, e_p, mem \rangle = \sigma^\sharp$ **in**
for all $mb \in mem$ **do** ▷ *Step 1: reduce from caches to base*
let $\langle cache, _, _ \rangle = mb$ **in**
$scalar' := \mathsf{reduce}(cache, scalar, e_{sf})$
for all $mb \in mem$ **do** ▷ *Step 2: reduce from base to caches*
let $\langle cache, sum, flag \rangle = mb$ **in**
let $cache' = \mathsf{reduce}(scalar', cache, e_{sf})$ **in**
$mb := \langle cache', sum, flag \rangle$ ▷ *Update mb directly*
$\langle scalar', e_{sf}, e_p, mem \rangle$

Figure 3.18: Domain reduction.

after the caches of both states are flushed. Finally, the widening operator is applied to reach a fixpoint, as shown in Fig. 3.17d.

As the memory and scalar properties are kept separately, we configure a domain reduction step to exchange information between each bank's *cache* and *scalar* through the equalities that are introduced during load and store. We use a bidirectional reduction (see red arrows on the right of Fig. 3.11): one direction flows from the $\mathsf{Cache}^\sharp$ of each memory bank to $\mathsf{Scalar}^\sharp$; the other is in the opposite. The domain reduction follows Fig. 3.18 which reduces an abstract state as $\sigma^\sharp$ in two steps by propagates numerical properties (1) from each *cache* into *scalar*, and (2) from the *scalar* back to each *cache*. The algorithm computes the iterated pairwise reduction through reduce which operates on each bank's *cache* and *scalar*. For example, Fig. 3.13 shows the *cache* after applying the reduction whose values are refined for `cap` and `len` based on equalities generated for field updates through scalars `sz` and `i` in *scalar*. The *cache* is reduced through the step (2) which involves reduce converting equalities ($len \approx i$ and $cap \approx sz$) into linear constraints and adding them to *scalar*. Then, it performs a meet with *cache* to propagate numerical information from *scalar*. Finally, it projects the result of the meet to the field variables, and obtains the new *cache*.

A single execution of reduction refines the abstract values within *cache* of each bank

and *scalar*. Because it propagates only numerical properties already implied by existing and unmodifed equality relations, the operation is guaranteed to be both reductive and sound, as formalized in Theorem 3.4.1. For efficiency, the reduction is terminated after one iteration for each of the two directions.

Lemma 3.4.1. reduce *computes a new value* $base_{dst2}$ *such that* $base_{dst2} \sqsubseteq^{Num} base_{dst}$.

Proof. Let $base_{src2} = \mathsf{addCons}(base_{src}, cons)$. Since $\mathsf{addCons}$ refines the input value by adding constraints, it follows that $base_{src2} \sqsubseteq^{Num} base_{src}$. Because the meet preserves $base_{src2} \sqcap^{Num} base_{dst} \sqsubseteq^{Num} base_{dst}$ and the projection of this result onto $\mathcal{V}_{dst}$ dervies $base_{dst2}$, we can conclude $base_{dst2} \sqsubseteq^{Num} base_{dst}$. □

Theorem 3.4.1 (Structural properties of π^{MRU})**.** *Let* $\pi^{MRU} : \mathsf{State}^\sharp \to \mathsf{State}^\sharp$ *denote the operator induced by Fig. 3.18. Then:*

1. *(Reductiveness)* $\forall \sigma^\sharp \in \mathsf{State}^\sharp$*,* $\pi^{MRU}(\sigma^\sharp) \sqsubseteq^{MRU} \sigma^\sharp$.
2. *(Soundness)* $\forall \sigma^\sharp \in \mathsf{State}^\sharp$*,* $\gamma^{MRU}(\pi^{MRU}(\sigma^\sharp)) = \gamma^{MRU}(\sigma^\sharp)$.

Proof. We prove each property separately.

Reductiveness. By construction, π^{MRU} only updates *scalar* and each abstract cache in each bank by applying reduce. By Lemma 3.4.1, for every such updated component, the result follows $\sqsubseteq^{Num}$ to the original one. That is, $scalar' \sqsubseteq^{Num} scalar$, and similarly for each updated cache. All other components remain unchanged. Hence, by the pointwise order on $\mathsf{State}^\sharp$, we have $\pi^{MRU}(\sigma^\sharp) \sqsubseteq^{MRU} \sigma^\sharp$.

Soundness. First, we show the forward direction $\gamma^{MRU}(\sigma^\sharp) \subseteq \gamma^{MRU}(\pi^{MRU}(\sigma^\sharp))$. Let $\sigma \in \gamma^{MRU}(\sigma^\sharp)$. By definition of γ^{MRU}, the concrete assignments in $\sigma.scalar$, the concrete contents of each memory bank $mb \in \sigma.mem$, should match equalities represented by $\sigma^\sharp.e_{sf}$ and $\sigma^\sharp.e_p$. The operator π^{MRU} does not modify these equality components; it only propagates numerical invariants already implied by the equality information. Therefore, every numerical constraint added by π^{MRU} is already satisfied by the same concrete state σ. The $\sigma \in \gamma^{MRU}(\pi^{MRU}(\sigma^\sharp))$ holds. Second, by reductiveness we have $\pi^{MRU}(\sigma^\sharp) \sqsubseteq^{MRU} \sigma^\sharp$. By monotonicity of γ^{MRU} with respect to $\sqsubseteq^{MRU}$, it follows that $\gamma^{MRU}(\pi^{MRU}(\sigma^\sharp)) \subseteq \gamma^{MRU}(\sigma^\sharp)$. □

In summary, we introduce MRUD, a composite abstract domain and its corresponding transformer for inferring object invariants. As a reduced product of domains for scalars and objects, MRUD is effective for scalable analysis. The reduction algorithm leverages equalities between variables to avoid precision loss.

3.5 Implementation

We have implemented the MRUD[2] in CRAB [55], a library for building abstract interpretation-based analyses. The $\mathsf{Memory}^\sharp$ is implemented using a Patricia tree [87] for *structural sharing* among multiple abstract elements during analysis. This approach prevents redundant copying of domain values when computing the outputs of domain operators and transfer functions, allowing efficient memory sharing for parts of the abstract state that remain unchanged after an operation. For example, two domain elements of $\mathsf{Memory}^\sharp$ share memory banks if they are unchanged during computation.

We have developed a custom equality domain based on a union-find data structure to represent variable equivalence (e.g., $x \approx y$). The details of this domain are presented in Section 3.2.2. Each equivalence class corresponds to a set of variables (e.g., $\{p^{base}, \mathit{cache}^{base}\}$ as $p^{base} \approx \mathit{cache}^{base}$ in Fig. 3.13). This structure fits the representation of equivalence relations and efficiently supports domain operation. Our implementation also partitions $\mathsf{E}^\sharp_{\mathsf{sf}}$ into reduced product of smaller domains for better alignment with variable packing [11]. Specifically, we use an equality domain $\mathsf{E}^\sharp_{\mathsf{s}}$ for scalars and $\mathsf{E}^\sharp_{\mathsf{f}}$, in each memory bank, for fields. The domain value of $\mathsf{E}^\sharp_{\mathsf{sf}}$ is the union of these smaller domain values. For example, $i \approx \mathit{len} \wedge \mathit{sz} \approx \mathit{cap}$ is maintained as two classes $e_{sf} := \{i, \mathit{len}\}, \{\mathit{sz}, \mathit{cap}\}$ which are equivalent to splitted classes as $e_s := \{i, \tilde{a}\}, \{sz, \tilde{b}\}$ and $e_f := \{\mathit{len}, \tilde{a}\}, \{\mathit{cap}, \tilde{b}\}$ with special representatives $\tilde{a}, \tilde{b}$.

For memory partitioning, we use SEADSA [46] to divide the memory used by the program into memory banks, with each bank containing objects from the same allocation site. As mentioned earlier, in `CrabIR`, a field variable represents an offset to access an object field. The findmb function of RUMM is defined by mapping fields to their corresponding bank. However, in practice, not all field offsets can be determined statically. We over-approximate the values of such field by $\top$. Improving this is left for future work.

For effective and efficient domain reduction, we use heuristics to balance precision and performance. MRUD tracks which direction needs reduction. For example, if equalities between fields and scalars only affect memory reads, there is no need to apply a reduction to refine the corresponding cache. We also allow reduction to be performed on demand. For instance, reduction is applied when an assertion is present in the program.

[2]Publicly available at `https://github.com/LinerSu/crab/tree/VMCAI-2025`.

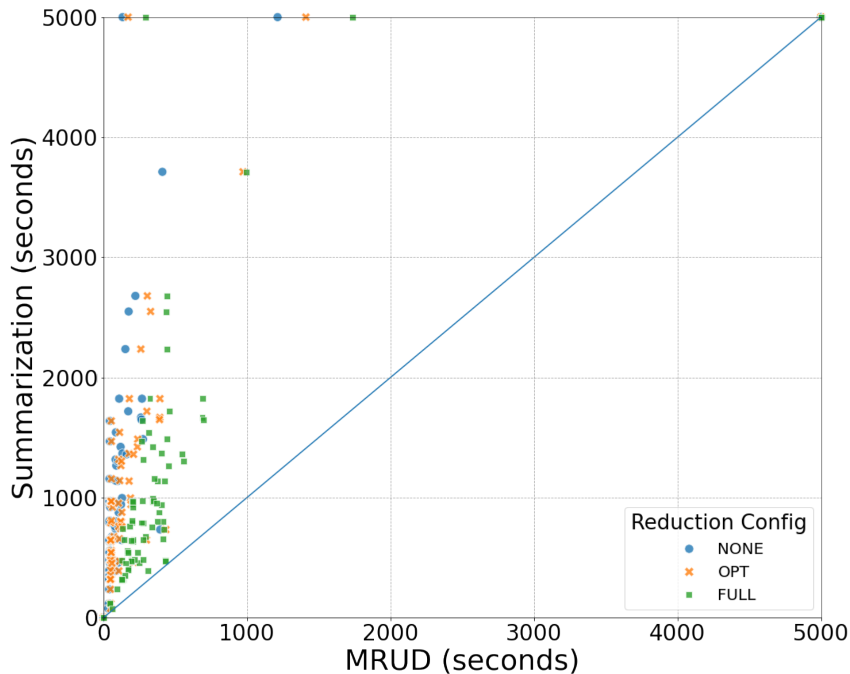


Figure 3.19: Scalability results. Summarization refers to $\mathcal{D}_{\mathcal{S}}$ and MRUD to $\mathcal{D}_{\mathcal{O}}$.

3.6 Evaluation

We performed three kinds of experiments: **scale**, **precision**, and **case study**. All experiments were conducted on a desktop computer with an Intel Xeon E5-2680 @2.50GHz, with 256 GB RAM, and are available at https://doi.org/10.5281/zenodo.13849174.

First, the **scale** experiment compares the performance of MRUD ($\mathcal{D}_{\mathcal{O}}$) with the summarization-based [55] domain ($\mathcal{D}_{\mathcal{S}}$) from CRAB by timing analysis of 114 programs: 5 from [55], and 109 from GNU Coreutils [47]. We used the Zones[3] [46] abstract domain for its simplicity and sufficiency in expressing (relational) memory safety invariants. The primary goal is to show that $\mathcal{D}_{\mathcal{O}}$ scales better than $\mathcal{D}_{\mathcal{S}}$ due to the effect of variable packing [11] in $\mathcal{D}_{\mathcal{O}}$ that follows from representing each partition with a different DBM, while $\mathcal{D}_{\mathcal{S}}$ relies on a single DBM for expressing all scalars (included ghost ones) and summary variables. Another goal is to measure the overhead introduced by domain reduction, which incurs extra costs. To evaluate this, we provide two additional strategies: FULL, which

[3]The Zones domain represents all the binary relationships between two-variable difference (including zero), stored in a Difference-Bound Matrix (DBM).

```
1 void foo(){
2   char ary1[1], ary2[2];
3   struct byte_buf o1 = {.len = 0, .cap
        = 1, .buf=ary1};
4   struct byte_buf o2 = {.len = 1, .cap
        = 2, .buf=ary2};
5   struct byte_buf *p;
6   if (/*some conditions*/) {
7     p = &o1;
8   } else {
9     p = &o2;
10  }
11  p->len = 15; p->cap = 20;
12  ...
13 }
```

Figure 3.20: Another C program.

Program	#A	$\mathcal{D}_\mathcal{O}$	$\mathcal{D}_\mathcal{S}$		$\mathcal{D}_\mathcal{R}$	
		safe	safe	warn	safe	warn
bytebuf	3	3	0	3	0	3
bytebuf_memcpy	3	3	0	3	0	3
bytebuf_path	3	3	1	2	1	2
ipc_handler	3	3	2	1	2	1
mult_bytebuf	3	3	0	3	0	3
object	1	1	0	1	0	1
range	2	2	1	1	0	2

Table 3.1: Precision results.

applies reduction at each transfer function, and NONE, where no reduction is applied, and compare them with the heuristic strategy, OPT. These three strategies highlight the different costs of reduction.

Fig. 3.19 shows the timing results, with a timeout of 5 000 seconds per program. Both domains time out on 6 cases, while $\mathcal{D}_\mathcal{S}$ times out on 2 more cases. Excluding timeout cases, $\mathcal{D}_\mathcal{O}$ outperforms $\mathcal{D}_\mathcal{S}$ on nearly every benchmark. On average, $\mathcal{D}_\mathcal{O}$ with NONE, OPT, and FULL configurations is 81x, 76x, and 57x faster than $\mathcal{D}_\mathcal{S}$, respectively. This demonstrates the advantage of composite abstract domains for inferring object invariants in large and complex programs, regardless of the domain reduction strategy used.

We analyze `ginstall` from GNU Coreutils to understand why $\mathcal{D}_\mathcal{O}$ is faster. The running time for $\mathcal{D}_\mathcal{S}$ is 1 846s, while for $\mathcal{D}_\mathcal{O}$, it takes 273s. Most of the time in both domains is spent on join operations, where $\mathcal{D}_\mathcal{S}$ spends 600s, while $\mathcal{D}_\mathcal{O}$ takes 95s. Joining in $\mathcal{D}_\mathcal{O}$ is also efficient because it allows to share DBMs across memory banks from other states (structural sharing for $\mathsf{Memory}^\sharp$ domain). Another reason is that most DBMs in $\mathcal{D}_\mathcal{O}$ are small, making their joins less costly compared to $\mathcal{D}_\mathcal{S}$, where large DBMs are involved. This efficiency is also reflected in the time to copy DBMs: $\mathcal{D}_\mathcal{S}$ takes 260s, while $\mathcal{D}_\mathcal{O}$ takes 20s.

As for domain reduction, applying it at each transfer function is inefficient, as FULL takes 144 (177) seconds longer than OPT (NONE) on average. The heuristics strategy (OPT) effectively handles complex programs without significant performance loss.

Second, the **precision** experiment compares $\mathcal{D}_\mathcal{O}$ against existing heap abstract domains: $\mathcal{D}_\mathcal{S}$ and Mopsa with recency abstraction ($\mathcal{D}_\mathcal{R}$). Since all three domains follow allocation-site abstraction, which summarizes multiple objects into one and treats them

indistinguishable, it becomes challenging to precisely track field updates on individual concrete objects. Specifically, $\mathcal{D}_{\mathcal{S}}$ cannot overcome this limitation. $\mathcal{D}_{\mathcal{R}}$ improves precision by differentiating the most recently allocated object at the same site. $\mathcal{D}_{\mathcal{O}}$ provides a more general strategy by distinguishing the most recently used object. As a result, $\mathcal{D}_{\mathcal{O}}$ still precisely models field updates after object initialization, such as field updates on lines 17 and 19 in Fig. 3.1, which either $\mathcal{D}_{\mathcal{R}}$ or $\mathcal{D}_{\mathcal{S}}$ cannot handle.

Another challenge is path sensitivity since unclear pointer aliasing leads to imprecise modeling of field updates. For example, in Fig. 3.20, two `byte_buf` objects, `o1` and `o2`, are allocated separately, and a pointer `p` is referred to either `o1` or `o2`. Modeling strong field updates in line 11 requires knowing which object is being updated, but it is unknown which object the pointer `p` refers to. Both $\mathcal{D}_{\mathcal{R}}$ and $\mathcal{D}_{\mathcal{S}}$ can track field updates precisely, but they need more accurate points-to information. $\mathcal{D}_{\mathcal{O}}$, however, allows strong updates by placing `o1` and `o2` in the same memory bank. When updating a field on either object, we load it into the cache and perform strong updates without precise pointer aliasing.

We provide a set of 7 benchmarks[4] with similar code pattern like examples in Figs. 3.1 and 3.20 for evaluation and configure all three domains using the octagon domain. Table 3.1 shows that $\mathcal{D}_{\mathcal{O}}$ successfully proves all assertions, showing the effectiveness of our methodology in providing a more precise memory abstraction. Conversely, $\mathcal{D}_{\mathcal{S}}$ and $\mathcal{D}_{\mathcal{R}}$ largely fail due to weak updates, as discussed above.

Third, we present a **case study** which integrates an Abstract Interpreter (AbsInt) into a Bounded Model Checker (BMC) pipeline for memory safety verification. This new pipeline, AI4BMC, uses AbsInt to verify and remove a number of assertions before passing the problem to the SMT solver.

The AI4BMC pipeline, shown in Fig. 3.21, starts by compiling and instrumenting the input program with buffer overflow checks. Next, AbsInt is applied to remove as many of these checks as possible. Now, the program still keeps the original loops. Then, the loops are unrolled using a user-supplied bound for BMC. Later, we run another AbsInt round to eliminate buffer overflow checks in the simplified program with unrolled loops. Last, we continue with the BMC pipeline, as in SeaBMC [90], that generates a Verification Condition (VC) in SMT-LIB and uses an SMT-solver to check the VC's satisfiability such that the original program is safe if and only if SMT-LIB formula is unsatisfiable.

The motivation for AI4BMC is that many memory safety arguments are simple and are established independently of loop bounds. We expect AbsInt to verify those, leaving less work for BMC. Thus, we consider AI4BMC pipeline successful if (a) AbsInt discharges

[4] Available at: https://github.com/LinerSu/MRU-Domain-Benchmarks.

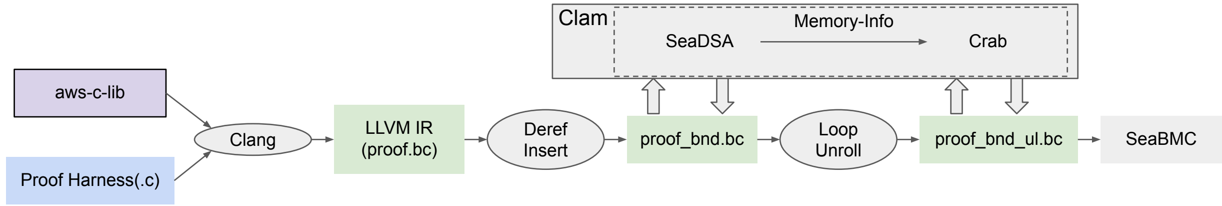


Figure 3.21: The AI4BMC pipeline.

some buffer overflow checks before loop unwinding, and (b) AI4BMC requires less overall runtime than the BMC pipeline.

We developed two benchmark suites adopted from industrial code. The first is based on `aws-c-commons` verification tasks, where we reduce assertions only to memory safety. The second is based on a more complex code from AWS C SDK in C99 implementation. Together, there are 109 verification tasks. The benchmarks[5] have been adapted to simplify control flow since proving all memory safety checks requires path-sensitivity.

We evaluate the effectiveness AI4BMC by comparing it with SeaBMC which was previously compared against other state-of-the-art tools in [90]. Our performance evaluation focuses on these metrics: (1) *Faster* indicates AI4BMC outperforms BMC; (2) *Slower* means AI4BMC is slower than BMC; (3) *AbsInt Time* expresses the run-time of AbsInt in the AI4BMC pipeline. For precision, we provide the *AbsInt Solving Rate*, showing how many checks are solved before or after loop unrolling (LU). We used MRUD for Crab (AbsInt) and chose two SMT-solvers for SeaBMC: Z3[6] [33], and Yices2 [38]. Experiments were conducted under 900 seconds timeout and all results are summarized in Fig. 3.22 and Table 3.2.

First, comparing performance between AI4BMC and BMC. With Z3, AI4BMC timed out in 5 cases, while BMC timed out in 7 cases; AbsInt helped solving 2 more cases. Excluding timeouts, AI4BMC is at least 5s *faster* than BMC in 16 cases. The speed-up comes from AbsInt proving and discharging assertions checks. In 10 of these 16 cases, the speed-up exceeds over 95%, with AbsInt completely solving the checks in 9 cases. The other 6 cases show at least a 20% speed-up. AbsInt takes under one second on average in all 16 cases. There are 4 cases in which AI4BMC is at least 5s *slower* than BMC. In two of these, the slowdowns are due to Z3 taking 6s extra solving time on average, which is not surprising since the SMT performance is not always deterministic. In the other two,

[5] Available at https://github.com/LinerSu/verify-c-common/tree/VMCAI-2025.

[6] We fixed the performance issue on Z3. The one we used is available at: https://github.com/LinerSu/z3/tree/fix-performance.

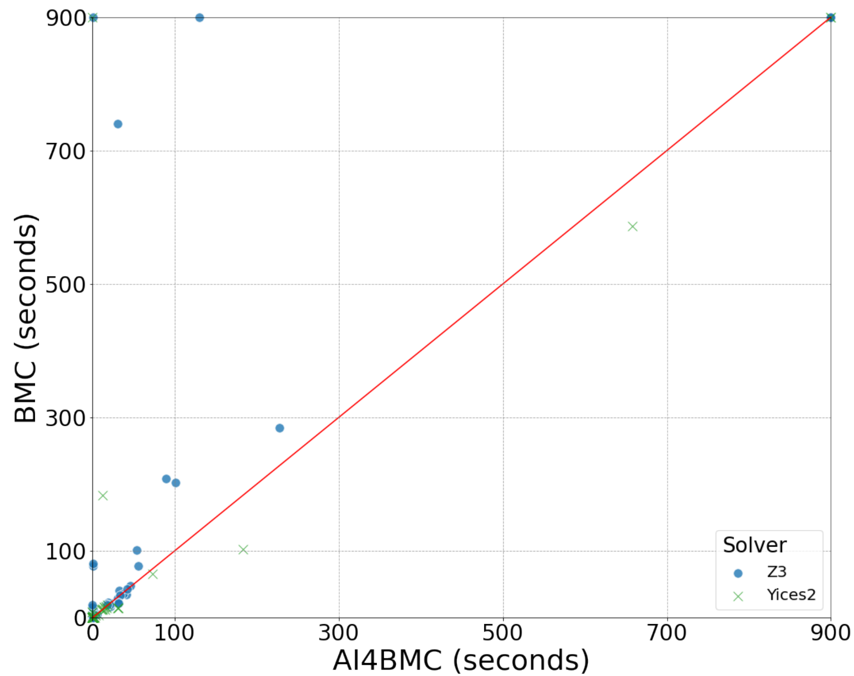


Figure 3.22: AI4BMC vs. BMC.

although Z3 solving time is decreased, AbsInt slows down by taking around 11s, roughly a third of the total run-time.

The results with YICES2 are similar, but YICES2 is faster and exhibits better stability. Both AI4BMC and BMC timed out in 4 cases and 5 cases individually, with 1 case where AbsInt improves performance. AI4BMC outperforms BMC in 3 cases with at least a 93% improvement. However, AI4BMC is slower in 6 cases, 4 of which are affected by the slowdown of AbsInt. The other 2 cases are due to the slows down of SEABMC and YICES2. The SEABMC experiences a slowdown due to lambda-encoding, where the beta-reduction simplification time is not deterministic. While switching to array-encoding shows the effectiveness of AbsInt, this slows overall performance for both AI4BMC and BMC.

Overall, the performance results show that AbsInt improves the overall performance of using BMC regardless of the solver used.

Second, in evaluating the performance of AbsInt, runtime ratios depend on the total running time of AI4BMC and the solver selected. With Z3, AbsInt takes over 40% of the time on 65 cases, but these cases terminate within 50s, with AbsInt averaging only 0.1s and maxing at 1.2s. For the rest of the 39 cases, AbsInt takes 40% or less, with 5 cases

Category	Metric	% Metric	Number of Cases	
			AI4BMC (Z3)	AI4BMC (Y2)
Performance Comparison	*Faster* (Time Difference > 5s)	> 95%	10	2
		others	6	1
	Slower (Time Difference > 5s)	≤ 50%	4	2
		others	0	4
AbsInt Performance	*AbsInt Time* in AI4BMC time	> 40%	65	74
		≤ 40%	39	29
Precision	*AbsInt Solving Rate* before LU	100%	37	37
		> 50%	52	52
	AbsInt Solving Rate after LU	100%	6	6
		> 50%	1	1

Table 3.2: AI4BMC vs. BMC details.

exceeding 50s and 34 cases under 50s. For these 5 longer cases, AbsInt accounts for under 2%, averaging 1s with a maximum of 1.5s. For the 34 shorter cases, AbsInt contribution was below 36%. With Yices2, the runtime percentage of AbsInt increases because Yices2 is efficient, with more cases where AbsInt accounts for a significant portion of the runtime. In summary, using AbsInt has no big cost, compared with the solving time of SMT solver.

Last, for assertion rate, AbsInt solved more than 50% of assertions in 89 cases before LU, completely solving 37 cases, and in 7 cases after LU, fully solving 6 cases. We only have 8 cases where AbsInt solves less than half of the checks. The reasons are: (1) the widening operation produces too imprecise invariants that cannot be recovered by narrowing. AbsInt needs more precise widening techniques to prove more checks; (2) Some memory safety invariants cannot be expressed by Zones or Octagons, and instead require more complex abstract domains such as Polyhedra; (3) Memory safety checks for C string require tracking the length of strings that our implementation does not support. We believe using [62] to determine the null character of each string will improve overall precision.

In this case study, we demonstrate the effectiveness of using AbsInt in the BMC pipeline. By using the Zones, it proves most memory safety checks in this industry project and reduces the number of checks BMC handles. This speeds up both BMC encoding and SMT solver performance.

3.7 Related Works

To deal with a potentially unbounded number of memory objects, most abstract analysis frameworks group memory objects together into *summary objects* (e.g., [49]). A summary

object represents properties that are common to all objects it stands for. The most common summarization is *Allocation Site Abstraction* (ASA) [14] that groups objects by their allocation site. In ASA, all concrete objects allocated at a certain line of a program are represented by one abstract summary object. Since each summary object represents a set of objects, it supports only *weak* updates – an assignment to the field of an object does not override previous value, but rather adds to it, to capture that the field update may modify only one object out of the summary. This significantly degrades analysis precision.

The loss of precision is specifically important during object creation, when an object is first allocated and then initialized field-by-field. In ASA, because of weak updates, this results in all properties of the summary being lost since the newly allocated object has no properties in common with already summarized objects. A common solution, e.g., used by Mopsa, is *recency abstraction* [5] that refines ASA into: (a) the most recently allocated object, and (b) the rest. Since most recent object is a singleton, it can be updated *strongly*, i.e., field updates overwrite previous values. Our approach is a further refinement that separates objects not by recency of *creation*, but by recency of *use*. In principle, other extensions of recency, such as [6] can be combined with our technique for further precision improvement.

The temporary isolation of recenctly-used objects avoids invariant violations in summarized objects during individual field updates. Our pack and unpack methods communicate changes between these two types of objects. This is similar to corresponding methods in [13], where the annotated *pack/unpack* statements manage transitions of mutable objects during class method calls, allowing temporary updates while maintaining class invariants (i.e., invariants for all instances of a given class). Similarly, JayHorn [63] uses *push/pull* statements for encoding each memory access. Each *pull* statement reads fields of an object to make invariants available, while a following *push* statement updates fields to ensure modifications preserve invariants. The concept of *pack/unpack* has been used in refinement types [94], where the inference algorithm obtains predicates with *fold/unfold* operations to prevent temporary invariant violations of objects from the same allocation site. Unlike our work, all prior work uses heuristics to manage placement of *fold/unfold* operations. In contrast, our analysis automatically processes these during analysis.

The domain hierarchy in our MRUD uses two strategies. First, variable packing [11] is used to pack program variables for fields of memory objects in each memory bank. With two numerical domains per pack, our approach allows for the independent updating of invariants for each bank. The packing is rarely used in computing memory properties, but Toubhans et al. [105] introduced a product of memory domains that pack variables used for lists, trees, and other fixed-size structures. Next, domain reduction [26] helps exchange equivalences between scalars and object fields. This is commonly used when abstract domains are

organized modularly. Astrée [28] combines various abstract domains in a sequence, using reduction steps for forward and backward propagation of information between them. [29] interprets the Nelson-Oppen procedure as a domain reduction, propagating (dis)equalities across domains.

3.8 Conclusion

In this work, we present a new methodology for inferring object invariants that avoids temporarily breaking invariants following the concept of caching. Our new abstract domain, parameterized by numerical and equality domains, organizes a structured hierarchy, enabling scalable analysis of complex programs. We design a reduction algorithm following equalities introduced across numerical domains to avoid significant precision loss. Our results demonstrate that MRUD enhances both precision and scalability and can be effectively integrated with other verification techniques for memory safety.

Chapter 4

Template DBM: A New Weakly Relational Domain

A primary goal of static analysis based on abstract interpretation is to infer invariants to verify programs. Memory safety checks (e.g., proving the absence of out-of-bound accesses) require tracking linear relationships between pointer offsets and object sizes, such as `4*idx + 4 <= sz` for accessing memory. Choosing the right abstract domain is crucial, as each domain captures different kinds of properties. For example, Zones and Octagons limit themselves to unit coefficients and cannot express the invariants required for memory safety. On the other hand, the Polyhedra domain can capture any linear relation but does not scale in real applications. In this chapter, as a compromise between expressiveness and efficiency while still covering our target properties, we introduce Template DBM — a new weakly relational numerical domain for expressing Two Variables per Inequality (TVPI) constraints with fixed coefficients. Template DBM supports efficient join, inclusion, and saturation. It strikes the balance of expressiveness and cost between Zones and TVPI. We implemented Template DBM in the Crab library and evaluated it against Zones and Polyhedra domains for memory safety analysis of `aws-c-common` from AWS and `firedancer` from Solana. Our results show that Template DBM maintains its intended level of precision, scales comparably to Zones, and is significantly more efficient than Polyhedra.

4.1 Introduction

Numerical program analysis powers static analyzers for verification and code optimization across real-world software systems. Although using a precise relational abstract domain

```c
struct array_list {
  int size;  // allocated size
  int len;   // used length
  int isz;   // item size
  void *data;
};

int get_at(const struct array_list *l,
      void *val, int idx) {
  if (l->len > idx) {
    int ofs = l->isz * idx;
    void *p = (uint8_t *)l->data + ofs;
    assert(valid_access(p, l->isz));
    assert(valid_access(val, l->isz));
    memcpy(val, p, l->isz);
    return 0;
  }
  return -1;
}
```

```c
void main(int len, int idx) {
  if (0 <= idx && idx < len) {
    struct array_list l;
    l.len = len;
    l.isz = 4; // 4 bytes
    l.size = l.isz * l.len;
    l.data = malloc(l.size);

    // Assume some code initializes l.data.

    void *item = malloc(sizeof(l.isz));
    int ret = get_at(&l, item, idx);
    assert(ret == 0);
  }
}
```

Figure 4.1: An example C program.

ensures accuracy, scalability emerges as a critical challenge when code size grows. For example, Polyhedra [30] domain achieves great precision by encoding arbitrary linear inequalities to capture highly precise program invariants, but its operations incur worst-case time and space complexity exponential in the number of variables, limiting performance. However, when scalability is prioritized, some generality tends to be sacrificed. Existing weakly relational numerical domains [79], such as [97, 78, 77], make their own sets of restrictions.

The TVPI [97] (Two Variables Per Inequality) domain permits any linear inequalities involving up to two variables. It operates in polynomial time, but its inequality set per variable pair can still grow without bound due to arbitrary coefficients. To bound the number of inequalities by the number of variables, one way is to restrict coefficients to unit values, yielding the Unit Two Variables Per Inequality (UTVPI) domains: Zones [77] and Octagons [78] that offer a compromise between precision and cost.

In many cases, program invariants demand more expressiveness than Zones and Octagons provide, but not necessarily the full generality of TVPI. An example in C is shown in Fig. 4.1. The program takes an `array_list` that manages an array with dynamic size but fixed item size (e.g., 4 bytes). Assuming the list is full, copying an array element through function `get_at` requires computing an intermediate pointer `p` with offset `idx * isz` before access. Establishing memory safety (e.g., proving the memory safety check `valid_access` at line 12 stays within buffer `p`) requires that the given domain automatically captures the

following invariants [1]:

$$p.\mathit{offset} + 4 \leq p.\mathit{size} \ \wedge \ p.\mathit{offset} = 4 * \mathit{idx} \ \wedge \ p.\mathit{size} = l.\mathit{size} \ \wedge$$
$$0 \leq \mathit{idx} \ \wedge \ \mathit{idx} < l.\mathit{len} \ \wedge \ l.\mathit{size} = 4 * l.\mathit{len} \ \wedge \ l.\mathit{isz} = 4$$

Neither Zones nor Octagons, which merely express Unit Two Variables Per Inequality (UTVPI) constraints, can prove the check.

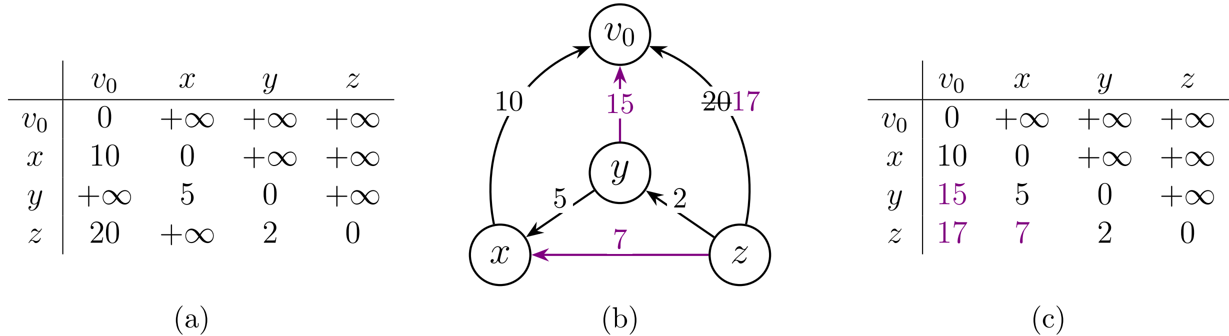
Although the generality of TVPI and Polyhedra is useful, it does not justify the computational expense. In our experience, proving the memory safety in low-level code requires reasoning about arrays that are traversed using a stride or a step. Often the size of the stride is the size of a memory word (4 or 8 bytes), or the size of a specific structure stored in the array (i.e., a specific `item_size`). In most cases, specializing the domain to deal with a few fixed coefficients, that are heuristically identified from program source code, is sufficient. Restricting the space of coefficients opens a new opportunity for designing an efficient domain that matches Zones in scalability while adding the desired precision.

We aim to extend Zones to handle TVPI constraints while leaving the complexity of the representation and operations intact. Specifically, we introduce Template DBM, a new numerical abstract domain to express Two Variables Per Inequality (TVPI) constraints with fixed coefficients. Each domain element retains the form of constraints $ax - by \leq c$, where x and y are program variables, a, b are fixed integer coefficients, and c is a constant. For example, a property constraint like `p.offset` $= 4 *$ `idx`, originally expressed by two TVPI inequalities, is encoded in UTVPI form as $\pm(p.\mathit{offset} - 4 \cdot \mathit{idx}) \leq 0$, where $4 \cdot \mathit{idx}$ is treated as a ghost variable [16] $4\mathit{idx}$. Therefore, TVPI constraints in UTVPI form, as used by Zones, can capture complex properties like array indexing. We fix the coefficients for representation as a template, so the cost of each operation is inherently bound by the number of variables and the size of the template.

In summary, Template DBM is more precise than Zones but less general than TVPI, since it only supports a subset of TVPI constraints. To validate our contributions, we built Template DBM in Crab [55] and evaluated it in terms of scalability and precision. The evaluation results show that the performance of Template DBM is comparable to that of Zones and the precision measured by the number of memory safety checks validated is close to that of Polyhedra.

The chapter is organized as follows. Section 4.2 covers necessary definitions and operations of the Zones domain. Section 4.3 introduces a strategy for encoding TVPI constraints

[1] We assume a pointer value with extra information [110]: `offset` is the position of the referred object, `size` is the object size. For brevity, we write $l.\mathit{len}$ to mean `l->len` for field access.



	v_0	x	y	z
v_0	0	$+\infty$	$+\infty$	$+\infty$
x	10	0	$+\infty$	$+\infty$
y	$+\infty$	5	0	$+\infty$
z	20	$+\infty$	2	0

(a)

(b)

	v_0	x	y	z
v_0	0	$+\infty$	$+\infty$	$+\infty$
x	10	0	$+\infty$	$+\infty$
y	15	5	0	$+\infty$
z	17	7	2	0

(c)

Figure 4.2: (a) a DBM example m, its (b) potential graph, and (c) the normal form after applying closure.

in a *difference bound matrix* and provides algorithms for saturation. Section 4.4 describes the core operations of Template DBM. Section 4.5 presents the implementation and experimental evaluation. Finally, Section 4.6 discusses related work.

4.2 Difference Bound Matrice and **Zones**

In this section, we discuss the necessary preliminaries for the Zones [77] domain, including key definitions and important operations used in this chapter.

Given a set of N program variables $\mathcal{V} = \{x, y, z, \ldots\}$ and a set of integer numbers $\mathbb{Z}$ extended by infinity $+\infty$, Zones represents a UTVPI constraint system $\mathcal{U}$ over $\mathcal{V}$ of the form: $\{x - y \leq c \mid x, y \in \mathcal{V} \wedge c \in \mathbb{Z} \cup \{+\infty\}\}$. The common data structure to encode UTVPI constraints is a Difference Bound Matrix (DBM), where rows and columns correspond to variables, and each entry represents an UTVPI constraint. For example, the constraint $x - y \leq 3$ is represented in a matrix m with $m_{xy} = 3$[2].

Typically, DBM adds an auxiliary variable v_0 that always takes the value 0 to encode unary bounds such as $\pm x \leq c$. That is, $x(-v_0) \leq c$ as $m_{x,v_0} \leq c$, and $(v_0) - x \leq c$ as $m_{v_0,x} \leq c$. We then work over the extended variable set $\mathcal{V}_0 = \mathcal{V} \cup \{v_0\}$, so a DBM has dimensions $(N+1) \times (N+1)$. For brevity, we suppose the variables are integers[3] and ignore explicitly describing constraints with infinite value in the rest of the chapter.

An alternative view of a DBM is as a *weighted directed graph*, where vertices correspond to variables and edges with weights represent constraints. For instance, an edge $x \to y$

[2]Our indexing convention is the transpose of that in [77].

[3]While DBMs can be defined over rationals, this chapter focuses exclusively on the integer case.

Example 4.2.1 (A DBM example).
Consider a set of UTVPI constraints over variables v_0, x, y, z:

$$x\ (-v_0) \leq 10 \qquad y - x \leq 5 \qquad z - y \leq 2 \qquad z\ (-v_0) \leq 20$$

A DBM m reresenting these constraints is shown in Fig. 4.2a. Each constraint is encoded as an entry to represent upper bound of the difference. If no constraint exists between variables, the entry is set to $+\infty$.

Algorithm 4.1 DBM saturation.

1: **function** DbmClosure(m)
2:     **for** $v \in \mathcal{V}_0$ **do**
3:         $m_{vv} := 0$
4:     **for** $k \in \mathcal{V}_0$ **do**
5:         **for** $i \in \mathcal{V}_0$ **do**
6:             **for** $j \in \mathcal{V}_0$ **do**
7:                 $m_{ij} := \min(m_{ij},\, m_{ik} + m_{kj})$

with a weight c represents the constraint $x - y \leq c$. In this way, we can represent a DBM compactly by storing only finite bounds (omitting $+\infty$ entries). The graph corresponding to Example 4.2.1 is shown in Fig. 4.2b.

As shown in Example 4.2.1, the set of UTVPI constraints *entails* the following consequent constraints:

$$y \leq 15 \quad z - x \leq 7 \quad z \leq 17 \quad x - x \leq 0 \quad y - y \leq 0 \quad z - z \leq 0 \quad v_0 - v_0 \leq 0$$

However, m only represents $z \leq 20$, even though $z \leq 17$ is implied. Since many different DBMs can represent the same constraint set, we want to keep a DBM that represents the tightest possible constraints between variables (i.e., a canonical representation). This representation of m is shown in Fig. 4.2c and is called the *normal / closed form* of the DBM. We refer to this procedure as *saturation* or *closure*.

Algorithm 4.1 computes a normal form of a DBM m. The algorithm saturates m with all constraints that are *logically implied* by the existing constraints. All implicit constraints can be derived by repeatedly applying transitivity:

$$\frac{\mathcal{U} \vdash x - y \leq m_{xy} \quad \mathcal{U} \vdash y - z \leq m_{yz}}{\mathcal{U} \vdash (x - y) + (y - z) \leq m_{xy} + m_{yz}}\ \textsc{Resultant}$$

Algorithm 4.2 DBM unsatisfiability check.

```
1: function DbmEmpty(m)
2:     DbmClosure(m)
3:     for i ∈ V_0 do
4:         if m_ii < 0 then
5:             m := ⊥^DBM
```

which follows the Fourier–Motzkin elimination to remove the intermediate variable y. In general, any transitive sequence of applying such rule as a dependency chain. The termination is guaranteed by a finite number of executions since the maximum dependency chain across the matrix has $N+1$ inequalities, for example:

$$(x - v_0) + (y - x) + \cdots + (z - w) + (v_0 - z) \leq m_{xv_0} + m_{yx} + \cdots + m_{zw} + m_{v_0z}$$

After the outer loop of line 4.1.4 finishes, the resulting DBM is closed. Equivalently, as a matrix can be alternatively viewed as a potential graph, closure is exactly an all-pairs shortest path (Floyd–Warshall) algorithm, where each iteration relaxes paths that are allowed to use the variable k as an intermediate vertex. The running time is $O(N^3)$.

Definition 4.2.1 (Closed DBM)**.** *A DBM m over the variable set $\mathcal{V}_0 = \{v_0, x, y, \ldots\}$ is* closed *iff:*

- $\forall i \in V_0,\ m_{ii} = 0$
- $\forall i, j, k \in V_0,\ m_{ij} \leq m_{ik} + m_{kj}$

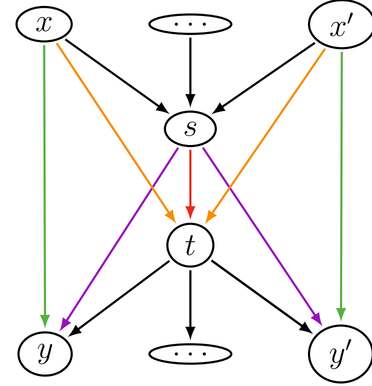
A DBM is *unsatisfiable* (*empty*) if the system of inequalities it represents has a logical contradiction (no solution). A trivial witness is $\exists x \in \mathcal{V}_0,\ x - x \leq c$ with $c < 0$, which is impossible since $x - x = 0$. More importantly, inconsistency is not always syntactically visible in a non-closed DBM. Thus, as shown in Algorithm 4.2, we first run the closure algorithm to ensure the DBM is closed and then check if any diagonal entry is negative. If so, we present a special element $\bot^{DBM}$ to denote the unsatisfiable DBM.

Zones forms a complete lattice $(\mathcal{M}, \sqsubseteq^{DBM}, \sqcup^{DBM}, \sqcap^{DBM}, \bot^{DBM}, \top^{DBM})$ where $\mathcal{M}$ is the set of DBMs over $\mathcal{V}$. The top element $\top^{DBM}$ represents the unconstrained DBM (i.e., all entries in the matrix are $+\infty$). In the following, we define the rest of lattice operations and the concretization function. All of them are computed entrywise.

Algorithm 4.3 DBM incremental closure.

```
1: function IncrementalDbmClosure(m, s, t, c)
Require: A closed m; new s − t ≤ c
Ensure: A closed and updated DBM
2:     W := {t}                              ▷ A worklist W
3:     if c < m_st then
4:         m_st := c
5:         for y ∈ V_0 do
6:             if m_st + m_ty < m_sy then
7:                 m_sy := m_st + m_ty
8:                 W := W ∪ {y}
9:         for x ∈ V_0 do
10:            for w ∈ W do
11:                if m_xs + m_sw < m_iw then
12:                    m_xw := m_xs + m_sw
13:    return m
```

(a)



(b)

Figure 4.3: (a) Incremental DBM closure algorithm; (b) Graph-based visualization.

Definition 4.2.2 (Concretization)**.** *Given a DBM m over the variable set $\mathcal{V}_0$, the concretization function γ^{DBM} of a DBM m is the set of all points in $\mathbb{Z}^n$ whose coordinates satisfy the bounds expressed in m. Formally:*

$$\gamma^{DBM}(m) \triangleq \{(s_1, \ldots, s_n) \in \mathbb{Z}^n \mid \forall i, j \in [0..n], s_i - s_j \leq m_{v_i v_j}\}$$

where each s_i (s_j) denotes the value assigned to variable v_i (v_j) and $s_0 = 0$.

Partial Order $\sqsubseteq^{DBM}$. The inclusion ordering is defined pointwise on the matrix entries. Formally, given two DBMs m and n over the same variable set $\mathcal{V}_0$ (or same dimensions), $m \sqsubseteq^{DBM} n$ iff $\forall x, y \in \mathcal{V}_0, m_{xy} \leq n_{xy}$. To compare two DBMs, the procedure takes $O(N^2)$ time.

Join $\sqcup^{DBM}$. The join of two DBMs m and n is defined as the pointwise maximum of their entries. Formally, $m \sqcup^{DBM} n$ computes a new DBM l whose entries match $\forall x, y \in$

$\mathcal{V}_0, l_{xy} := \max(m_{xy}, n_{xy})$. The operation runs in $O(N^2)$ time.

Meet $\sqcap^{DBM}$. The meet as intersection follows the pointwise minimum of the entries. Formally, $m \sqcap^{DBM} n$ computes l with entries $\forall x, y \in \mathcal{V}_0, l_{xy} := \min(m_{xy}, n_{xy})$. The operation takes $O(N^2)$ time.

The standard DbmClosure has a cubic $O(N^3)$ worse-case running time, which is inefficient for adding or updatring matrix when there are just a few new constraints added, especially when the transfer function for assignments or guards. Moreover, analyses over Zones typically maintain DBMs in closed form, because key operations, such as inclusion checks $\sqsubseteq^{DBM}$, join, and meet, are simplest (and most precise) when applied to closed DBMs. Therefore, it is preferable to have a special closure algorithm, called *incremental closure*, that restores closure only on affecting entries when a new constraint is added. The work of [7] describes the details of this algorithm and proof, we summarize it in Fig. 4.3a.

The algorithm takes a closed DBM m and a new UTVPI constraint $s - t \leq c$ as inputs. We show the visualization of the algorithm in Fig. 4.3b. It first checks if the new constraint is strictly tighter than the current entry m_{st}. If so, line 4.3a.4 updates the entry for this new constraint (the corresponding edge shown in red). The algorithm then updates other affected entries since the new constraint may tighten others. The only affected entries are variables that are connected to s or t. That is, any constraint that include s or t. To do so, the algorithm first combines $s - t \leq c$ with the existing constraint $t - y \leq m_{ty}$ to derive a new constraint $s - y \leq c + m_{ty}$, if this is a tighter one, the algorithm updates m_{sy} in line 4.3a.7 (edges shown in purple) and adds y to the worklist W. Note that the worklist contains all variables $w \in W$ where m contains UTVPI constraints $s - w \leq m_{sw}$. After finalizing W, the algorithm updates the matrix entries (in line 4.3a.12) by combining $x - s \leq m_{xs}$ and $s - w \leq m_{sw}$, to derive $x - w \leq m_{is} + m_{sw}$ (edges shown in orange and green). The algorithm runs in $O(N^2)$ time, which is more efficient than the full closure shown in Algorithm 4.1.

We have described the necessary preliminaries for the DBM and Zones domain. In the following sections, we introduce our new domain Template DBM based on an extension of the DBM to represent TVPI constraints with fixed coefficients.

4.3 Template DBMs

In this section, we present a new DBM, tDBM (for *template* DBM), that extends the classical DBM for representing TVPI constraints. For any TVPI constraint $ax - by \leq c$

with non-unit coefficients (i.e., $max\{a,b\} > 1$), tDBM introduces extra dimensions for variables with non-unit factors and storing c at the ax, by entry. For example, tDBM associates a dimension for $4idx$ to capture a constraint like $4 * idx - y \leq 0$. These extra dimensions depend on a predefined coefficient template $\mathcal{T}$. We assume $\mathcal{T} = \{a, b, \ldots\}$ with size $|\mathcal{T}| = K$ and always include 1 in $\mathcal{T}$. As usual, we use v_0 with value 0 for $\pm ax \leq c$.

tDBM falls between the system $\mathcal{U}$ and a TVPI constraint system $\mathcal{I} \stackrel{\text{def}}{=} \{ax - by \leq c \mid x, y \in \mathcal{V} \wedge a, b \in \mathbb{Z}^{\geq 0} \wedge c \in \mathbb{Z}\}$ where a, b are positive coefficients. Specifically, a tDBM represents a constraint system $\mathcal{I}_{\mathcal{T}} \subset \mathcal{I}$ over variables $\mathcal{V}$ and coefficients $\mathcal{T}$ of the form $\{ax - by \leq c \mid x, y \in \mathcal{V} \wedge a, b \in \mathcal{T} \wedge c \in \mathbb{Z}\}$.

Fig. 4.4 shows how a tDBM represents the TVPI constraints necessary to guarantee that the `memcpy` (i.e., the assertion on line 4.1:12) access buffer `p` remains within bounds. The given tDBM extends two extra dimensions for expressing relations for $4idx$ and $4l.len$. To improve readability, we decompose the tDBM $\bar{m} := (m, m^+)$ into two submatrices: a classical sub-DBM, m, that covers dimensions for UTVPI constraints $x - y \leq c$, and an extended sub-DBM, m^+, that handles TVPI constraints with fixed non-unit coefficients $ax - by \leq c$.

The constraint $p.offset - p.size \leq -4$ (denoted as c) is needed to prove the assertion on line 4.1:12 is valid. However, this constraint (and all constraints colored in purple) is implicit and can only be inferred from the constraints colored in green. Resolving c requires inferring $c2$: $p.offset - l.size \leq -4$, where $c2$ also needs another constraint $c3$: $p.offset - 4l.len \leq -4$ to be implied. Section 4.3.1 shows how to compute those implicit constraints.

For the rest of the chapter, we use $row(ax)$ and $col(ax)$ to refer to the row and column indexes, respectively, in the $\bar{m}$ associated with variable x and coefficient a. To access the item in $\bar{m}$, we use $\bar{m}_{row(ax)col(by)}$ or simply $\bar{m}_{ax,by}$ to refer to the difference bounds for inequality $ax - by \leq \bar{m}_{ax,by}$. For elements represented for UTVPI constraints, we write $\bar{m}_{x,y}$. For elements of other TVPI constraints, we write $\bar{m}_{ax,by}$, $\bar{m}_{ax,y}$ or $\bar{m}_{x,by}$. To access the difference bound for any TVPI constraint l, we denote it as $\bar{m}_l$.

A single TVPI constraint can be represented in a variety of equivalent ways. For example, $2x - y \leq 3$ is equivalent to $4x - 2y \leq 6$, $6x - 3y \leq 9$, etc. To keep as many constraints as possible in tDBM, we normalize each TVPI constraint $ax - by \leq c$ into a *template expressible form* so that its coefficients fit the coefficient template $\mathcal{T}$ (if possible). This is justified by the following inference:

$$\frac{\begin{array}{c}\mathcal{I} \vdash ax - by \leq c \quad d \in \mathbb{Z}^{>1} \quad (d \mid a) \quad (d \mid b) \\ a' = a/d \quad a' \in \mathcal{T} \quad b' = b/d \quad b' \in \mathcal{T} \quad c' = \lceil c/d \rceil \quad c' \in \mathbb{Z}\end{array}}{\mathcal{I}_{\mathcal{T}} \vdash a' \cdot x - b' \cdot y \leq c'} \textsc{Scaling}$$

$m:$

$$-idx \leq 0 \qquad -l.len \leq -1$$
$$4 \leq l.isz \leq 4 \qquad idx - l.len \leq -1$$
$$p.size - l.size \leq 0 \qquad l.size - p.size \leq 0$$
$$p.offset - p.size \leq -4 \qquad p.offset - l.size \leq -4$$

$m^+:$

$$p.offset - 4idx \leq 0 \qquad 4idx - p.offset \leq 0$$
$$4l.len - l.size \leq 0 \qquad p.offset - 4l.len \leq -4$$
$$l.size - 4l.len \leq 0$$

(a)

$m:$

	v_0	$l.isz$	$l.len$	$l.size$	idx	$p.offset$	$p.size$
v_0	$+\infty$	-4	-1	$+\infty$	0	$+\infty$	$+\infty$
$l.isz$	4	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$l.len$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$l.size$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	0
idx	$+\infty$	$+\infty$	-1	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$p.offset$	$+\infty$	$+\infty$	$+\infty$	-4	$+\infty$	$+\infty$	-4
$p.size$	$+\infty$	$+\infty$	$+\infty$	0	$+\infty$	$+\infty$	$+\infty$

$m^+:$

	$p.offset$	$4idx$	$4l.len$	$l.size$
$p.offset$	$+\infty$	0	-4	$+\infty$
$4idx$	0	$+\infty$	$+\infty$	$+\infty$
$4l.len$	$+\infty$	$+\infty$	$+\infty$	0
$l.size$	$+\infty$	$+\infty$	0	$+\infty$

(b)

Figure 4.4: (a) DBM-related constraints and (b) a tDBM $\bar{m}$.

The rule scales the coefficients by a common divisor d to produce an equivalent inequality, where the new coefficients a' and b' are elements of $\mathcal{T}$. For example, for $\mathcal{T} = \{1, 2, 3, 4\}$, the rule scales the constraint $8x - 4y \leq 8$ to $2x - y \leq 2$ (divided by 4). In practice, we keep just one equivalent form, as others like $4x - 2y \leq 4$ are ignored. The purpose of this rule is to convert TVPI constraints into the expressible forms before matrix updates.

4.3.1 Saturation

To achieve a full (closed) representation, we *saturate* tDBM by exhaustively deriving all implicit constraints until a fixpoint is reached. We present a saturation algorithm for tDBM based on Fourier-Motzkin variable elimination. Each implicit inequality is deduced following this rule:

$$\frac{\mathcal{I} \vdash ax - by \leq c \quad \mathcal{I} \vdash dy - ez \leq f \quad g = gcd(b, d) \quad \lambda_1 = d/g \quad \lambda_2 = b/g}{\mathcal{I} \vdash (\lambda_1 a) \cdot x - (\lambda_2 e) \cdot z \leq \lambda_1 c + \lambda_2 f} \text{ Resultant}$$

Algorithm 4.4 Nelson-driven [86] tDBM saturation.

```
1: function NelsonTvpiSaturation(m̄)
2:     for i ∈ {0, . . . , ⌈lg(N)⌉ − 1} do
3:         for x, y, z ∈ V do
4:             for a, b, d, e ∈ T do
5:                 c := m̄_{ax,by}, f := m̄_{dy,ez}
6:                 if a′x − b′z ≤ c′ = ScaledResultant(ax − by ≤ c, dy − ez ≤ f) then
7:                     m̄ := m̄ ∪ {a′x − b′z ≤ c′}
```

which eliminates variable y and yields a new inequality. For instance, consider the two TVPI constraints $2x - 3y \leq 5$ and $9y - 2z \leq 5$. Eliminating y through the rule yields $6x - 2z \leq 20$. If this constraint cannot fit the coefficient template (e.g., $\mathcal{T} = \{1, 2, 3, 9\}$), we apply SCALING rule to rewrite it as $3x - z \leq 10$.

Algorithm 4.4 saturates an input tDBM by iteratively applying RESULTANT to pairs of existing inequalities until the iteration limit $\lceil lg(N) \rceil - 1$ is reached. We cap this bound since by then applying the RESULTANT guarantees a contradiction witness [86]. The RESULTANT is extended as SCALEDRESULTANT which applies SCALING after to produce an expressible form. The time complexity is $O(K^4 N^3 lg(N))$ when a matrix is dense.

Since the tDBM does not express arbitrary TVPI constraints, the algorithm only introduces implicit inequalities within the available dimensions or tightens the existing ones. A tDBM $\bar{m}$ as *complete* if and only if, for every constraint c over a variable set U that $\bar{m}$ satisfies, the projection of $\bar{m}$ (i.e., restricting constraints) to U still satisfies c.

During the saturation process, however, Algorithm 4.4 cannot guarantee the result tDBM is complete, since the derived constraints may not be expressible in the tDBM. For example, for $\mathcal{T} = \{1, 2, 3, 4\}$, a tDBM $\bar{m}$ represents $\{2x - 3y \leq 6 \wedge y - 4z \leq 8 \wedge 3z - 2w \leq 7\}$. Despite the coefficient template, applying the RESULTANT rule iteratively derives a new set of constraints $\{x - 6z \leq 15, 3y - 8w \leq 52, x - 4w \leq 29\}$ at the fixpoint. However, to derive $x - 4w \leq 29$, tDBM requires representing either $x - 6z \leq 15$ or $3y - 8w \leq 52$. Thus, $\bar{m}$ is not complete.

Choosing the coefficient template $\mathcal{T}$ is crucial for completeness. For instance, if the template contains coefficients that are powers of two, running the Algorithm 4.4 guarantees that all implicit constraints are derived and representable.

Theorem 4.3.1. *Given a tDBM $\bar{m}$, Algorithm 4.4 computes a complete tDBM $\bar{m}'$ under the TVPI system $\mathcal{I}_{\mathcal{T}}$ if and only if all implicit constraints are expressible.*

Proof. Since all implicit constraints are expressible, Algorithm 4.4 strictly follows Lemma 1 in [86]. □

While Algorithm 4.4 does not promise full completeness, it is sufficient for our purposes. Consider the example shown in Fig. 4.4, to derive the marked constraint $p.\mathit{offset} - p.\mathit{size} \leq -4$, below are the constraints derived at each iteration:

1. $p.\mathit{offset} - 4idx \leq 0$ and $idx - l.\mathit{len} \leq -1$ derives $p.\mathit{offset} - 4l.\mathit{len} \leq -4$.
2. $p.\mathit{offset} - 4l.\mathit{len} \leq -4$ and $4l.\mathit{len} - l.\mathit{size} \leq 0$ produces $p.\mathit{offset} - l.\mathit{size} \leq -4$.
3. $p.\mathit{offset} - l.\mathit{size} \leq -4$ and $l.\mathit{size} - p.\mathit{size} \leq 0$ gives $p.\mathit{offset} - p.\mathit{size} \leq -4$.

The Resultant rule specializes to the standard DBM closure whenever $b = d$, and this closure is not limited to UTVPI constraints. For example, any pair of constraints such as $2x - 3z \leq 4$ and $3z - y \leq -1$ can also use the DBM closure to derive $2x - y \leq 3$. This gives us the opportunity to reuse that routine and to build a tDBM saturation on top of it. Overall, we iteratively apply Resultant by decomposing it into two steps:

1. $(b \neq d)\ ax - by \leq c \wedge dy - ez \leq f \xrightarrow{\text{ScaledResultant}} a'x - b'z \leq c'$
2. $(b = d)\ ax - by \leq c \wedge dy - ez \leq f \xrightarrow{\text{DbmClosure}} ax - ez \leq c + f$

Step 1 aligns TVPI constraint coefficients to eliminate y. Step 2 reuses the standard DBM closure. Accordingly, Algorithm 4.5 shows a tDBM version in which TvpiReduce handles the first step and DbmClosure operates the second. By Theorem 4.3.2, algorithm is equivalent to Algorithm 4.4.

Theorem 4.3.2. *Given a tDBM $\bar{m}$,*

$$\text{NelsonTvpiSaturation}(\bar{m}) \equiv \text{DbmTvpiSaturation}(\bar{m})$$

Proof. For any input tDBM $\bar{m}$, Algorithm 4.5 repeatedly invokes TvpiReduce (with coefficient alignment) and DbmClosure (no alignment). This is identical to applying the Resultant rule in Algorithm 4.4. Even the new derived constraint can be used directly during iteration i, with the guarantee of the loop upper bound $\lceil lg(N) \rceil - 1$, ensuring that all implicit constraints are ultimately captured in the matrix, regardless of the order of steps (earlier or later). □

Algorithm 4.5 A DBM closure based saturation for tDBM.

```
1: function TvpiReduce(m)
2:     for x, y, z ∈ 𝒱 do
3:         for a, b, d, e ∈ 𝒯 ∧ b ≠ d do
4:             c := m̄_{ax,by}, f := m̄_{dy,ez}
5:             if a′x − b′z ≤ c′ = ScaledResultant(ax − by ≤ c, dy − ez ≤ f) then
6:                 m̄ := m̄ ∪ {a′x − b′z ≤ c′}
7:
8: function DbmTvpiSaturation(m̄)
9:     for i ∈ {0, ..., ⌈lg(N)⌉ − 1} do
10:        TvpiReduce(m̄)
11:        DbmClosure(m̄)
```

As the above example shows, the first implicit constraint $p.\mathit{offset} - 4l.\mathit{len} \leq -4$ is derived from TvpiReduce, and the other two from DbmClosure.

Our purpose is extending DBM to support TVPI constraints using a small coefficient template. Experiments in Section 4.5 show that using three active coefficients suffices, and TVPI constraints with non-unit coefficients remain few compared with UTVPI constraints. Thus, running TvpiReduce is cheap, and DbmClosure runs nearly as efficiently when only UTVPI constraints are present.

4.3.2 Incremental Saturation

Existing abstract domains for DBM apply an incremental closure [46, 80, 7, 15] to restore DBM in closed form after each assignment or constraint strengthening, making this procedure the dominant use. By updating only the affected entries in the DBM, the algorithm runs more efficiently than the full saturation (closure) algorithm. In this section, we present a worklist-based procedure that incrementally applies the Resultant rule to propagate new constraints.

Algorithm 4.6 gives the pseudocode for incremental saturation. It adds the new constraint $ax - by \leq c$ into $\bar{m}$. Once the input constraint is tighter, the algorithm first collects all existing constraints involving y (in positive occurrence) and applies the Resultant rule to eliminate it, as shown in the purple box, then repeats for x (orange box). Each elimination step produces new constraints that contain only one of the two variables, x or y; we store them in the worklists W_x and W_y, respectively. We present a graph representation (shown in Fig. 4.5a) that illustrates and highlights the edges added to each

Algorithm 4.6 Incremental saturation for tDBM.

```
1: function TvpiIncrementalSaturation(m̄, ax − by ≤ c)
2:     m̄ := m̄ ∪ {ax − by ≤ c}
3:     W_x := {}, W_y := {}
4:     for d, e, w ∈ T × T × V do                                   ▷ successors related to y
5:         if a′x − e′w ≤ c′ = ScaledResultant(ax − by ≤ c, dy − ew ≤ m̄_{dy,ew}) then
6:             m̄ := m̄ ∪ {a′x − e′w ≤ c′}
7:             W_x := W_x ∪ {a′x − e′w ≤ c′}
8:     for d, e, z ∈ T × T × V do                                   ▷ predecessors related to x
9:         if e′z − b′y ≤ c′ = ScaledResultant(ez − dx ≤ m̄_{ez,dx}, ax − by ≤ c) then
10:            m̄ := m̄ ∪ {e′z − b′y ≤ c′}
11:            W_y := W_y ∪ {e′z − b′y ≤ c′}
12:    for ez − by ≤ c ∈ W_y do
13:        for d, g, w ∈ T × T × V do                               ▷ successors related to y
14:            if e′z − g′w ≤ c′ = ScaledResultant(ez − by ≤ c, dy − gw ≤ m̄_{dy,gw}) then
15:                m̄ := m̄ ∪ {e′z − g′w ≤ c′}
16:    for ax − ew ≤ c ∈ W_x do
17:        for d, g, z ∈ T × T × V do                               ▷ predecessors related to x
18:            if g′z − e′w ≤ c′ = ScaledResultant(gz − dx ≤ m̄_{gz,dx}, ax − ew ≤ c) then
19:                m̄ := m̄ ∪ {g′z − e′w ≤ c′}
```

list in their corresponding colors. Next, we reuse those derived constraints to completely eliminate either y or x (green box). As a result, none of the new constraints include y and x. Specifically, we derive constraints between z and w through the transitive chain $\{z, x\}, \{x, y\}, \{y, w\}$ (see the green edge in Fig. 4.5a).

To illustrate how Algorithm 4.6 works, given a set of constraints with no implicit ones, represented by a tDBM over the coefficient set $\mathcal{T} = \{1, 4\}$:

$$\begin{array}{lll} idx - len \leq -1 & size - 4len \leq 0 & 4len - size \leq 0 \\ 4idx - 4len \leq -4 & 4idx - size \leq -4 & \end{array}$$

Suppose we add $ofs - 4idx \leq 0$ and perform incremental saturation. The algorithm finds all inequalities involving idx and ofs: $idx - len \leq -1$, $4idx - 4len \leq -4$, and $4idx - size \leq -4$. Applying the Resultant rule to eliminate idx yields $ofs - 4len <= -4$

and $ofs - size \leq -4$. With no existing constraints involving ofs, no further constraints can be derived, and the tDBM is once again closed.

Lemma 4.3.1. *Suppose a tDBM $\bar{m}$ is closed under system $\mathcal{I}_{\mathcal{T}}$, adding a new constraint $l = ax - by \leq c$ by Algorithm 4.6 yields $\bar{m}'$ which is satisfiable if and only if:*

1. $\bar{m}'_{ax,by} \leq c$
2. $\exists o \in \bar{m}, dx - ew \leq f := \textsc{ScaledResultant}(l, o), \bar{m}'_{dx,ew} \leq f$
3. $\exists o \in \bar{m}, dz - ey \leq f := \textsc{ScaledResultant}(o, l), \bar{m}'_{dz,ey} \leq f$
4. *for any constraint $dz - ew \leq f$ derived through the transitivity chain: $\{z, x\}, l, \{y, w\}$, $\bar{m}'_{dz,ew} \leq f$*

Theorem 4.3.3. *Given a tDBM $\bar{m}$ and a new constraint $ax - by \leq c$, Algorithm 4.6 computes a complete tDBM $\bar{m}'$ under the TVPI system $\mathcal{I}_{\mathcal{T}}$ if and only if all implicit constraints are expressible.*

Proof. Since all implicit constraints are expressible, Algorithm 4.6 strictly follows Lemma 4.3.1. □

The time complexity is $O(K^4 N^2)$. The correctness of Algorithm 4.6 is guaranteed by the Lemma 4.3.1. Due to the limitation of the coefficient template, running Algorithm 4.6 guarantees completeness if all implicit constraints are derived and representable in the tDBM. After incremental saturation, contradictions can be detected by identifying negative cycles (Section 4.4).

4.4 Template DBM Abstract Domain

Template DBM is a new weakly relational domain focused on inferring and checking program properties expressible as TVPI constraints. It is based on tDBM, where each constraint $ax - by \leq c$ corresponds to a specific entry in the matrix.

The concretization function γ^{tDBM} maps a tDBM $\bar{m}$ to a set of all possible values assigned for variables that satisfy all potential constraints in $\bar{m}$. Formally:

$$\gamma^{tDBM}(\bar{m}) \triangleq \{(s_1, \ldots, s_n) \in \mathbb{Z}^n \mid \forall i, j \in [1..n], a, b \in \mathcal{T}, a \cdot s_i - b \cdot s_j \leq \bar{m}_{av_i bv_j}\}$$

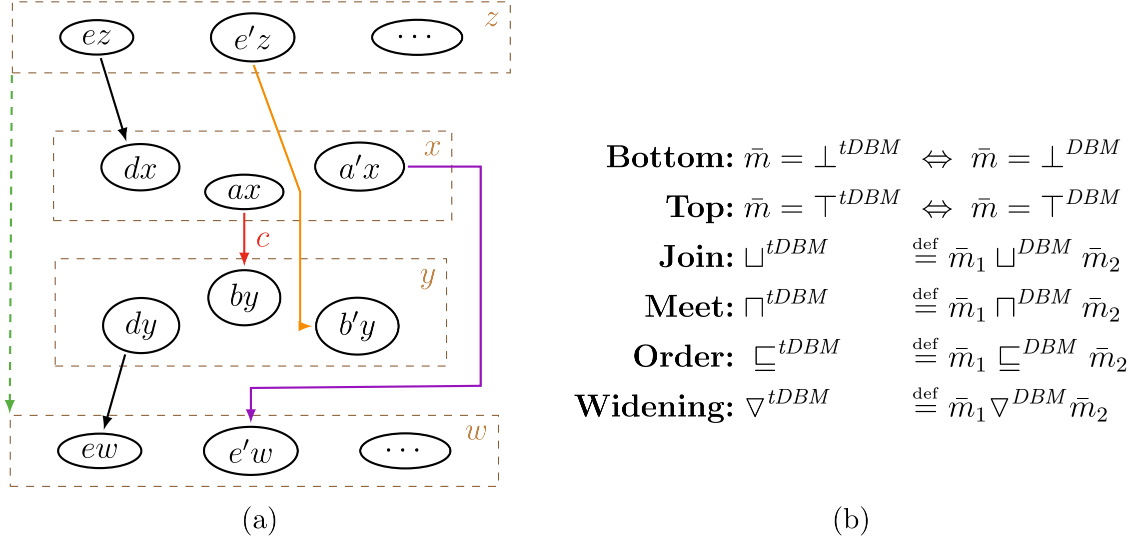


Figure 4.5: (a) Graph view of the tDBM update after adding $ax - by \leq c$ with new edges highlighted and the green dashed edge marking all implicit constraints between z and w; (b) The lattice operations of Template DBM.

s_i (s_j) represents a value for a variable v_i (v_j).

A tDBM $\bar{m}$ is unsatisfiable (empty) when repeatedly applying the RESULTANT rule yields a contradiction. For example, a tDBM $\bar{m}$ with coefficient template $\mathcal{T} = \{1, 2, 3, 4, 5\}$:

$$2y - x \leq 3 \qquad 5x - 3z \leq -22 \qquad 2z - 5p \leq -2 \qquad 3p - 4y \leq 0$$

It is unsatisfiable because an implicit inequality $3p - 2x \leq 6$ contradicts another implicit one, $2x - 3p \leq -10$, following:

1. applying RESULTANT rule for $3p - 4y \leq 0$ and $2y - x \leq 3$ derives $3p - 2x \leq 6$.
2. from $5x - 3z \leq -22$ and $2z - 5p \leq -2$, RESULTANT derives $2x - 3p \leq -10$.

As Nelson [86] showed that iterating the RESULTANT rule through saturation Algorithm 4.5 exposes a contradiction. Thus, the unsatisfiable check boils down to inspecting a saturated tDBM $\bar{m}$: $\exists\, x, y \in \mathcal{V},\ a, b \in \mathcal{T}:\ \bar{m}_{ax,by} + \bar{m}_{by,ax} < 0$.

A tDBM is bottom (resp. top) if and only if its DBM is bottom (resp. top). For other domain operations, we leverage DBM operations for efficiency. All are defined in Fig. 4.5b. Each operation performs element-wise matrix updates and runs in quadratic

Algorithm 4.7 Transfer function for assignment.

```
1: function TvpiAssign(m̄, ⟦x := e⟧)
2:     T := {}
3:     if e = a_1·x_1 + a_2·x_2 + ... + a_n·x_n + c ∧ ∀i ∈ [1..n] : a_i ∈ Z^{≥0} then
4:         for i ∈ [1..n] do
5:             e_{1i} := Σ_{j≠i} a_j·x_j + (a_i − 1)·x_i + c                ▷ dropping x_i
6:             T := T ∪ {y − x_i ≤ e^+_{1i}; x_i − y ≤ e^-_{1i}}
7:             if a_i ∈ 𝒯 then
8:                 e_{ai} := Σ_{j≠i} a_j·x_j + c                          ▷ dropping a_i·x_i
9:                 T := T ∪ {y − a_i x_i ≤ e^+_{ai}; a_i x_i − y ≤ e^-_{ai}}
10:    else
11:        T := T ∪ {x ≤ e^+, −x ≤ −e^-}
12:    for t ∈ T do
13:        m̄ := TvpiIncrementalSaturation(m̄, t)
```

time $O(K^2N^2)$ at worst. All domain operations are safe approximations, but not the best (except for meet). For example, $\sqcup^{tDBM}$ combines two tDBMs by taking the element-wise maximum of their entries. However, a more precise approximation is obtained by finding the extreme points of the convex hull and reconstructing the TVPI constraints accordingly. Consider a join of two abstract states s_1 and s_2:

$$s1 : -i \leq 0 \wedge i \leq 9 \wedge -c \leq 10 \wedge c \leq -1 \qquad\qquad s2 : i = 10 \wedge c = 0$$

The convex hull join computes the result state as $-i \leq 0 \wedge i \leq 10 \wedge -c \leq 10 \wedge c \leq 0 \wedge 10c - i \leq -10 \wedge 10i - c \leq 100$. In contrast, $s1 \sqcup^{tDBM} s2$ as $-i \leq 0 \wedge i \leq 10 \wedge -c \leq 10 \wedge c \leq 0$. While join can be designed using the convex hull algorithm, our design is simpler and takes operations from DBM directly.

The primitive operations during analysis are the addition or removal of variables. For an assignment $x := e$, we define the transfer function (see Algorithm 4.7) that handles two cases. When e is not a linear expression, we over-approximate its value by its interval $[-e^-, e^+]$; otherwise, we approximate it more precisely by using interval information to iteratively drop variables on e until the remaining constraint follows the TVPI form. We first approximate assignment in UTVPI form, since our tDBM natively represents them. Next, we attempt to convert the assignment to TVPI form. If e involves any unbounded variable, we conservatively approximate its value as $\top$. Finally, we insert each new constraint with incremental saturation to maintain closure. Although more precise approximations exist, this simple approach is effective for analyzing programs such as Fig. 4.1. As an example,

let us consider the assignment `ofs = l->isz * idx` at line 10 with a pre-abstract state:

$$l.isz \leq 4 \ \wedge \ -l.isz \leq -4 \ \wedge \ -idx \leq 0 \ \wedge \ \cdots$$

Although the expression `l->isz * idx` is non-linear, we know from the pre-state that `l->isz` is $l.isz \leq 4 \wedge -l.isz \leq -4$. It is safe to rewrite that expression as `4 * idx` and then invoke the Algorithm 4.7. The assignment thereby is abstracting as two TVPI constraints, $\pm(ofs - 4idx) \leq 0$, and one UTVPI constraint, $idx - ofs \leq 0$, since idx has a known lower bound. After incremental saturation completed, the resulting state remains closed.

For variable removal, we denote $\exists x.\bar{m}$ for eliminating all constraints on x from the tDBM $\bar{m}$. In practice, this simply means dropping all rows and columns for x (including ghost variables). To preserve precision, saturating $\bar{m}$ is required before existential quantification.

4.5 Implementation and Experimental Evaluation

We have implemented Template DBM[4] in the Crab library [55]. We reuse the implementation from [46] as the underlying DBM which is tailored to make use of a direct graph $\bar{m} := \langle V, E \rangle$. Nodes V correspond to the combination of variables and coefficients $\mathcal{V} \times \mathcal{T}$ and each constraint $ax - by \leq c$ is represented as a directed edge $ax \xrightarrow{c} by$. The graph representation avoids the $O(K^2N^2)$ space of a matrix, since inferred constraints are often quite sparse during analysis [45], especially after widening in loop-invariant computation [98]. Besides, the graph representation can efficiently perform the domain operations. For example, the join can be implemented by merging two graphs and taking the maximum weight for each edge. The inclusion check $\bar{m}_1 \sqsubseteq^{tDBM} \bar{m}_2$ can also be done by checking if all edges in $\bar{m}_2$ can be entailed by edges in $\bar{m}_1$, which can be done in linear time w.r.t the number of edges (inequalities) $|E|$. The implementation for the remaining domain operations follows algorithms discussed in Section 4.4.

For efficiency, we implement incremental saturation instead of full saturation (Algorithm 4.5) for analysis. It performs local updates on each new assignment or assumption for low amortized cost. Our implementation of Algorithm 4.6 is based on DBM incremental closure. This special version splits into two phases: first, it runs Algorithm 4.6 where the Resultant rule only applies to cases requiring coefficient alignment; then it invokes IncrementalDbmClosure with previously derived constraints to finish saturation. While this version misses constraints since implied constraints from phase two can

[4] Available at https://github.com/LinerSu/crab/tree/tvpi_dbm

be used at once, later experiments show that this version preserves good precision and performance.

In the chapter, we demonstrate how Template DBM infers constraints to check for buffer overflows in Fig. 4.1. This example is from a case study in [101], where an abstract interpreter preprocesses the program to prove and remove memory safety checks before a bounded model checker completes verification. These checks as assertions guard each memory access, and the interpreter proves them both before and after loop unrolling. We reuse this study to evaluate Template DBM performance and precision in proving memory safety, while the details of how the interpreter works are outside the scope here. We also omit discussion on interpreter effectiveness, as it has already been discussed in [101].

The benchmark suites originate from two open-source production codebases: `aws-c-sdk` and `firedancer`, with a total of 139 benchmarks[5]. Note that benchmarks drawn from `aws-c-sdk` include the `aws-c-common` component. These suites cover programs from simple arithmetic computations to complex data structure manipulations. The benchmarks also include loops to evaluate domain operation performance and precision in proving assertions before and after loop unrolling. The experiment here measures how the interpreter using different numerical domains performs as the program size varies.

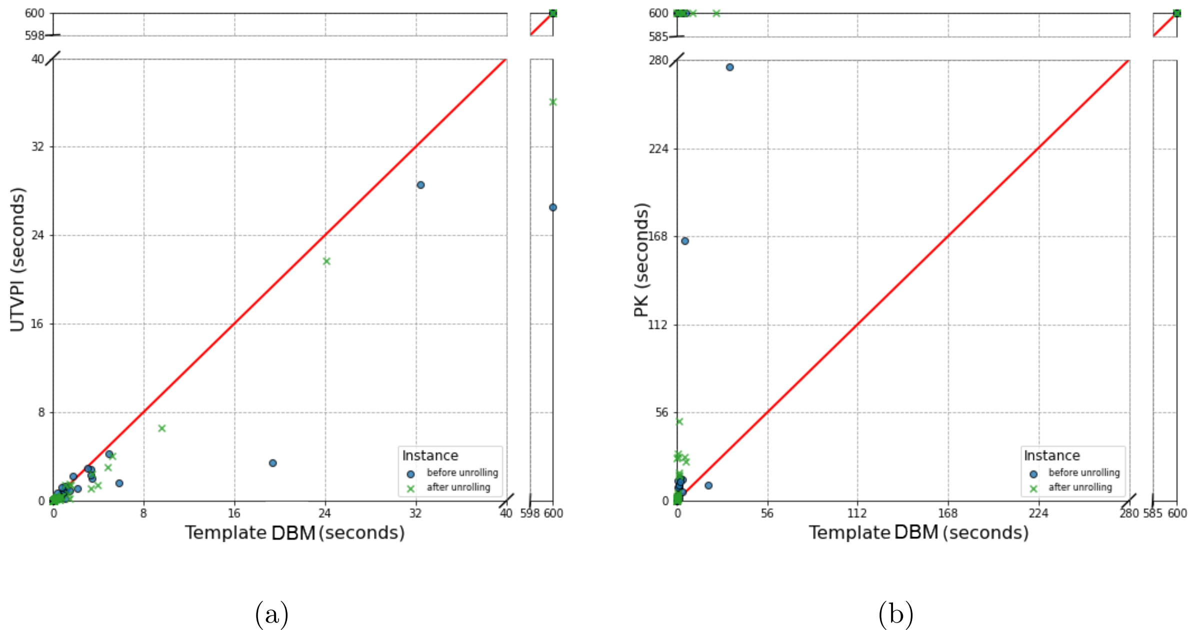
We compare Template DBM with the Zones (UTVPI) and the Polyhedra (linear inequality) domains. Because Template DBM and Zones use the same DBM implementation, Zones serves as the baseline for precision and performance comparison. We choose Polyhedra instead of the original TVPI [97] because our experimental results (as shown later) indicate that Template DBM achieves nearly the same precision as Polyhedra on most benchmarks. Besides, the TVPI implementation[6] is unmaintained. Having a direct side-by-side evaluation is challenging. For Polyhedra, we use the implementation from the ELINA library [99].

For consistency, we choose the same configurations for running all domains. Template DBM employs a predefined coefficient template $\{1, 2, 3, 4, 5, 8, 10, 16, 24, 32, 40\}$ with 11 heuristic numbers. Each task is given a timeout of 600 seconds. All experimental results are collected from a machine with an Intel Xeon E5-2680 @2.50GHz, with 256 GB RAM. The artifact and results are available at https://doi.org/10.5281/zenodo.16075045.

All performance results for analyzing each program before and after loop unrolling are shown in Fig. 4.6. Zones timed out on 5 before unrolling, and on 2 after. Template DBM has 1 more pre-unrolling timeout case than Zones. Polyhedra timed out 8 more times than Zones before unrolling and 1 more after. As shown in Fig. 4.6a, Template DBM runs

[5]Available at https://github.com/LinerSu/TVPI-Domain-Benchmarks

[6]The original implementation is at https://github.com/axel-simon/tvpi.



(a) (b)

Figure 4.6: (a) Zones vs. Template DBM and (b) Polyhedra (PK) vs. Template DBM.

slower but remains within a similar range to Zones after excluding timeouts. Before loop unrolling, Zones averages 0.2s (SD = 0.6) and Template DBM 0.4s (SD = 1.9); after loop unrolling, Zones takes 0.1s (SD = 0.5) and Template DBM 0.2s (SD = 0.7). We achieve similar running times because it extends dimensions to support TVPI constraints. Since not all variables require extra dimensions, the size of Template DBM remains comparable to Zones in most cases. However, as the graph shows one additional timeout and one major slowdown (i.e., the 19 seconds to complete), we diagnosed these and conclude that heavily dimensioned matrices harm operation speed. This limitation can be addressed in future work by implementing a more efficient join algorithm and involving a geometric approach to remove redundant constraints, such as the *filter* operation introduced from the original TVPI work [97]. In Fig. 4.6b, Polyhedra does not scale well, with 2.5s (SD = 15.0) average analysis time before unrolling and 2.2s (SD = 7.4) after unrolling. Overall, Template DBM has performance comparable to Zones and is faster than Polyhedra.

Precision is evaluated in terms of how many assertions are successfully proved in each domain. The compared results before and after loop unrolling are shown in Table 4.1. We keep assertions proved before loop unrolling instead of discharging and prevent reprov-

suite	category	Before loop unroll				After loop unroll			
		Total	Zones	tDBM	PK	Total	Zones	tDBM	PK
aws-c-sdk	array_list	24	75%	83%	83%	62	61%	65%	65%
	hash_table	498	83%	85%	85%	2171	54%	58%	58%
	others	1689	67%	67%	67%	2853	56%	59%	59%
firedancer	tango	33	36%	85%	100%	151	17%	85%	100%
	util	106	62%	62%	62%	195	75%	75%	87%
	others	270	24%	24%	11%	305	95%	95%	86%
	total	2620	65%	66%	65%	5737	57%	62%	62%

Table 4.1: Precision across Zones, Template DBM (tDBM), and Polyhedra (PK).

ing them by adding assumptions. This approach explains why the total assertions grow dramatically after unrolling.

In the table, regardless of loop unrolling stage, the number of assertions proved by Template DBM lies between Zones and Polyhedra. We are more precise than Zones because proving assertions requires TVPI constraints, which Zones cannot express. For instance, in array_list, hash_table, and tango categories, most assertion checks require constraints like $\mathit{offset} * 4 - \mathit{size} \leq 0$ with *offset* as pointer offset and *size* as object size, thus driving significantly different assertion rates across domains. Compared to the "others" category from `aws-c-sdk`, where almost all domains prove the same number of assertions since most checks only require UTVPI constraints to prove. An important observation is that analyzing each benchmark uses fewer than three coefficients from the template. Consequently, the number of non-unit coefficient TVPI constraints grows sparsely yet remains sufficient to verify most assertions.

On the other hand, Template DBM and Polyhedra solve assertions at similar rates across most benchmarks. There are 8 benchmarks where Polyhedra solves extra 9 assertions before and 59 after unrolling. Among these 8 cases, Polyhedra covers more general linear inequalities, such as $x+y+z \leq 10$, that neither Zones nor Template DBM support. Template DBM also fails to prove some assertions due to its limited support for the assignment transfer function. The implementation only handles the form $x := e$ and not scaled assignments such as $24 * x := 24 * e$, though it can be notably improved. However, these assertions represent only a small fraction of all benchmarks and assertions.

In the "others" category from `firedancer`, Polyhedra proves fewer assertions than either Zones or Template DBM. It fails on 79 assertions on 4 cases before loop unrolling and 52 across 4 cases after unrolling. ELINA library logs report coefficient-overflow and vector-

product exceptions [7] during these analyses, which cause imprecision. To isolate the ELINA flaw, we use the APRON [60] and PPL [4] Polyhedra as back-ends to verify these cases; either they got the same assertion rate as Template DBM or timed out. We therefore conclude that Polyhedra can, in theory, prove these assertions but, as Table 4.1 shows, it fails due to limitations in the ELINA implementation.

Overall, our experiment demonstrates that Template DBM offers greater scalability than Polyhedra while providing higher precision than Zones.

4.6 Related Work

We have already seen some abstract domains close to our work in Section 4.1. This section explores their deeper connections and examines alternative approaches.

The TVPI domain, originally from [97], represents arbitrary inequalities of the form $ax + by \leq c$ with $a, b, c \in \mathbb{Q}$. Our work restricts this to $ax - by \leq c$ where $a, b \in \mathcal{T}$ (a predefined coefficient template) and $c \in \mathbb{Z}$, matching the DBM structure for difference bounds. One way to extend our tDBM is to introduce dimensions ax^+ and ax^- (where $ax^+ = -ax^-$), as in the Octagons domain, to represent more general TVPI constraints $\pm ax \pm by \leq c$. However, this dimensional increase may cause a blow-up and thus degrade performance.

Our work instead prioritizes scalability, which relies on the underlying DBM operations. Template DBM performs saturation to expose all implicit constraints, directly applying the standard DBM operations without altering its structure. In contrast, the original work treats constraints as a geometric polyhedron, leading to very different design choices for domain operations.

Our work aims to limit the form of TVPI constraints by fixing the coefficient template. A similar approach has been applied in other abstract domains. Logahedra [56] is a TVPI-based domain which restricts coefficients to powers of two. The work also introduces a bounded version, which limits the exponent and thus represents a finite set of inequalities. This version preserves cubic time complexity for core operations like completion and join. Unlike Logahedra, our domain allows custom coefficients, offering a more flexible configuration since array strides are not always powers of two [8]. Template Polyhedra [95] domain fixes linear expressions ahead of time by a predefined template and tracks

[7] Issue report: https://github.com/eth-sri/ELINA/issues/39

[8] In the hash_table category, we capture TVPI constraints requiring a coefficient of 24, which is the allocation size of a C structure without alignment.

linear inequalities only for those expressions. As a result, each abstract operation runs in polynomial time relative to the number of template expressions. However, the template must be chosen heuristically at each program location, and each post-condition operation invokes a linear programming (LP) solver to compute the tightest bound for each template expression. Our approach requires only one coefficient template to restrict the TVPI form and computes post abstract states efficiently using the incremental saturation algorithm we propose whenever a new assignment or assumption is added.

The Weighted Hexagon [44] domain captures invariants of the form $x \in [-a, b] \wedge x \leq a \cdot y$ where $a, b \in \mathbb{I}^{\geq 0}$ with $\mathbb{I}$ representing reals or rationals. It features at most six edges per pair of variables and provides a transitive closure algorithm in cubic time. However, it is less expressive because $x \leq a \cdot y$ relations are limited to only the maximal and minimal slopes, and the domain cannot represent constraints with constant offsets. For example, $x - 2y \leq -3$ can only be over-approximated as $x \leq 2y$. The Stripes [41] domain expresses linear inequalities of the form $x - a \cdot (y[+z]) \geq b$ with $a, b \in \mathbb{Z}$. It serves only as a subdomain for the symbolic representation of such inequalities and for propagating information back and forth between other subdomains. It is built in CLOUSOT [40] and serves a similar purpose, but with a different trade-off between precision and efficiency, and with different expectations of surrounding domains. For example, Stripes assumes that equalities are maintained by some other domain along it. This makes direct empirical comparison with Template DBM difficult since they are not easily implemented within the same system.

4.7 Conclusion

We introduce a new weakly numerical abstract domain, Template DBM, representing inequalities of the form $ax - by \leq c$, between pairs of variables x and y, where a and b come from a predefined coefficient template and c is an integer constant. This work shows how to use a DBM to represent domain elements. We provide algorithms for full saturation, incremental saturation, and lattice operations for the domain. In terms of precision and performance, Template DBM lies between the Zones and Polyhedra domains. Our experiments demonstrate that its runtime is comparable to Zones, while the number of assertion checks it solves for memory-safety verification is close to that of Polyhedra.

Chapter 5

Taint Analysis via Heap-Aware Propagation

Taint analysis for C depends on how precisely the analysis tracks tainted values as they flow through memory objects. Without field-sensitivity, a tainted object field can spuriously taint an entire object, producing significant false alarms. In this chapter, we present a field-sensitive taint analysis that combines Data Structure Analysis (DSA) for field-sensitive points-to information with a data-flow analysis for tracking explicit dependencies. We identify that existing DSA-style analyses lose precision on a common object pattern: a C struct object that mixes scalar fields with a flexible array member. When such an array is accessed with symbolic indices, DSA loses points-to field sensitivity for the entire object, inflating false alarms in the data-flow analysis.

To address this, we propose I-DSA, which extends DSA by representing field positions as intervals, allowing tracking of pointers with symbolic bounds without losing field sensitivity. We further develop a data-flow analysis based on abstract interpretation and integrate it with an existing value analysis to prune flows along paths that are provably infeasible. We implement I-DSA on top of SeaDSA and the data-flow analysis inside Crab. Our tool targets LLVM IR and supports configurable taint policies by instrumenting the IR with taint intrinsics. Experimental results demonstrate that I-DSA reduces false alarms on programs with the targeted memory patterns with no measurable performance overhead.

5.1 Introduction

Modern software systems, including network utilities, databases, and serialization libraries, continuously process external inputs and API requests. These interactions increase the risk of unintended information leaks [85] or malicious data breaches [83, 84]. Taint analysis [3, 58, 108] addresses this risk by checking whether untrusted or sensitive data can reach critical sinks, such as diagnostic logging, where private information may be exposed.

The main challenge of taint analysis is the high rate of false positives caused by over-approximation of taint flows, which leads to spurious warnings and unreliable leak detection. This imprecision arises from two main sources: *field sensitivity* and *semantic reachability*.

In C programs, data is often organized into complex memory objects such as structs and arrays. Data flow through such objects is rarely direct, so taint often affects a specific field rather than the entire object. Without field sensitivity, taint analysis may treat a whole object as tainted because it cannot distinguish between tainted and untainted fields within the same object. This over-tainting produces false alarms and unreliable leak detection. Existing works such as CompTaint [8] and P/Taint [51] show that precision depends heavily on the memory model on which the taint analysis is built. Integrating pointer analysis to preserve field sensitivity therefore improves precision.

For semantic reachability, it is crucial to determine whether tainted data can reach a vulnerable function under the current program conditions. Purely syntactic approaches often miss that a flow is semantically unreachable, which also leads to false positives. Semantics-based taint analysis, such as Mopsa-Nexp [88], uses abstract interpretation [25] to compute numerical invariants that prune unreachable paths. This refinement ensures that the analysis focuses on reachable vulnerabilities.

In this chapter, our taint analysis combines Data Structure Analysis [68, 54] (DSA) with data-flow analysis (DFA): DSA first infers points-to information, and DFA then uses it for memory-aware taint propagation. To improve precision, we introduce a new DSA variant with interval-based offsets, allowing taint analysis to preserve more field-level precision than when using standard DSA. We further build DFA on an existing abstract-interpretation-based value analysis, using numerical invariants to rule out infeasible taint flows. Together, these analyses make taint propagation more precise while preserving practical scalability.

We implement our taint analysis on top of the SeaHorn [53] verification framework, targeting LLVM IR [67]. Our new DSA extends the implementation of SeaDSA [54], and our DFA is implemented in Crab [55]. The analysis takes user-specified taint policies in YAML and automatically instruments LLVM IR with taint intrinsics. We evaluate it

on 183 benchmarks, including 151 real-world programs from IONC [2], CURL [32], and COREUTILS [43]. The results show that replacing standard SeaDSA with our new DSA reduces false warnings by 6.6%–7.9%, with comparable overall performance.

Our contributions are threefold: we formalize the semantics of explicit-flow taint analysis; we design a more precise pointer-analysis abstraction than existing DSA-style approaches; and we implement the resulting analysis for practical use.

The chapter is organized as follows. Section 5.2 provides an overview of our approach with a motivating example. Section 5.3 presents the concrete semantics of our intermediate language. Section 5.4 specifies our new DSA, and Section 5.5 defines the data-flow analysis. Section 5.6 describes the implementation, Section 5.7 presents the evaluation, and Section 5.8 discusses related work.

5.2 Overview

In this section, we motivate our approach with a simplified example adapted from a real-world codebase. Throughout, we illustrate how points-to analysis informs data-flow analysis for taint propagation. We also highlight precision loss in previous approaches to motivate our refinement.

Consider the example[1] in Fig. 5.1a. A `struct repl_block` object referenced by `b` is allocated at line 15. Its allocation size depends on the user-provided parameter `len` (line 11) because `struct repl_block` ends with a flexible array member `buf[]`. This array storage is obtained by allocating extra bytes beyond the static structure size. The code then initializes `b->buf` from user-controlled inputs under the bound `i < copy`. At the end of the function, the program calls `log` to report block information.

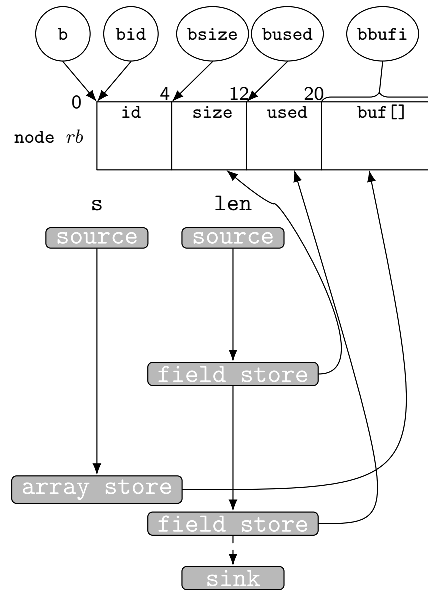
Clearly, the field `id` is not influenced by user input. At line 16, its value is derived from the global serial number `block_id`. Showing this fact requires two ingredients: (1) alias information proving that no other pointer may access `p->id`, and (2) data-flow information showing that user-controlled inputs are not propagated into this field. The first ingredient comes from pointer analysis, and the second from data dependency tracking. We therefore combine a points-to analysis (PTA) to identify possible memory targets with a taint data-flow analysis (DFA) to propagate taint across assignments and memory operations.

PTA computes alias information from memory usage, indicating which objects each pointer may reference and how pointers are related. Our PTA builds on Data Structure

[1] Source code: https://github.com/redis/redis/blob/8.6.2/src/replication.c#L476.

```c
typedef struct repl_block {
  int id;
  size_t size;
  size_t used;
  char buf[]; // flexible array member
} repl_block_t;

static int block_id = 0;

void feedReplicationBuffer(
          const char *s, size_t len) {
  repl_block_t *b;
  size_t copy;

  b = malloc(sizeof(repl_block_t) + len);
  b->id = block_id++;
  b->size = len;
  b->used = 0;

  size_t rem = b->size - b->used;
  copy = len < rem ? len : rem;
  for (size_t i = 0; i < copy; ++i) {
    b->buf[i] = s[i];
  }
  b->used += copy;

  log('block id=b->id, b->used, b->size);
}
```

(a)



(b)

Figure 5.1: (a) A program, and (b) its taint-flow graph over the data structure graph.

Analysis [68] (DSA), which represents aliasing relationships with a data-structure graph (DSG). DSG nodes represent abstract memory objects (for example, by grouping objects allocated at the same site), and DSG edges represent points-to relations. DSG preserves field sensitivity by using byte offsets within a node as edge source and destination positions, indicating which pointer can access which object field. In our example (top of Fig. 5.1b), a precise DSG contains a node *rb* for the object pointed to by `b`, with a distinct field `id` at offset 0. Other fields are represented at different offsets.

A DFA for taint computes how values from external sources (e.g., user or file inputs) propagate to sensitive operations that may expose data (e.g., logging or system calls). Such untrusted inputs are modeled as *sources*, and potential leakage points are modeled as *sinks*. The analysis tracks taint propagation under explicit policies for assignments, memory loads/stores, and function calls. In our example, function input parameters are sources and logging calls are sinks. The flow graph is shown in Fig. 5.1b. Assignments such

as lines 16 and 17 propagate taint along data dependencies from read locations to written locations. Concretely, line 16 sets `id` from `block_id`; together with DSA alias information, this implies that no user-controlled input reaches `id`. By contrast, fields such as `size`, `used`, and elements of `buf` may depend on tainted inputs. At the sink, output operations may therefore leak tainted information. Overall, this example shows how user-controlled inputs from line 11 can flow to critical output at line 27.

However, taint-analysis precision depends on preserving field sensitivity in the underlying PTA, and DSA-style PTA can lose field sensitivity on objects like the one in our example. The main challenge appears at the memory access in line 23, where path-dependent aliasing makes the pointer target ambiguous and admits multiple possible locations. In such cases, existing DSA variants [68, 54] conservatively merge offsets within a node into a single abstract location, which loses field sensitivity. For a taint client, taint flowing into fields such as `size` (or `buf`) can then spuriously taint `id`, producing false positives at sinks (e.g., line 27). In this example, however, no ambiguous access targets `id`, so preserving a separate abstract field for `id` is expected. Our first contribution is a refinement that reduces this loss of field sensitivity, as presented in Section 5.4.

Building on this refined memory abstraction, we present a DFA in Section 5.5 that annotates variables and memory locations with taint *tags*, where each tag is a lattice element identifying a tainted source (e.g., a particular user input). The DFA propagates taint tags along explicit data dependencies induced by assignments, memory loads/stores, and function calls. For precision, we combine the DFA with a value analysis (VA) that tracks numerical invariants, allowing us to rule out taint flows along paths that are provably infeasible.

5.3 Language and Semantics

This section describes the target language for analysis and its concrete semantics.

To facilitate analysis, we lower C programs to a simple intermediate representation (IR), called C-IR, shown in Fig. 5.2. Compound C statements are translated into sequences of primitive statements in C-IR. We assume the input language is a subset of C with integer variables $i \in \mathcal{V}_{\mathcal{I}}$ and pointer variables $p, q \in \mathcal{V}_{\mathcal{P}}$, where $x \in \mathcal{V} = \mathcal{V}_{\mathcal{I}} \cup \mathcal{V}_{\mathcal{P}}$. Pointers range only over heap-object addresses (no stack-address values). C-IR supports dynamic allocation, integer arithmetic, and pointer arithmetic with either fixed non-negative byte offsets ($o \in \mathbb{N}$) or indexed byte offsets ($i \times s$) under a constant stride $s \in \mathbb{N}^*$. Load/store statements model memory access, while I/O primitives identify taint sources and sinks.

$$
\begin{array}{llll}
\mathsf{P} \ni pg & ::= \mathsf{S}^* & & \text{(Program)} \\
\mathsf{S} \ni s & ::= p = \mathsf{alloc}(i) & & \text{(Allocation)} \\
& \quad p = q + o & \mid\ p = q + (i \times s) & \text{(Pointer Arithmetic (Offset \& Indexed))} \\
& \quad x = *p & \mid\ *p = x & \text{(Mem Load \& Store)} \\
& \quad s;\ s & \mid\ i = a & \text{(Sequencing \& Int Arithmetic)} \\
& \quad x = \mathsf{input}() & \mid\ \mathsf{output}(x) & \text{(Input \& Output)} \\
& \quad \mathsf{if}(b)\ \mathsf{then}\ s\ \mathsf{else}\ s & \mid\ \mathsf{while}(b)\ s & \text{(Control Flow)} \\
\mathsf{B} \ni b & ::= a\ \mathsf{op}_{cmp}\ a\ \mid\ \neg\, b & \mid\ b\ \mathsf{op}_{logic}\ b & \text{(Guard Expressions)} \\
& \quad p = q & \mid\ p \neq q & \text{(Pointer Comparisons)} \\
\mathsf{A} \ni a & ::= c \in \mathsf{Const}\ \mid\ i \in \mathcal{V}_{\mathcal{I}} & \mid\ a\ \mathsf{op}_{int}\ a & \text{(Int Expressions)}
\end{array}
$$

Figure 5.2: The syntax of C-IR.

C-IR is a presentation IR used to define semantics concisely, while the implementation analyzes LLVM IR directly. Accordingly, C-IR does not assume a fixed integer width (e.g., 16, 32, or 64 bits). Instead, integer widths are inherited from LLVM types and the target DataLayout; the same applies to byte offsets used in pointer arithmetic. For presentation simplicity, C-IR models memory accesses as non-overlapping field accesses, restricts pointer arithmetic to non-negative offsets, and omits function calls and deallocation. These are presentation-level assumptions only; all are supported in the implementation.

We model memory as a collection of objects, where each allocation creates a fresh, unique object. A pointer is a pair consisting of a base address and a numeric offset in bytes that identifies a field. A memory cell (i.e., a field of a memory object) stores either an integer or a pointer. Pointer arithmetic constructs new pointers through constant-offset addressing (e.g., `&b->id` as `bid = b + 0`) or indexed addressing (e.g., `&b->buf[i]` as `bbuf = b + 20; bbufi = bbuf + (i * 1)`). The IR therefore covers field and array-element accesses over heap objects.

Formally, a memory state $\sigma \in \Sigma$ is a pair of an *environment* map ρ and a *heap* h. The environment maps variables to values, $\mathcal{V} \mapsto \mathcal{X}$, and the heap maps addresses to values, $\mathcal{X}_{\mathcal{P}} \mapsto \mathcal{X}$. Values are pairs in $\mathcal{X} \overset{\text{def}}{=} \mathbb{N} \times \mathbb{Z}$, representing either integers or addresses. By convention, an integer n is encoded as $(0, n)$, and an address a is encoded as (b, o) with nonzero base b. For example, 3 is represented as $(0, 3)$, while $(10_{16}, 4)$ denotes offset 4 of the object at base address 10_{16}. For brevity, we write $a.b$ and $a.o$ for the base and offset of an address, use $\mathcal{X}_{\mathcal{P}}$ for the set of address values, and define $\mathcal{L} \overset{\text{def}}{=} \mathcal{V} \cup \mathcal{X}_{\mathcal{P}}$ as the set of memory locations.

We define the concrete small-step operational semantics of C-IR as a transition system

Next-state transformers:

$$\frac{n = \rho(i) \quad (b, h') = h.\mathsf{alloc}(n) \quad \rho' = \rho[p \mapsto (b, 0)] \quad T' = i \xrightarrow{T} \{\}}{[\![p = \mathsf{alloc}(i)]\!]((\rho, h, T)) = (\rho', h', T')}$$

$$\frac{n = \rho.\mathtt{eval}(\mathsf{A}) \quad \rho' = \rho[i \mapsto n] \quad T' = \mathtt{vs}(\mathsf{A}) \xrightarrow{T} i}{[\![i = \mathsf{A}]\!]((\rho, h, T)) = (\rho', h, T')}$$

$$\frac{(b, o_q) = \rho(q) \quad \rho' = \rho[p \mapsto (b, o_q + o)] \quad T' = q \xrightarrow{T} p}{[\![p = q + o]\!]((\rho, h, T)) = (\rho', h, T')}$$

$$\frac{(b, o_q) = \rho(q) \quad n = \rho(i) \quad o_p = o_q + n * s \quad \rho' = \rho[p \mapsto (b, o_p)] \quad T' = q \xrightarrow{T} p}{[\![p = q + (i \times s)]\!]((\rho, h, T)) = (\rho', h, T')}$$

$$\frac{val = \text{input}() \quad \rho' = \rho[x \mapsto val] \quad T' = T \cup \{x\}}{[\![x = \mathsf{input}()]\!]((\rho, h, T)) = (\rho', h, T')}$$

$$\frac{a_p = \rho(p) \quad val = h(a_p) \quad \rho' = \rho[x \mapsto val] \quad T' = a_p \xrightarrow{T} x}{[\![x = *p]\!]((\rho, h, T)) = (\rho', h, T')}$$

$$\frac{a_p = \rho(p) \quad val = \rho(x) \quad h' = h[a_p \mapsto val] \quad T' = x \xrightarrow{T} a_p}{[\![*p = x]\!]((\rho, h, T)) = (\rho, h', T')}$$

Set transformers:

$$[\![s]\!](X) = \{\sigma' \mid \sigma \in X, \sigma' = [\![s]\!](\sigma)\} \text{ for atomic } s \qquad [\![s_1;\ s_2]\!](X) = [\![s_2]\!]([\![s_1]\!](X))$$

$$[\![\mathsf{if}(b) \text{ then } s_t \text{ else } s_f]\!](X) = [\![s_t]\!](\mathsf{grd}_b(X)) \cup [\![s_f]\!](\mathsf{grd}_{\neg b}(X))$$

$$[\![\mathsf{while}(b)\ s]\!](X) = \mathsf{grd}_{\neg b}\Big(\mathsf{lfp}\ \Big(\lambda Y.\ X \cup [\![s]\!](\mathsf{grd}_b(Y))\Big)\Big)$$

Figure 5.3: Concrete collecting semantics (taint rules highlighted for Section 5.5).

over states. Specifically, we define a state transformer $[\![s]\!] : \Sigma \to \Sigma$, shown in Fig. 5.3. The initial state consists of empty maps. To simplify memory modeling, we assume that h provides an allocation operation alloc that always succeeds and returns a fresh base address. We also assume that ρ provides an evaluation function eval for arithmetic expressions. Invalid dereference and invalid load/store behavior are outside the scope of this semantics. The collecting semantics lifts the single-state transformer to sets of states, yielding a set transformer $[\![s]\!] : \mathbb{D} \to \mathbb{D}$ over $\mathbb{D} \stackrel{\text{def}}{=} \mathcal{P}(\Sigma)$ (Fig. 5.3). For guards, we define $\mathsf{grd}_b(X) \triangleq \{\sigma \in X \mid [\![b]\!](\sigma) = \mathsf{true}\}$. The semantics of while loops is given by the least fixpoint of the corresponding set transformer, following Section 2.2.

5.4 Data Structure Analysis with Interval Offsets

In this section, we introduce I-DSA, a new DSA that improves precision by abstracting cell offsets by intervals, rather than constants. We first define our graph representation, then identify the specific cases where the original DSA loses precision due to imprecise offset abstraction, and finally formalize the interval-based offset abstraction used by our analysis.

DSA [68, 54] summarizes the memory objects and points-to relations induced by pointer operations using a Data Structure Graph (DSG). A DSG contains a set of abstract *nodes* where each node summarizes one or more memory objects. Each node contains a set of abstract *cells* that represent distinguished fields of these objects. A field with a statically known location is identified by its numeric offset within the object. Edges between cells encode memory-to-memory points-to relations, while pointer variables to cells show variable points-to facts.

Formally, a DSG is a tuple $G \stackrel{\text{def}}{=} \langle N, E, E_{\mathcal{P}} \rangle$, where N is the set of nodes, E is the set of edges $e \stackrel{\text{def}}{=} c_1 \rightharpoonup c_2$, mapping from a source cell to a target cell, a cell $c \in C : N \times O^\sharp \stackrel{\text{def}}{=} (n, o^\sharp)$ represents fields at offset $o^\sharp$ within node n, and $E_{\mathcal{P}} : \mathcal{V}_{\mathcal{P}} \rightharpoonup C$ is a partial mapping from pointer variables to cells. For later use, we write $E[c_2/c_1]_{\text{key}}$ to replace the key c_1 with c_2, and $E_{\mathcal{P}}[c_2/c_1]_{\text{value}}$ (or $E[c_2/c_1]_{\text{value}}$) to replace every occurrences of c_1 with c_2 in the map values. If the key or value does not occur, the corresponding replacement is a no-op.

Let $C_n \subseteq C$ denote the cells that belong to a node n. A DSG is *well-formed* if it meets the following conditions:

C.1 (Disjointness) $\forall n \in N, c_1, c_2 \in C_n.\ c_1 \neq c_2 \Leftrightarrow \neg\texttt{overlap}(c_1.o^\sharp, c_2.o^\sharp)$;

C.2 (Collapsed) $\texttt{isCollapsed}(n) \Rightarrow |C_n| = 1$;

C.3 (Univalent) $\forall c \in \texttt{dom}(E) : |E(c)| = 1$.

Conditions C.1 and C.2 characterize whether a node is field sensitive (multiple disjoint cells) or *collapsed* (a single summary cell). Condition C.3 and the partial map $E_{\mathcal{P}}$ enforces the invariant from unification-based PTA [100] that each pointer/cell has at most one outgoing edge. As DSA represents the offset of each cell as a constant, `overlap` is defined by numeric comparison and `isCollapsed` is a special node property that is set to true when the field sensitivity is lost.

In our example in Fig. 5.1a, the DSG at the top of Fig. 5.1b includes a node *rb* that separates the constant-offset fields referred by pointers `b`, `bid`, `bsize`, and `bused`. The access

at line 23, however, introduces a symbolic offset through `i`. Existing DSA works represent this node rb by collapsing all cells within it. This is sound but at the expense of losing field sensitivity for the node rb.

I-DSA addresses this limitation by representing each cell offset as an interval. An array access with an uncertain index is modeled by an interval $[s, e]$ that over-approximates all potential offsets within the accessed range. For example, the access `p->buf[i]` at line 23 is represented by the interval $[12, +\infty]$, where the lower bound shows the buffer begins at offset 12 but upper bound is unknown since the array size is statically unknown. The interval abstraction preserves field sensitivity for the part of the object not affected by this access (in our example, these are the first 12 bytes of the object). When intervals of two accesses overlap, the affected cells are merged, whereas cells outside the overlap remain separate.

Formally, an interval offset $o^\sharp$ is written as $[s, e]$, where $o^\sharp.s$ is the start offset and $o^\sharp.e$ is the end offset. We define a domain of intervals following [23] for representing offsets. The domain $\mathcal{O}$ is a complete lattice $(O^\sharp, \sqsubseteq_\mathcal{O}, \sqcup_\mathcal{O}, \sqcap_\mathcal{O}, \bot_\mathcal{O}, \top_\mathcal{O})$ with partial order $\sqsubseteq_\mathcal{O}$, join $\sqcup_\mathcal{O}$, and meet $\sqcap_\mathcal{O}$ operations. The lattice includes bottom $\bot_\mathcal{O}$, and top $\top_\mathcal{O}$. The top element is $[0, +\infty]$, since offsets are non-negative. We assume predefined abstraction and concretization functions: $\alpha_\mathcal{O} : \mathcal{P}(\mathbb{N}) \mapsto O^\sharp$ and $\gamma_\mathcal{O} : O^\sharp \mapsto \mathcal{P}(\mathbb{N})$. We also use $+_\mathcal{O}$ and $-_\mathcal{O}$ over intervals for arithmetic operations. The $[a, b] +_\mathcal{O} [c, d] \stackrel{\text{def}}{=} [a + c, b + d]$, while $[a, b] -_\mathcal{O} [c, d] \stackrel{\text{def}}{=} [\max(0, a - d), \max(0, b - c)]$.

In I-DSA, to meet well-formed conditions of a graph, we let $\texttt{overlap}(o^\sharp_1, o^\sharp_2) \stackrel{\text{def}}{=} o^\sharp_1 \sqcap_\mathcal{O} o^\sharp_2 \neq \bot_\mathcal{O}$. A node is collapsed when it has only has one cell whose offset is maximum, denoted as $\texttt{isCollapsed}(n) \stackrel{\text{def}}{=} |C_n| = 1 \land \forall c \in C_n.\ c.o^\sharp = \top_\mathcal{O}$.

The concretization γ_{dsa} for DSA-style analyses is inherently complicated because it must relate an abstract graph to concrete program states. Although the idea is conceptually simple, formalization requires an auxiliary witness to show how memory objects are represented by the abstract graph. We therefore introduce a simulation relation μ between a program state σ and a DSG g. The μ is a graph of a total function $\mathcal{X}_\mathcal{P} \to C$ between a program state σ and a DSG g, mapping each memory location $a \in \sigma.h$ to exactly one abstract cell $c \in g.C$.

Consider a concrete memory object obj in $\sigma.h$. Let $A = \{a_0, a_1, \ldots, a_m\}$ be the field locations of obj, ordered by increasing offset, with $a_0.o = 0$. Suppose a node n in $g.N$ with cells $C_n = \{c_0, c_1, \ldots, c_p\}$ representing obj, ordered by the minimum value of their abstract offsets. The μ maps the first field at offset 0 to some cell c_i $(0 \leq i \leq p)$ so that $(a_0, c_i) \in \mu$. For every other field locations $a_j \in A$, μ maps a_j to a cell c_k with $i \leq k \leq p$

whose relative abstract offset covers the relative concrete offset from a_0:

$$\forall a_j \in A.\, \exists c_k \in C_n.\, (a_j, c_k) \in \mu \iff a_j.o \in \gamma_{\mathcal{O}}(c_k.o^\sharp -_{\mathcal{O}} c_i.o^\sharp)$$

Intuitively, (a_0, c_i) serves as an anchor for relative offsets, allowing μ to derive the remaining field-to-cell mappings. The remaining field locations are then mapped to cells whose abstract offsets cover their concrete offsets relative to this base.

Consider the code example in Fig. 5.1a and its DSG (Fig. 5.1b). Suppose the object at line 15 is allocated on the heap at address 100_{16} with a total size of 24 bytes, so the flexible array contains exactly four one-byte elements. The fields are located at 100_{16} for `id`, 104_{16} for `size`, 112_{16} for `used`, and $120_{16} + i$ for `buf[i]` where $0 \leq i \leq 3$. In the DSG, the node rb contains cells with offsets $[0, 0]$, $[4, 4]$, $[8, 8]$, and $[12, +\infty]$ for `id`, `size`, `used`, and `buf[]`, respectively. The simulation relation μ maps these physical addresses to their corresponding cells:

$$(100_{16}, c_{id}) \in \mu \qquad (104_{16}, c_{size}) \in \mu$$
$$(112_{16}, c_{used}) \in \mu \qquad \forall i \in [0, 3].\ (120_{16} + i, c_{buf[]}) \in \mu$$

Using μ, we define when a graph g simulates a program state σ, written $\sigma \models_\mu g$. The relation requires all points-to relations in σ to be represented by the corresponding edges in g. That is, $\models_\mu$ is defined as:

$$\sigma \models_\mu g \Leftrightarrow \begin{cases} \forall p \in \mathcal{V}_{\mathcal{P}}.\ (\rho(p), E_{\mathcal{P}}(p)) \in \mu \\ \forall (a, a') \in h.\ \exists c \in C.\ (a, c) \in \mu \wedge (a' \in \mathcal{X}_{\mathcal{P}} \Rightarrow (a', E(c)) \in \mu) \end{cases}$$

The first case states that each variable-to-memory points-to relation in the concrete state is represented by a points-to relation from it to a cell. The second condition requires every concrete memory-to-memory points-to relation to be represented by an abstract edge between the corresponding cells.

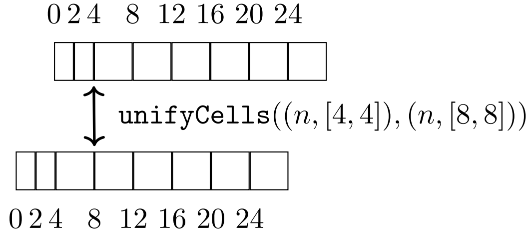
The concretization function $\gamma_{dsa} : G \mapsto \mathcal{P}(\Sigma)$ maps a DSG to the set of all concrete states with respect to the simulation relation $\models_\mu$:

$$\gamma_{dsa}(g) \stackrel{\text{def}}{=} \{\sigma \in \Sigma \mid \exists \mu.\ \sigma \models_\mu g\}$$

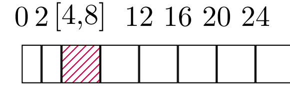
As DSA incrementally constructs the graph, a pointer/cell may temporarily have multiple outgoing edges. For example, a pointer p may point to both $(n_1, o_1^\sharp)$ and $(n_2, o_2^\sharp)$. This violates the invariant C.3, which requires every pointer/cell to have only one outgoing edge.

```c
struct in_addr { uint32_t s_addr; };
struct in6_addr { uint32_t s6_addr[4]; };
struct ifaddrs{struct sockaddr *ifa_addr;};

struct sockaddr_in {
  uint16_t sin_family, sin_port;
  struct in_addr sin_addr;
  uint8_t sin_zero[8];
};
struct sockaddr_in6 {
  uint16_t sin6_family, sin6_port;
  uint32_t sin6_flowinfo;
  struct in6_addr sin6_addr;
  uint32_t sin6_scope_id;
};
int if2ip(int af, char *interf, char *buf) {
  struct ifaddrs *iface;
  ...
  if (getifaddrs(&iface) >= 0) {
    void *addr;
    if (af == AF_INET6)
      addr = &((struct sockaddr_in6 *)
               iface->ifa_addr)->sin6_addr;
    else
      addr = &((struct sockaddr_in *)
               iface->ifa_addr)->sin_addr;
    ...
  }
}
```

(a)



(b)



(c)

Figure 5.4: Effect of `unifyCells` on the same node: (a) before unify; (b) after unify.

DSA resolves this by *unification*, which merges the two target cells into a new cell $(n_3, o_3^\sharp)$ and redirects p to point to it so that it has a single target. We capture this step with the rule $g \vdash \texttt{unifyCells}(c_1, c_2) \Downarrow g_1$, where g violates C.3, g_1 restores it by unifying two cells c_1 and c_2 after all recursive unifications required by their outgoing edges.

Unfortunately, unification does not always preserve field sensitivity. In DSA, cells carry constant offsets. When $n_1 \neq n_2$, unification merges the two nodes directly. To match the offsets for unification, the analysis computes the relative offset (e.g., unifying $(n_1, 0)$ with $(n_2, 4)$ use $\delta = 4$) and shifts one node (e.g., shifting n_1 by 4 bytes). When $n_1 = n_2$, this adjustment is impossible because a node cannot be shifted relative to itself; thus, if $\delta > 0$, DSA collapses the node and loses field sensitivity. I-DSA avoids this by using interval offsets: unifying two same-node cells creates a merged cell with offset $o_1^\sharp \sqcup_{\mathcal{O}} o_2^\sharp$ (e.g., $[0,0] \sqcup_{\mathcal{O}} [4,4] = [0,4]$), merges only cells overlapping that interval, and leaves the others unchanged. The rest of this subsection presents these two unification cases.

The unification rules for I-DSA are given in Fig. 5.5. Without loss of generality, we

$$\texttt{unifyCellsSame}\ \frac{\begin{array}{cc} c_1.n = c_2.n & o^\sharp = c_1.o^\sharp \sqcup_{\mathcal{O}} c_2.o^\sharp \\ S = \{c \in C_{c_1.n} \mid c.o^\sharp \sqcap_{\mathcal{O}} o^\sharp \neq \bot_{\mathcal{O}}\} & g \vdash \texttt{mergeCells}(S, o^\sharp) \Downarrow c_m, g_1 \end{array}}{g \vdash \texttt{unifyCells}(c_1, c_2) \Downarrow g_1}$$

$$\texttt{unifyCellsOther}\ \frac{c_1.n \neq c_2.n \quad \delta = c_2.o^\sharp.s - c_1.o^\sharp.s \quad g \vdash \texttt{unifyNodes}(c_1.n, c_2.n, \delta) \Downarrow g_1}{g \vdash \texttt{unifyCells}(c_1, c_2) \Downarrow g_1}$$

$$\texttt{unifyNodes}\ \frac{\{c_1, \ldots, c_k\} = C_{n_1} \quad g_1 = g \quad \forall i \in [1, k]. \begin{array}{l} o_i^\sharp = c_i.o^\sharp +_{\mathcal{O}} [\delta, \delta] \\ S = \{c \in C_{n_2} \mid c.o^\sharp \sqcap_{\mathcal{O}} o_i^\sharp \neq \bot_{\mathcal{O}}\} \\ g_i \vdash \texttt{mergeOrNew}(S, n_2, o_i^\sharp) \Downarrow c_m, g_i' \\ g_i' \vdash \texttt{redirectEdges}(c_i, c_m) \Downarrow g_{i+1} \end{array}}{g \vdash \texttt{unifyNodes}(n_1, n_2, \delta) \Downarrow g_{k+1}}$$

$$\texttt{mergeOrNew-}\textsc{New}\ \frac{S = \emptyset \quad c_m = (n, o^\sharp)}{g \vdash \texttt{mergeOrNew}(S, n, o^\sharp) \Downarrow c_m, g}$$

$$\texttt{mergeOrNew-}\textsc{Merge}\ \frac{S \neq \emptyset \quad g \vdash \texttt{mergeCells}(S, o^\sharp) \Downarrow c_m, g_1}{g \vdash \texttt{mergeOrNew}(S, n, o^\sharp) \Downarrow c_m, g_1}$$

Figure 5.5: Unification rules (Part I).

assume $o_1^\sharp.s \leq o_2^\sharp.s$; otherwise, the arguments are swapped before calling `unifyCells`. The operation unifies n_1 into n_2, redirects incoming edges to n_2, and recursively unifies outgoing edges.

For same-node unification (`unifyCellsSame`), I-DSA computes the joined interval $o^\sharp$, merges all and only cells (in a set S) overlapping $o^\sharp$ into one (denoted as c_m) using `mergeCells` (shown in Fig. 5.6), and leaves the rest of cells on the given node unchanged. During `mergeCells`, it is necessary to redirect edges for each cell in S to c_m following `redirectEdges` (explained later). During this process, if we compute $o^\sharp = \top_{\mathcal{O}}$, this indicates that the current node is collapsed.

We illustrate `unifyCells` on a single DSA node using the example[2] in Fig. 5.4a. At lines 22 and 25, `addr` accesses the raw IP address according to the protocol indicated by `af`. Since IPv4 and IPv6 store the address in different sub-structures, the assignment to `addr` requires different offset. We show the DSA node n representing the object referenced by `iface->ifa_addr` in Fig. 5.4b. n includes two distinct cells for the raw IP addresses $\{(n, [4, 4]), (n, [8, 8])\}$. Since `addr` cannot point to two cells at once by

[2]Original Code: https://github.com/curl/curl/blob/curl-8_19_0/lib/if2ip.c#L94.

$$\texttt{mergeCells}\ \frac{S = [c_1, \ldots, c_k] \quad o^\sharp_m = (\sqcup_{c \in S} c.o^\sharp) \sqcup_{\mathcal{O}} o^\sharp \quad c_m = (c_1.n, o^\sharp_m) \quad g_1 = g \quad \forall i \in [1,k].\ g_i \vdash \texttt{redirectEdges}(c_i, c_m) \Downarrow g_{i+1}}{g \vdash \texttt{mergeCells}(S, o^\sharp) \Downarrow c_m, g_{k+1}}$$

$$g[c_2/c_1]_{\text{value}} \stackrel{\text{def}}{=} \langle g.N,\ g.E[c_2/c_1]_{\text{value}},\ g.E_{\mathcal{P}}[c_2/c_1]_{\text{value}} \rangle$$

$$\texttt{redirectEdges-}\textsc{Rec}\ \frac{c_1, c_2 \in \texttt{dom}(E) \quad g_1 = g[c_2/c_1]_{\text{value}} \quad g_1 \vdash \texttt{unifyCells}(E(c_1), E(c_2)) \Downarrow g_2 \quad g_2.E = g_2.E[c_2/c_1]_{\text{key}}}{g \vdash \texttt{redirectEdges}(c_1, c_2) \Downarrow g_2}$$

$$\texttt{redirectEdges-}\textsc{Easy}\ \frac{c_1 \notin \texttt{dom}(E) \vee c_2 \notin \texttt{dom}(E) \quad g_1 = g[c_2/c_1]_{\text{value}} \quad g_1.E = g_1.E[c_2/c_1]_{\text{key}}}{g \vdash \texttt{redirectEdges}(c_1, c_2) \Downarrow g_1}$$

$$\texttt{find-1}\ \frac{k \in \texttt{dom}(m) \quad c = m[k]}{g \vdash \texttt{find}(m,k) \Downarrow c, g} \qquad \texttt{find-0}\ \frac{k \notin \texttt{dom}(m) \quad n \notin g.N \quad c = (n, [0,0]) \quad g_1 = g[N \mapsto g.N \cup \{n\}, m \mapsto m \cup \{k \mapsto c\}]}{g \vdash \texttt{find}(m,k) \Downarrow c, g_1}$$

Figure 5.6: Unification rules (Part II).

C.3, `unifyCells`$((n,[4,4]),(n,[8,8]))$ is invoked to merge cells into $(n,[4,8])$, as shown in Fig. 5.4c.

For unification across distinct nodes (`unifyCellsOther`), the two nodes are first aligned and then merged by `unifyNodes`. The alignment uses a relative offset δ, computed as the difference between the minimum (or starting) offsets of the two cells, to shift n_1 for unification compatibility.

In `unifyNodes`, each cell c_i of n_1 is shifted by δ (denoted as $o^\sharp_i$). To ensure unified results maintain cell disjointness, we collect cells S in n_2 that overlap $o^\sharp_i$, and then follow the rule `mergeOrNew`, to either merge cells in S (`mergeOrNew`-Merge) or create a new one (`mergeOrNew`-New). If $S = \emptyset$, we create a new cell c_m in n_2, and redirect all edges of c_i to c_m using `redirectEdges` in Fig. 5.6. If $S \neq \emptyset$, we merge all cells in S into a single cell c_m by `mergeCells`. In the end, when the merged cell c_m is obtained, we finally redirect edges from c_i to c_m.

The `redirectEdges` serves to move all edges from c_1 to c_2 and has two cases. If the given c_1 and c_2 both have outgoing edges, we need to recursively unify their points-to cells to maintain the invariant C.3. Once the recursive unification is done, we then update the edge set key to remove c_1. Otherwise, we simply replace c_1 with c_2 on the graph (`redirectEdges`-Easy).

We provide an example of unification across distinct nodes in Fig. 5.7a, where $c_1 =$

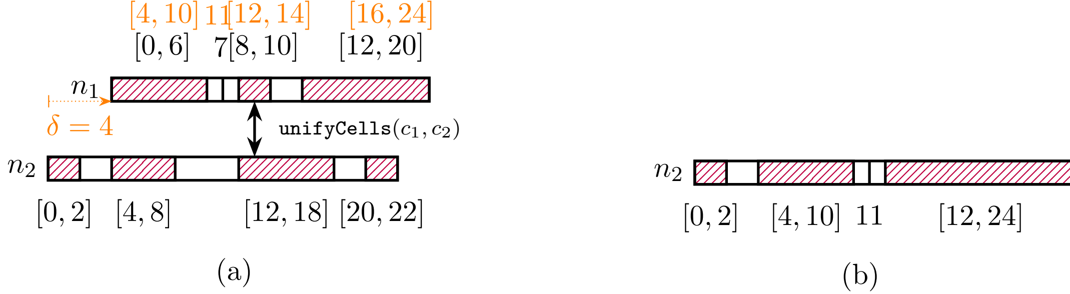


Figure 5.7: Effect of `unifyCells` on different nodes: (a) before unify; (b) after unify.

$(n_1, [8, 10])$ and $c_2 = (n_2, [12, 18])$. When `unifyCells`(c_1, c_2) is called, rule `unifyCellsOther` computes a relative offset $\delta = \min(c_2.o^\sharp) - \min(c_1.o^\sharp) = 4$ and shifts n_1 by δ in `unifyNodes`. Each shifted cell of n_1 is then unified with overlapping cells in n_2. If no overlap exists, such as $(n_1, [11, 11])$, a new cell covering the range is created. If one overlaps, such as merging $(n_1, [4, 10])$ and $(n_2, [4, 8])$, a new cell $(n_2, [4, 10])$ is created to cover both ranges. If multiple overlaps exist, like $(n_1, [16, 24])$ overlapping $(n_2, [12, 18])$ and $(n_2, [20, 22])$, we apply `mergeCells` to compute a cell $(n_2, [12, 24])$. As always, we apply `redirectEdges` when cells are merged. Finally, any cells remaining on n_2 with no overlap, such as $(n_2, [0, 2])$, are preserved. The resulting n_2 is shown in Fig. 5.7b.

Our transfer functions $[\![s]\!]^\sharp_D : G \to G$ for I-DSA are shown in Fig. 5.8. The analysis builds a DSG by evaluating pointer-manipulating statements. As a flow-insensitive analysis, when a statement queries a points-to relation, it either reuses an existing one (`find`-1) or creates a new one (`find`-0).

Specifically, for allocation, the analysis creates a fresh node n_f for the allocated object and adds a points-to edge $p \mapsto (n_f, 0)$, using unification if needed. For struct-field address computation, it maps the derived pointer p to a cell at the cumulative offset $o^\sharp_q +_{\mathcal{O}} [o, o]$, given by c_q referenced by q. For array pointer arithmetic with symbolic index i, the analysis approximates the possible index range and creates a cell covering all reachable array elements. Assuming the program has no undefined behavior, array indices are non-negative, so we approximate the range of i as $[0, +\infty]$. Note that this range can be refined using precomputed value information, e.g., from value-set analysis, or when the array size is statically known. For a memory load, if the loaded value assigned to x is a pointer, the analysis looks up the cell for x and unifies it with the cell reached by dereferencing p. A store is handled similarly.

Following the code snippet in Fig. 5.1a, I-DSA eventually constructs the graph shown at the top of Fig. 5.1b with cells: $c_{id} = (rb, [0, 0])$, $c_{size} = (rb, [4, 4])$, $c_{used} = (rb, [12, 12])$,

$$\frac{g.N = g.N \cup \{n_f\} \qquad \begin{array}{c} n_f \notin g.N \\ g \vdash \texttt{find}(E_{\mathcal{P}}, p) \Downarrow c_p, g_1 \end{array} \qquad g_1 \vdash \texttt{unifyCells}(c_p, (n_f, [0,0]))) \Downarrow g_2}{[\![p = \mathsf{alloc}(i)]\!]^{\sharp}_{D}(g) = g_2}$$

$$\frac{\begin{array}{c} g \vdash \texttt{find}(E_{\mathcal{P}}, q) \Downarrow (n, o^{\sharp}_{q}), g_1 \qquad g_1 \vdash \texttt{find}(E_{\mathcal{P}}, p) \Downarrow c_p, g_2 \\ c'_p = (n, o^{\sharp}_{q} +_{\mathcal{O}} [o, o]) \qquad g_2 \vdash \texttt{unifyCells}(c_p, c'_p) \Downarrow g_3 \end{array}}{[\![p = q + o]\!]^{\sharp}_{D}(g) = g_3}$$

$$\frac{\begin{array}{c} g \vdash \texttt{find}(E_{\mathcal{P}}, q) \Downarrow (n, o^{\sharp}_{q}), g_1 \qquad g_1 \vdash \texttt{find}(E_{\mathcal{P}}, p) \Downarrow c_p, g_2 \\ c'_p = (n, o^{\sharp}_{q} +_{\mathcal{O}} [0, +\infty]) \qquad g_2 \vdash \texttt{unifyCells}(c_p, c'_p) \Downarrow g_3 \end{array}}{[\![p = q + (i \times s)]\!]^{\sharp}_{D}(g) = g_3}$$

$$\frac{\begin{array}{c} x \in \mathcal{V}_{\mathcal{P}} \qquad g \vdash \texttt{find}(E_{\mathcal{P}}, p) \Downarrow c_p, g_1 \\ g_1 \vdash \texttt{find}(E, c_p) \Downarrow c_d, g_2 \qquad g_2 \vdash \texttt{find}(E_{\mathcal{P}}, x) \Downarrow c_x, g_3 \qquad g_3 \vdash \texttt{unifyCells}(c_d, c_x) \Downarrow g_4 \end{array}}{[\![x = *p \ / * p = x]\!]^{\sharp}_{D}(g) = g_4}$$

Figure 5.8: The transfer functions of I-DSA.

and $c_{buf[]} = (rb, [20, +\infty])$. At line 15, the analysis creates the node rb and maps `b` to $(rb, [0,0])$. At lines 16 and 17, it establishes points-to relations for the intermediate pointers `bid`, `bsize`, and `bused` to cells c_{id}, c_{size}, and c_{used}, respectively. The array access at line 23 then constructs $c_{buf[]}$ with the range $[20, +\infty]$, referenced by `pbufi`.

Theorem 5.4.1 (Correctness of $[\![s]\!]^{\sharp}_{D}$)**.**

$$\forall g \in G \ . \ [\![s]\!](\gamma_{dsa}(g)) \subseteq \gamma_{dsa}([\![s]\!]^{\sharp}_{D}(g))$$

Proof. Let $\sigma' := [\![s]\!](\gamma_{dsa}(g))$. By the definition of γ_{dsa}, $\exists \sigma \in \gamma_{dsa}(g)$ produces σ' and $\exists \mu.\ \sigma \models_{\mu} g$. We then show $\exists \mu'.\ g'$ such that $g' := [\![s]\!]^{\sharp}_{D}(g)$ and $\sigma' \models_{\mu'} g'$. We show how to construct μ' by cases.

For allocation, σ creates a fresh address $a = (b, 0)$ that p refers to. In g, we construct a new node n_f and make an edge $p \mapsto (n_f, 0)$, unifying with any existing target of p. Assume the result c. We have $\mu' := \mu[a \mapsto c]$.

For $p = q + \theta$, the concrete semantics computes the address of p by shifting the base address (b, o_q) by θ, where $\theta = o$ for struct field or $\theta = \rho[i]{+}s$ for array indexing. Abstractly,

we derive the target cell based on c_q and the abstract offset through abstraction $\alpha_{\mathcal{O}}(\{\theta\})$. When $\theta = o$, $\alpha_{\mathcal{O}}(\{\theta\}) = [o, o]$. When $\theta = \rho[i] + s$, we approximate this offset by $[0, +\infty]$, which is the top element. As the predefined $\alpha_{\mathcal{O}}$ is sound, the offset value for c'_p is a sound approximation. We apply unification if needed. The target c is what p points to. Then, $\mu' := \mu[(b, o_q) \mapsto c_q, (b, o_q + \theta) \mapsto c]$.

For a load or store, since three points-to relations are involved: p, the dereferenced location of p (denoted as d), and x, we let a_p, a_d and a_x be the concrete addresses, with $a_d = h(\rho(p))$. In abstract semantics, let c_p, c_d and c_x be their abstract counterparts. Since $a_x = a_d$, we unify c_d and c_x for sound approximation. Suppose c be the result. Then $\mu' := \mu[a_p \mapsto c_p, a_d \mapsto c, a_x \mapsto c]$.

By construction, we have $\sigma' \models_{\mu'} g'$. Thus, Theorem 5.4.1 holds. □

In summary, we have shown how the analysis constructs the points-to graph for a single function. While I-DSA uses interval to abstract offsets, the approach can be instantiated with other non-relational domains, such as constant sets or reduced products of intervals and congruences. DSA also propagates graph information between callers and callees to support context sensitivity; this mechanism is orthogonal to our extension, and I-DSA retains that capability following [68]. Next, we show how DFA uses computed graphs to track taint propagation.

5.5 Taint Semantics

This section presents a data-flow analysis (DFA) built on the I-DSA memory abstraction to track propagation through explicit data dependencies. We first define the rules governing taint flow across variables and memory locations, and then formalize the analysis as a reduced product of a memory domain and a taint domain. This lets the analysis leverage inferred memory facts, such as constant values or unreachable states, to improve precision.

We extend the concrete state $\sigma = (\rho, h)$ to a tainted state $\sigma_t \in \Sigma_t = (\rho, h, T)$, where $T \in \mathcal{P}(\mathcal{L})$ records tainted locations and $\mathcal{L} \stackrel{\text{def}}{=} \mathcal{V} \cup \mathcal{X}_{\mathcal{P}}$ includes program variables and memory addresses. This set representation is equivalent to a map $\mathcal{L} \rightarrow \mathcal{T}$, where $\mathcal{T} = \{\mathsf{clean}, \mathsf{tainted}\}$ is ordered by $\mathsf{clean} \sqsubseteq_{\mathcal{T}} \mathsf{tainted}$ with $\mathsf{clean} \sqcup_{\mathcal{T}} \mathsf{tainted} = \mathsf{tainted}$. Since $\mathcal{T}$ has only two elements, we store $l \in T$ when l is tainted. More generally, $\mathcal{T}$ can be any finite lattice to support richer taint properties.

To track taint flows through data dependencies, we define a propagation rule that updates the set of tainted locations. Given source locations S and target locations D, the

rule describes a taint flow from S to D, marking locations in D as tainted if and only if some source in S is tainted. Formally,

$$T' := S \xrightarrow{T} D \stackrel{\text{def}}{=} T \setminus D \ \cup \ \{l \in D \mid S \ \cap \ T \neq \emptyset\}$$

Fig. 5.3 highlights the propagation rules defined in the transfer function $[\![s]\!]_T : \Sigma_t \to \Sigma_t$. When a taint source is introduced through `input`, we add the tainted variable directly to T. For assignment statements, the propagation rule follows data dependencies between variables. For example, for `x = input(); y = x + 1; z = 3`, the rule adds `x` to the taint set via $T \cup \{x\}$. Subsequently, $x \xrightarrow{T} y$[3] adds y since $x \in T$, while the rule for z does not add z, following $\{\} \xrightarrow{T} z$.

For allocation, our analysis does not introduce taint, even if the allocation size is tainted. More conservative analyses may propagate taint from allocation sizes to the resulting pointer, $i \xrightarrow{T} p$ (e.g., [1]), or to the entire allocated object, $i \xrightarrow{T} \{p, (b, 0), \ldots, (b, n)\}$ (e.g., [37]). For pointer arithmetic, we propagate taint only between pointers and ignore the computed offset. For memory load and store operations, taint propagates between the variable and the memory location of the stored content. For instance, given `p = malloc(4); x = input(); *p = x`, the analysis propagates taint from `x` to the memory location pointed to by `p`. Overall, we write $[\![s]\!]_T : \mathbb{D}_t \to \mathbb{D}_t$ for the collecting taint semantics induced by the statement-level transfer function, where $\mathbb{D}_t \stackrel{\text{def}}{=} \mathcal{P}(\Sigma_t)$.

Our DFA is combined with a memory-aware numeric analysis that tracks numerical invariants and data dependencies. Let Num be a numerical domain, instantiated, for example, as Intervals [23], Octagons [78], or Polyhedra [30]. Formally, we define the abstract domain $\mathbb{D}_t^\sharp \stackrel{\text{def}}{=} \mathbb{D}_m^\sharp \times \mathbb{D}_s^\sharp$ as a reduced product of a memory domain $\mathbb{D}_m^\sharp$ and a taint domain $\mathbb{D}_s^\sharp$, with a reduction operator $\rho \stackrel{\text{def}}{=} \mathbb{D}_t^\sharp \to \mathbb{D}_t^\sharp$ for normalization. The memory domain $\mathbb{D}_m^\sharp$ uses Num to capture relational or non-relational properties over abstract memory locations $\mathcal{L}^\sharp$. The taint domain is defined as $\mathbb{D}_s^\sharp \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{L}^\sharp)$, where $l^\sharp \in \mathcal{L}^\sharp$ means that location may be tainted, rather than mapping locations to elements of the lattice $\mathcal{T}$.

Analysis precision is determined by the granularity of abstract locations $\mathcal{L}^\sharp$, which form the dimensions of our product domain. Each dimension carries two abstract values: a numerical bound and a taint element, all under the same memory abstraction. We instantiate this abstraction with I-DSA, representing memory locations as variables and memory cells. After I-DSA constructs DSGs G, we define $\mathcal{L}^\sharp \stackrel{\text{def}}{=} \mathcal{V} \cup \bigcup_{g \in G} g.C$. Following the summarization approach in [14], $\mathbb{D}_m^\sharp$ treats memory objects as summary objects (DSA

[3] $x \xrightarrow{T} y$ is a shorthand for $\{x\} \xrightarrow{T} \{y\}$.

$$\frac{\sigma_1^\sharp = [\![s]\!]_m^\sharp(\sigma^\sharp) \qquad c = g.E_{\mathcal{P}}[p] \qquad T_1^\sharp = \{c\} \xrightarrow{T^\sharp} \{x\}}{[\![x = *p]\!]_T^\sharp((\sigma^\sharp, T^\sharp)) = (\sigma_1^\sharp, T_1^\sharp)}$$

$$\frac{\sigma_1^\sharp = [\![s]\!]_m^\sharp(\sigma^\sharp) \qquad c = g.E_{\mathcal{P}}[p] \qquad T_1^\sharp = \{x\} \xrightarrow{T^\sharp} \{c\}}{[\![*p = x]\!]_T^\sharp((\sigma^\sharp, T^\sharp)) = (\sigma_1^\sharp, T_1^\sharp)}$$

Figure 5.9: Abstract transfer functions $[\![s]\!]_T^\sharp$ for taint analysis.

nodes) and models reads and writes with specialized operations. We omit memory-domain design and implementation details because they are orthogonal to taint abstraction, and assume $\mathbb{D}_m^\sharp$ provides $\bot_m$ (bottom), $\sqsubseteq_m$ (partial order), and $\sqcup_m$ (join).

The abstract state $\sigma_t^\sharp \in \mathbb{D}_t^\sharp$ consists of a memory state $\sigma^\sharp \in \mathbb{D}_m^\sharp$ and a taint state $T^\sharp \in \mathbb{D}_s^\sharp$. The taint state keeps a set of abstract locations (variables or memory cells) that are tainted, and the propagation function is defined as:

$$T_1^\sharp := S^\sharp \xrightarrow{T^\sharp} D^\sharp \stackrel{\text{def}}{=} T^\sharp \setminus \texttt{kill}^\sharp(D^\sharp) \cup \{\, l^\sharp \in D^\sharp \mid S^\sharp \cap T^\sharp \neq \emptyset \,\}$$

to soundly abstract the concrete rule. $\texttt{kill}^\sharp$ filters a location set and keeps only locations whose taint can be safely cleared. We allow killing taint on variables, but not on memory cells. Since each cell summarizes multiple object fields, we conservatively keep cells tainted. Formally, we define $\texttt{kill}^\sharp(D^\sharp) = D^\sharp \cap \mathcal{V}$.

The abstract transfer function $[\![s]\!]_T^\sharp : \mathbb{D}_t^\sharp \to \mathbb{D}_t^\sharp$ combines the transfer functions of its underlying subdomains. We assume $\mathbb{D}_m^\sharp$ provides a sound transfer function $[\![s]\!]_m^\sharp : \mathbb{D}_m^\sharp \to \mathbb{D}_m^\sharp$ for reasoning about numerical properties of abstract locations $\mathcal{L}^\sharp$. Initially, the abstract state maps all locations to clean ($T^\sharp = \emptyset$) and all numerical elements to $\top$. Most abstract taint rules use the same source and target location sets as the concrete semantics, including integer/pointer arithmetic and external input, so we only show load and store in Fig. 5.9. For these memory operations, the analysis follows the precomputed DSA graph g to identify the target cell c and propagates taint between that cell and the variable to soundly approximate the data flow of the memory access.

Following the reduction ρ, our taint analysis performs a semantic reduction between the memory and taint domains to improve precision. For instance, if an integer variable is tainted but its numerical bounds indicate it evaluates to a constant, we safely remove it from the taint set. We also prune taint facts from unreachable program states whenever

$\sigma^\sharp = \bot_m$. These refinements are formally applied through the reduction operator $\rho : \mathbb{D}_t^\sharp \to \mathbb{D}_t^\sharp$, defined as follows:

$$\rho((\sigma^\sharp, T^\sharp)) = \begin{cases} (\sigma^\sharp, \emptyset) & \text{if } \sigma^\sharp = \bot_m \\ (\sigma^\sharp, T^\sharp \setminus \{x\}) & \text{if } \sigma^\sharp.\texttt{isConstant}(x) \\ (\sigma^\sharp, T^\sharp) & \text{otherwise} \end{cases}$$

where $\sigma^\sharp.\texttt{isConstant}(x)$ is defined to look up the numerical property of x and return true if its abstract interval is a singleton, and false otherwise.

We assume the memory domain has a predefined concretization function $\gamma_m \stackrel{\text{def}}{=} \mathbb{D}_m^\sharp \to \mathbb{D}$. We define the concretization function for the taint domain as $\gamma_s : \mathbb{D}_s^\sharp \to \mathcal{P}(\mathcal{P}(\mathcal{L}))$, which maps a set of tainted abstract locations to a set of tainted concrete locations. Specifically, if $T^\sharp \in \mathbb{D}_s^\sharp$,

$$\gamma_s(T^\sharp) = \{T \mid T \subseteq ((T^\sharp \cap \mathcal{V}) \cup \bigcup_{c \in (T^\sharp \setminus \mathcal{V})} \gamma_{dsa}|_C(c))\}$$

where $\gamma_{dsa}|_C : C \to \mathcal{P}(\mathcal{X}_\mathcal{P})$ returns γ_{dsa} projected on cells. Following γ_m and γ_s, the concretization function for the taint analysis is defined compositionally as $\gamma_t : \mathbb{D}_t^\sharp \to \mathbb{D}_t$. That is, for an abstract state $\sigma_t^\sharp = (\sigma^\sharp, T^\sharp)$,

$$\gamma_t(\sigma_t^\sharp) \stackrel{\text{def}}{=} \{(\rho, h, T) \mid (\rho, h) \in \gamma_m(\sigma^\sharp) \wedge T \in \gamma_s(T^\sharp)\}.$$

Theorem 5.5.1 (Correctness of $[\![s]\!]_T^\sharp$)**.**

$$\forall \sigma_t^\sharp \in \mathbb{D}_t^\sharp, [\![s]\!]_T(\gamma_t(\sigma_t^\sharp)) \subseteq \gamma_t([\![s]\!]_T^\sharp(\sigma_t^\sharp))$$

Proof. Let $\sigma_t^\sharp = (\sigma^\sharp, T^\sharp)$ and take $\sigma_t' \in [\![s]\!]_T(\gamma_t(\sigma_t^\sharp))$. Then there exists $\sigma_t = (\rho, h, T) \in \gamma_t(\sigma_t^\sharp)$ such that $\sigma_t' = [\![s]\!]_T(\sigma_t)$.

By definition of γ_t, we have $(\rho, h) \in \gamma_m(\sigma^\sharp)$ and $T \in \gamma_s(T^\sharp)$. As the predefined γ_m is sound, the memory part of σ_t' belongs to $\gamma_m([\![s]\!]_m^\sharp(\sigma^\sharp))$. We only show γ_t is sound. For each transfer function, the taint component of $\sigma_t.T$ follows the statement-specific propagation rule to compute $\sigma_t'.T$. In short, $\sigma_t'.T = S \xrightarrow{T} D$. Thus, to show $\sigma_t'.T \in \gamma_s(T_1^\sharp)$, we compute $T_1^\sharp = S^\sharp \xrightarrow{T^\sharp} D^\sharp$.

We prove this by structural induction on the statement S. For assignments and pointer arithmetic, both concrete and abstract rules follow the same source-to-target propagation pattern. For input, both rules add the tainted variable to the taint set. For load/store, the abstract rule propagates through the corresponding abstract cell and therefore over-approximates concrete memory locations represented by that cell. Finally, $\texttt{kill}^\sharp$ removes

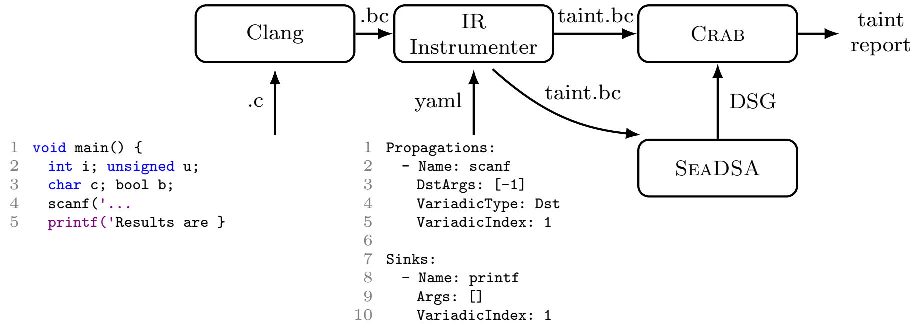


Figure 5.10: Overview of the taint-analysis workflow.

taint only on variables, never on summary memory cells, so killing remains conservative. Hence each concrete taint successor is represented by the abstract taint successor.

Therefore $\sigma'_t \in \gamma_t(\llbracket s \rrbracket^\sharp_T(\sigma^\sharp_t))$, Theorem 5.5.1 holds. □

5.6 Implementation

We implement our taint analysis on top of the SeaHorn [53] verification framework, targeting LLVM IR [67]. The implementation has two stages. The first stage constructs a memory abstraction with I-DSA, which we build on top of SeaDSA [54], a context-, field-, and array-sensitive pointer analysis. To improve precision, I-DSA represents memory-cell offsets as intervals rather than fixed constants. This design preserves field sensitivity better when offsets are symbolic or non-deterministic.

The second stage performs data-flow analysis using Crab [55], an abstract-interpretation library. Crab provides a language-independent IR (`CrabIR`), whose statements are annotated with explicit memory cells inferred by SeaDSA; this lets our taint analysis track taint across memory locations. We use the region domain [55] in Crab as the underlying memory domain. It relies on object summarization [14] to represent multiple concrete memory objects as a single abstract object. On top of this implementation, we add our taint domain described in Section 5.5 with domain reduction, which refines taint propagation using numerical invariants computed at each program point.

Fig. 5.10 presents the overall taint-analysis workflow. Following the configuration style of ClangTaint [19], we specify source functions, sinks, and propagation rules in YAML rather than hard-coding them on the source code. The input program is first compiled to LLVM IR and then instrumented according to the user-defined taint policy. For each

Suite	#Bench.	T/O		#Succ.	#Assert.	Warnings		Red.	Time (s)	
		OA	IA	Bench.		OA	IA		OA	IA
Experiment 1: Not inlined										
SMALL	32	0	0	32	83	39	39	0	2.5	2.4
IONC	25	2	2	23	107	43	41	4.7%	401.1	486.9
CURL	17	0	0	17	138	53	46	13.2%	4.2	4.2
COREUTILS	109	2	3	106	1753	181	169	6.6%	1994.3	1980.1
Subtotal	183	4	5	178	2081	316	295	6.6%	2402.1	2473.6
Experiment 2: Inlined										
SMALL	32	0	0	32	87	41	41	0	2.1	2.0
IONC	25	7	7	18	1970	66	65	1.5%	248.7	237.4
CURL	17	0	0	17	254	84	77	8.3%	3.0	3.3
COREUTILS	109	15	15	94	6689	138	120	13.0%	1178.2	1224.5
Subtotal	183	22	22	161	9000	329	303	7.9%	1432.1	1467.2

Figure 5.11: Comparison of SeaDSA + DFA (OA) and I-DSA + DFA (IA).

function with a taint policy, the IR instrumenter inserts intrinsics at the corresponding call sites to encode that policy explicitly in LLVM IR. For source functions, the inserted intrinsics mark the designated arguments and return value as tainted; for example, `scanf` marks its variadic output arguments as tainted. For sink functions, assertions are added to check that the designated sink arguments are untainted; for example, `printf` checks its format-dependent arguments before the call. This instrumentation makes taint behavior explicit in the low-level IR, avoids source-level annotations, and keeps the taint model configurable and analysis-independent.

5.7 Evaluation

To evaluate the effectiveness of our taint analysis (DSA + DFA), we compare a baseline configuration, SeaDSA + DFA, with the proposed configuration, I-DSA + DFA. The goal is to determine whether I-DSA improves field sensitivity in taint analysis while preserving the scalability of the baseline. We use the number of reported warnings as evidence of the effectiveness of I-DSA: a field-insensitive analysis may taint an entire object even when only one field is tainted, which can produce extra warnings. To assess scalability, we report

total running time and DSA analysis time. Both configurations use the same taint policies, the same DFA implementation in Crab, and Intervals as the numerical domain. The only difference is the DSA component used to compute the memory graphs.

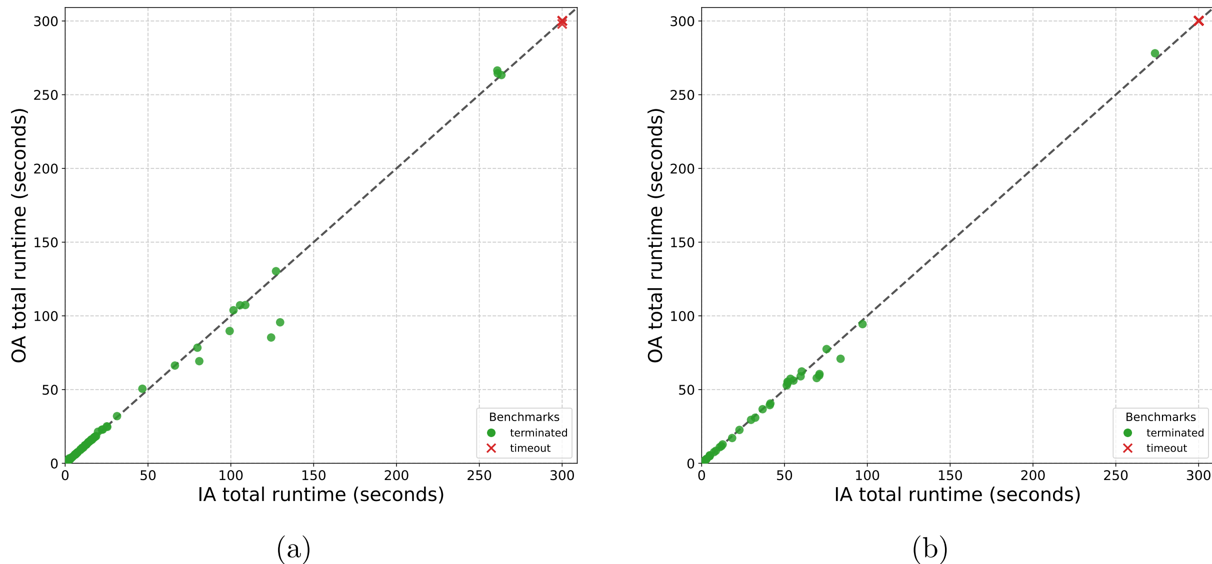
We ran two experiments: one without function inlining and one with function inlining. Without inlining, we assessed how well the interprocedural analysis detects taint flows across function calls and scales to larger programs. With inlining, more interprocedural flows are exposed within procedures, creating a stronger stress test for both precision and performance. Comparing the two settings helps evaluate whether the effect of I-DSA remains stable across different levels of program complexity. All experiments were run on an Ubuntu desktop with 12 Intel Core i5-11400F CPUs @ 2.60GHz and 93 GB RAM. We used a 300-second timeout for each benchmark. Timed-out cases are reported separately and excluded from the precision and time comparisons.

We evaluated four C benchmark suites. The SMALL suite contains 32 manually written programs that test core taint-analysis functionality. We also included three real-world suites: 25 benchmarks from IONC, a data-serialization library [2] that converts data between JSON and Amazon Ion; 17 benchmarks from CURL, a command-line tool [32] for URL-based data transfer; and 109 command-line programs from COREUTILS [43]. For IONC and CURL, we created API-level benchmarks instead of relying on top-level `main` functions so that we could directly exercise their public interfaces. For COREUTILS, we analyzed each program independently, using its `main` function as the entry point.

For each benchmark, we injected taint sources and sink checks (encoded as assertions) based on common C-library functions such as `scanf`/`printf` for stream I/O and `read`/`write` for file I/O. For real-world programs, we also added project-specific sources and sinks based on log functions and third-party libraries used by each project. These injections are only for evaluation and do not imply vulnerabilities in the original projects. Determining whether the reported warnings are true alarms requires manual inspection by the project developers.

Fig. 5.11 summarizes the results over 183 benchmarks (column #Bench.). We denote SeaDSA + DFA as (OA) and I-DSA + DFA as (IA).

In Experiment 1, IA reports 6.6% fewer warnings than OA on the 178 successful benchmarks (#Succ. Bench.), which contain 2081 checks (#Assert.). These results indicate that I-DSA improves taint-tracking precision by reducing spurious tainting of untainted fields. Although IA incurs a slightly higher total running time (column Time) than OA, both configurations remain on the same scale (Fig. 5.12a). This outcome is expected because I-DSA generates a more precise DSA graph for DFA, while IA still maintains comparable performance. DSA times for SeaDSA and I-DSA are also nearly identical: the averages



(a) (b)

Figure 5.12: OA vs. IA: (a) without inlining; (b) with inlining.

are 0.061s and 0.060s, the standard deviations are 0.136s and 0.134s, and the maxima are 0.860s and 0.870s, respectively. Outside these successful cases, both OA and IA time out on the same 4 benchmarks (column T/O) during the DFA phase, and IA has one additional timeout on a benchmark that OA completed in 298s.

In Experiment 2, both OA and IA successfully completed on 161 benchmarks with 9000 checks. Consistent with the previous findings, IA reports 7.9% fewer warnings than OA, confirming the precision benefit of I-DSA. Because more timed-out cases are excluded, the total running time is lower than in Experiment 1. IA still takes slightly more total time than OA, and both remain on the same scale (Fig. 5.12b). DSA times are again similar: SeaDSA and I-DSA average 0.048s and 0.047s, have standard deviations of 0.094s and 0.094s, and reach maxima of 0.490s and 0.500s, respectively. Beyond these successful cases, both configurations encounter the same 22 timeouts, including 6 caused by function inlining, 3 by DSA, and 13 by DFA. We attribute the three DSA-related timeouts in both configurations to specific tool implementation issues rather than algorithmic limitations. The increase in DFA-related timeouts is expected because function inlining raises the cost of value analysis by requiring interval invariants for many more variables and memory cells.

Overall, the two experiments indicate that I-DSA improves precision where SeaDSA loses field sensitivity while preserving comparable behavior on the existing benchmarks, with no measurable performance penalty.

5.8 Related Work

Many static taint analyzers formulate taint propagation as a data-flow problem, though they differ in their flow-, field-, and context-sensitivity, as well as their underlying heap abstraction.

Precision in memory-aware taint tracking is fundamentally limited by the accuracy of the underlying points-to analysis (PTA). A common strategy is to perform PTA to model memory, followed by a data-flow analysis (DFA) for taint propagation. We leverage a unification-based pointer analysis [100] called Data Structure Analysis (DSA) [68] for memory-effects inference. Other approaches, such as CompTaint [8] and PTaint [51], pursue similar integration with different designs. CompTaint runs a heap analysis to generate per-function points-to summaries, similar to DSA, and extends those summaries directly with taint data-flow facts. This yields reusable summaries that can be instantiated at different call sites to recover context sensitivity, avoiding whole-program reasoning for scalability. PTaint [51] also integrates DFA but it directly tracks taint flows inside a pointer analysis. As it discovers points-to information, it simultaneously updates taint facts. This unified design allows points-to information to inform taint propagation immediately.

Our approach refines DSA through partial node collapse, unlike standard DSA, which merges all cells within a node into a field-insensitive state (i.e., collapse) after any incompatible unification. Our approach uses interval offsets to localize cell merges. When a unification conflict occurs, we merge only the affected cells, allowing the remaining cells to preserve precision. This directly improves taint-tracking precision. Other works also improve DSA field sensitivity. SeaDSA [54] formalizes unification for array-representative nodes. TeaDSA [66] leverages type information to mitigate type-incompatible unifications. Our approach is closest to DSAbin [9], which uses constant-set offset abstractions to prevent full node collapse. However, we achieve higher precision through single-node unification and, unlike DSAbin [9], which focuses on binary analysis, we analyze LLVM IR, the original target of DSA.

Our DFA treats taint facts as lattice elements to identify tainted variables or memory locations. Other taint-tracking strategies have also been proposed. A common framework is *Inter-procedural Finite Distributive Subset* (IFDS) [92], which constructs an *Exploded Super-Graph* with instruction-fact pairs as nodes and data-flow edges. Ultimately, IFDS reduces data-flow analysis to graph reachability. Practical implementations include Phasar [96], which applies IFDS to analyze LLVM IR programs and builds a taint analysis over a taint-fact domain, and FlowDroid [3], which uses IFDS for taint analysis on Android apps.

Our DFA follows abstract interpretation [25] to approximate taint flow. Specifically, it follows the abstract semantic dependency framework introduced by Cousot [24], where taint analysis is formulated as a sound abstraction for tracking value dependencies. At each program point, the analysis over-approximates the set of variables whose values may depend on tracked (tainted) inputs. Mopsa-Nexp [88] also introduces taint analysis to verify that attacker-controlled inputs cannot reach and trigger critical runtime errors (i.e., nonexploitability). Although both works leverage numerical domains for precision, they differ in scope and granularity. Mopsa-Nexp focuses on leaks that can trigger runtime errors, whereas our work identifies data leaks to sinks such as diagnostic logging. Unlike Mopsa-Nexp, which does not track taint at the level of individual memory locations (field-insensitive), our work uses I-DSA to provide the field sensitivity required by this objective.

5.9 Conclusion

In this work, we present a taint analysis for LLVM IR that follows user-specified taint sources, sinks, and propagation rules. The analysis combines Data Structure Analysis (DSA) for memory modeling with data-flow analysis (DFA) for taint propagation. We introduce I-DSA, a DSA refinement that improves field sensitivity, and build DFA on top of an existing abstract-interpretation-based value analysis to prune infeasible taint facts. Our results show that this design provides a practical precision-cost trade-off for memory-aware taint analysis.

Chapter 6

Conclusion and Future Work

In this thesis, we developed and implemented a suite of automated, sound static analyses for memory safety and information-flow security. Within the abstract-interpretation framework, our approach combines abstract-domain design, sound abstraction, and practical effectiveness demonstrated on motivating examples. We designed these analyses to be targeted where needed, modular where possible, and efficient where practicable. Specifically, we contributed analyses for memory safety (Chapter 3), numerical reasoning (Chapter 4), and taint analysis integrated with pointer analysis and dataflow analysis (Chapter 5). Each analysis was engineered to satisfy verification requirements, address precision bottlenecks, and scale well. Our evaluation on real-world benchmarks demonstrates the practical usability of our analyses for real codebases.

6.1 Summary

Object Invariants. We designed an abstract domain (Chapter 3) and developed a static analysis for memory safety. The core idea is to infer object invariants, that is, properties that hold for all memory objects in a memory bank (i.e., a group of objects originating from the same allocation site). By representing these objects with a single summary object, we can infer shared properties, such as bounds on buffer accesses. A key observation is that an additional cache slot can isolate invariants for the Most Recently Used (MRU) object. Because the MRU object is the most frequently accessed and modified, its invariants are often temporarily “unstable.” Once updates are complete, these invariants are typically restored and can be merged back into the summary representation. This design maintains high precision for active objects while efficiently summarizing shared properties

of the remaining objects in the same bank. Following this idea, we made the following contributions:

- Introduced the Recently-Used Memory Model (RUMM), which partitions memory objects into banks. Each bank maintains a dedicated slot to track the Most Recently Used (MRU) object independently of the rest of the collection.

- Defined a small-step operational semantics under RUMM based on a "cache-aware" memory access pattern. In this model, memory operations first target the MRU slot (a cache hit). If a different object is accessed (a cache miss), the current MRU object is flushed back to the bank storage, and the newly accessed object is loaded into the MRU slot.

- Developed an abstract domain based on RUMM, implemented as a reduced product of per-bank domains and a scalar domain. Each bank domain is further structured as a product of a summary-object domain and an MRU-object domain. Reduction between the MRU and scalar domains enables refinement of relational invariants between object fields and scalar variables, with the entire framework parameterized by the choice of underlying numerical domain.

- Implemented a hybrid verification pipeline, AI4BMC, which leverages abstract interpretation to prove memory safety properties. Any safety checks that remain unproven are passed to a Bounded Model Checker (BMC). This pipeline is designed to mitigate the incompleteness of static analysis while significantly reducing the computational overhead of BMC.

Numerical Abstractions. In Chapter 4, we designed a numerical domain, Template DBM, and integrated it into the AI4BMC pipeline for memory-safety verification. The central challenge was to develop a domain that can express memory-safety constraints while remaining efficient in practice. We observed that existing abstract domains, such as Intervals and Zones, lack the precision needed to capture Two Variables Per Inequality (TVPI) constraints, such as $4x - y \leq -4$. These relational constraints are essential for relating buffer-access offsets to allocated sizes. To address this, Template DBM captures a specific subset of TVPI constraints under a coefficient template, offering higher expressivity than Zones while remaining more efficient than Polyhedra. For efficiency, we use the Difference-Bound Matrix (DBM) data structure to represent these constraints, which enables efficient implementations of domain operations. In this work, we:

- Defined a new Template DBM by assigning dimensions to variables scaled by specific coefficients. We developed the essential saturation operations (i.e., explicit representation of implicit constraints) by adapting Fourier-Motzkin elimination to the Template DBM framework.

- Defined the Template DBM numerical domain based on the Template DBM, with essential domain operations for abstract interpretation.

- Evaluated Template DBM in the AI4BMC pipeline for memory safety verification. Our results showed that Template DBM successfully captures the relational constraints we target while maintaining competitive scalability on production-level benchmarks.

Taint Analysis. We designed a taint analysis that integrates pointer analysis and dataflow analysis to reason about how tainted sources reach sinks. The core challenge is to track taint through memory accesses and to combine this reasoning with other analyses to improve precision. By combining taint analysis with pointer analysis, we can track taint propagation through memory more accurately. However, as the motivating examples in Chapter 5 show, the pointer analysis we used (i.e., data structure analysis (DSA)) loses precision. This limitation motivated us to improve DSA by abstracting access offsets as intervals rather than constants. This design avoids the loss of field sensitivity when pointer accesses use symbolic indices. We then defined taint semantics systematically and built a dataflow analysis to track taint flow. Concretely, we:

- Built a new DSA with interval offset, which overcomes the field sensitivity loss of the original DSA when analyzing symbolic accesses. This improvement allows us to more accurately track taint flow through memory accesses.

- Defined a formal semantics for taint propagation. This semantics shows how variables and memory locations become tainted and how taint information can be propagated through transfer functions.

- Experimented on both small artificial examples and real-world benchmarks. We showed that with the improved DSA, we can track taint flow more accurately while still maintaining good scalability.

6.2 Future Work

In closing, we discuss future directions for each component of this thesis.

6.2.1 "Cache" More

In Chapter 3, RUMM maintains only one cache slot for the most recently used (MRU) object in each memory bank. This design is simple and efficient, but it relies on a heuristic assumption about memory-access patterns: programs update one object at a time. However, this assumption does not always hold. For example, consider the following C program:

```c
struct byte_buf ary[2];
ary[0].len = 10;
ary[1].len = 20;
ary[0].cap = 15;
ary[1].cap = 25;
assert(ary[0].len <= ary[0].cap);
assert(ary[1].len <= ary[1].cap);
```

Here, the two objects are allocated at the same site and updated in an interleaved order. With a single cache slot, our analysis repeatedly packs the current MRU object into the summary and unpacks the newly accessed object, eventually losing invariants such as length not exceeding capacity. In other words, the single-MRU design works well when one object is under construction or update, but not when updates to multiple objects from the same allocation site are interleaved.

A possible solution is to use k-object sensitivity, similar to the k-limiting approach introduced by [61]. In this extension, each bank includes multiple cache slots organized as a circular (ring) buffer to preserve access order. When all cache slots miss, the object at the end of the buffer is flushed, that slot is updated, and the index advances to the next position.

Supporting this extension would require changes to both the semantics and the abstract domain. The semantics must account for multiple cache slots per bank, and the abstract domain must maintain relationships between the summary object and the cached objects. The reduction operators would also need to be adapted so that relational information can be propagated soundly between cache slots and the summary.

The main trade-off is precision versus efficiency. Increasing k allows the analysis to preserve invariants for more simultaneously active objects, which can improve precision for the example listed above. However, each additional cache slot introduces more ghost variables and more relational constraints, increasing the cost of transfer functions, joins, reductions, and fixpoint computation.

Algorithm 6.1 Inclusion test by local linear reconstruction

```
1: function LogInclusion(m̄, n̄)
2:     for all ℓ ≡ ax − by ≤ c in n̄ do
3:         proved = m̄_{ax,by} ≤ c
4:         if ¬proved ∧ (ℓ1, ℓ2) = FindClosestEdges(m̄, ax − by) then
5:             proved = LinearCombine(ℓ1, ℓ2, ax − by) ≤ c
6:         if proved = FALSE then
7:             return FALSE
8:     return TRUE
```

6.2.2 Toward More Precise Template DBM Operations

We described the Template DBM domain and its operations in Chapter 4. However, this design still leaves several directions for improving precision, especially in its join and inclusion-test operators.

Inclusion test. The current $\sqsubseteq^{tDBM}$ operator is defined by directly reusing the underlying DBM operation. This design keeps inclusion checking efficient, but it can be incomplete with respect to the geometric meaning of the represented constraints. For example, consider the following two abstract states:

$$s_1 = \{1 \leq x,\ y - x \leq 0\} \qquad s_2 = \{1 \leq x,\ y - 2x \leq 0\}.$$

Semantically, $s_1 \sqsubseteq^{tDBM} s_2$, because $1 \leq x$ implies $x \leq 2x$, and therefore $y \leq x$ implies $y \leq 2x$. DBM stores constraints syntactically, so it does not derive $s_1 \sqsubseteq^{tDBM} s_2$ from this relationship. Put differently, $2x$ is treated as a separate dimension, without explicit knowledge of its relation to the dimension for x. Therefore, DBM inclusion fails to prove this case; the result remains sound but incomplete.

A simple solution is to follow the strategy used in the TVPI domain. As shown in Algorithm 6.1, for each TVPI constraint ℓ in $\bar{n}$, we first check whether $\bar{m}$ already contains a constraint with the same difference and a tighter bound. If not, we try to reconstruct a TVPI constraint by taking a linear combination of two TVPI constraints from $\bar{m}$ to derive a new TVPI constraint for the same difference. If such a combination is found and its bound is tighter, then entailment holds for ℓ. Otherwise, the inclusion test returns FALSE.

Join. We chose to implement join by reusing the underlying DBM join, which is efficient but still imprecise. We showed an example in Section 4.4. This could be improved by

implementing convex-hull-inspired joins that better balance precision and operational cost.

6.2.3 Better DSA and DFA.

In Chapter 5, we presented data structure analysis and dataflow analysis; however, both analyses still have limitations, and further improvements are possible.

Parameterized offset abstraction of DSA. We described an improved DSA with interval offset abstraction. Although interval abstraction is effective in many cases, it can still be too coarse. For example, suppose a pointer is computed as $p = q + (i \times s)$ through a symbolic offset i. Intervals only approximate the range of i. Suppose we compute the offset range of p as $[12, +\infty]$. We still do not know the exact offsets that this pointer can reference; a naive interpretation is $\{12, 13, 14, \ldots\}$. However, this pointer arithmetic is usually used for array access, where elements have a fixed stride. Since s is constant, the feasible offsets should follow $\{12, 12 + s, 12 + 2 \times s, \ldots\}$. We can therefore refine the offset abstraction beyond plain intervals. One option is to adopt the abstraction used in DSABin [9], where a constant-set abstract domain is applied. Another option is to use a reduced product of an interval domain and a congruence domain [50]. In the above example, we can maintain $[12, +\infty] \wedge 12 \mod s$ to capture the access pattern more precisely. More generally, offset abstraction is a key design choice that determines DSA precision, so it is useful to formalize a parameterized offset-abstraction framework that supports multiple offset abstractions.

DFA We used DFA for taint-flow tracking, which handles only explicit data flows. However, implicit flows [35] through control dependencies also matter. For example,

```
1 int cond = source();
2 int x = 0;
3 if (cond > 0) {
4     x = 1;
5 }
6 sink(x);
```

Here, even though `x` is not directly assigned from `cond`, its value is still controlled by `cond` through control dependence. A standard solution is to introduce a program-counter taint pc, which tracks whether the current program point is control-dependent on tainted condition variables. That is, $pc \in \{\mathsf{clean}, \mathsf{tainted}\}$. We then extend the propagation rule as follows:

$$T' := S \xrightarrow{T} D \stackrel{\text{def}}{=} T \setminus D \ \cup \ \{l \in D \mid S \ \cap \ T \neq \emptyset \vee pc = \mathsf{tainted}\}$$

We define $\texttt{tainted}(e, T) \stackrel{\text{def}}{=} (\texttt{vs}(e) \cap T \neq \emptyset)?\mathsf{tainted} : \mathsf{clean}$ to compute the taint status of expression e by checking whether any variable in $\texttt{vs}(e)$ is tainted. We can then define the semantics of the if-then-else statement as follows:

$$b = \mathsf{true}\ \frac{pc' = pc \sqcup_{\mathcal{T}} \texttt{tainted}(b, T) \qquad [\![s_t]\!](\rho, h, (T, pc')) = (\rho_1, h_1, (T_1, _))}{[\![\mathsf{if}(b)\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f]\!](\rho, h, (T, pc)) = (\rho_1, h_1, (T_1, pc))}$$

$$b = \mathsf{false}\ \frac{pc' = pc \sqcup_{\mathcal{T}} \texttt{tainted}(b, T) \qquad [\![s_f]\!](\rho, h, (T, pc')) = (\rho_2, h_2, (T_2, _))}{[\![\mathsf{if}(b)\ \mathsf{then}\ s_t\ \mathsf{else}\ s_f]\!](\rho, h, (T, pc)) = (\rho_2, h_2, (T_2, pc))}$$

This rule shows how a single transition in the concrete semantics can involve condition-based taint propagation. If taint arises from variables used in condition b, then pc' records the taint status of the current control context. We then use pc' to propagate taint to variables in the branch body. The same idea can be extended to while statements.

References

[1] Salman Ahmed, Hans Liljestrand, Hani Jamjoom, Matthew Hicks, N. Asokan, and Danfeng Yao. Not all data are created equal: Data and pointer prioritization for scalable protection against data-oriented attacks. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 1433–1450. USENIX Association, 2023.

[2] Amazon Ion Contributors. Amazon Ion. https://amazon-ion.github.io/ion-docs/. Accessed: 2026-04-27.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269. ACM, 2014.

[4] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008.

[5] Gogul Balakrishnan and Thomas W. Reps. Recency-abstraction for heap-allocated storage. In Kwangkeun Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 221–239. Springer, 2006.

[6] George Balatsouras and Yannis Smaragdakis. Structure-sensitive points-to analysis for C and C++. In Xavier Rival, editor, *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, volume 9837 of *Lecture Notes in Computer Science*, pages 84–104. Springer, 2016.

[7] Kenny Ballou and Elena Sherman. Incremental transitive closure for zonal abstract domain. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *Lecture Notes in Computer Science*, pages 800–808. Springer, 2022.

[8] Subarno Banerjee, Siwei Cui, Michael Emmi, Antonio Filieri, Liana Hadarean, Peixuan Li, Linghui Luo, Goran Piskachev, Nicolás Rosner, Aritra Sengupta, Omer Tripp, and Jingbo Wang. Compositional taint analysis for enforcing security policies at scale. In Satish Chandra, Kelly Blincoe, and Paolo Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1985–1996. ACM, 2023.

[9] Sadra Bayat-Tork, Nicholas Coughlin, Alicia Michael, James Tobler, and Kirsten Winter. Data structure analysis for binaries. In Sebastian Junges and Guy Katz, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2026)*, pages 235–254, Cham, 2026. Springer Nature Switzerland.

[10] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[11] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In Ron Cytron and Rajiv Gupta, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 196–207. ACM, 2003.

[12] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[13] Bor-Yuh Evan Chang and K. Rustan M. Leino. Inferring object invariants: Extended abstract. In Agostino Cortesi and Francesco Logozzo, editors, *Proceedings of the First International Workshop on Abstract Interpretation of Object-oriented Languages, AIOOL@VMCAI 2005, Paris, France, January 21, 2005*, volume 131 of *Electronic Notes in Theoretical Computer Science*, pages 63–74. Elsevier, 2005.

[14] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In Bernard N. Fischer, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, pages 296–310. ACM, 1990.

[15] Aziem Chawdhary, Edward Robbins, and Andy King. Incrementally closing octagons. *Formal Methods Syst. Des.*, 54(2):232–277, 2019.

[16] Marc Chevalier and Jérôme Feret. Sharing ghost variables in a collection of abstract domains. In Dirk Beyer and Damien Zufferey, editors, *Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings*, volume 11990 of *Lecture Notes in Computer Science*, pages 158–179. Springer, 2020.

[17] Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. Code-level model checking in the software development workflow. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE-SEIP 2020: 42nd International Conference on Software Engineering, Software Engineering in Practice, Seoul, South Korea, 27 June - 19 July, 2020*, pages 11–20. ACM, 2020.

[18] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000.

[19] Clang Static Analyzer. Taint analysis configuration. https://clang.llvm.org/docs/analyzer/user-docs/TaintAnalysisConfiguration.html. Accessed: 2026-04-27.

[20] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[21] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In John R. Wright, Larry Landweber, Alan J. Demers, and Tim Teitelbaum, editors, *Conference Record of the Tenth Annual ACM Symposium on Principles of*

Programming Languages, Austin, Texas, USA, January 1983, pages 117–126. ACM Press, 1983.

[22] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[23] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[24] Patrick Cousot. Abstract semantic dependency. In Bor-Yuh Evan Chang, editor, *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, volume 11822 of *Lecture Notes in Computer Science*, pages 389–410. Springer, 2019.

[25] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[26] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282. ACM Press, 1979.

[27] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[28] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the astrée static analyzer. In Mitsu Okada and Ichiro Satoh, editors, *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers*, volume 4435 of *Lecture Notes in Computer Science*, pages 272–300. Springer, 2006.

[29] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In Martin Hofmann, editor, *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6604 of *Lecture Notes in Computer Science*, pages 456–472. Springer, 2011.

[30] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. ACM Press, 1978.

[31] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c - A software analysis perspective. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*, Lecture Notes in Computer Science, pages 233–247. Springer, 2012.

[32] curl Contributors. curl. https://github.com/curl/curl. Accessed: 2026-04-27.

[33] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[34] David Delmas and Jean Souyris. Astrée: From research to industry. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, volume 4634 of *Lecture Notes in Computer Science*, pages 437–451. Springer, 2007.

[35] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[36] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[37] Ioannis Doudalis, James A. Clause, Guru Venkataramani, Milos Prvulovic, and Alessandro Orso. Effective and efficient memory protection using dynamic tainting. *IEEE Trans. Computers*, 61(1):87–100, 2012.

[38] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, 2014.

[39] Mark W. Eichin and Jon A. Rochlis. With microscope and tweezers: An analysis of the internet virus of november 1988. *ACM SIGOPS Operating Systems Review*, 23(2):32–40, 1989.

[40] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, volume 6528 of *Lecture Notes in Computer Science*, pages 10–30. Springer, 2010.

[41] Pietro Ferrara, Francesco Logozzo, and Manuel Fähndrich. Safer unsafe code for .net. In Gail E. Harris, editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 329–346. ACM, 2008.

[42] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In Yuval Yarom and Sarah Zennou, editors, *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*. USENIX Association, 2020.

[43] Free Software Foundation. GNU Coreutils. `https://www.gnu.org/software/coreutils/`, 2026. Accessed: 2026-04-27.

[44] Jdrzej Fulara, Konrad Durnoga, Krzysztof Jakubczyk, and Aleksy Schubert. Relational abstract domain of weighted hexagons. *Electron. Notes Theor. Comput. Sci.*, 267(1):59–72, October 2010.

[45] Graeme Gange, Zequn Ma, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. A fresh look at zones and octagons. *ACM Trans. Program. Lang. Syst.*, 43(3):11:1–11:51, 2021.

[46] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Exploiting sparsity in difference-bound matrices. In Xavier Rival, editor, *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, volume 9837 of *Lecture Notes in Computer Science*, pages 189–211. Springer, 2016.

[47] GNU Project. Gnu core utilities official page.

[48] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223. ACM, 2005.

[49] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas W. Reps, and Shmuel Sagiv. Numeric domains with summarized dimensions. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 512–529. Springer, 2004.

[50] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30(3-4):165–190, 1989.

[51] Neville Grech and Yannis Smaragdakis. P/taint: unified points-to and taint analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA):102:1–102:28, 2017.

[52] Sumit Gulwani, Ashish Tiwari, and George C. Necula. Join algorithms for the theory of uninterpreted functions. In Kamal Lodaya and Meena Mahajan, editors, *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16-18, 2004, Proceedings*, volume 3328 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 2004.

[53] Arie Gurfinkel, Temesghen Kahsai, and Jorge A. Navas. Seahorn: A framework for verifying C programs (competition contribution). In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 447–450. Springer, 2015.

[54] Arie Gurfinkel and Jorge A. Navas. A context-sensitive memory model for verification of C/C++ programs. In Francesco Ranzato, editor, *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, volume 10422 of *Lecture Notes in Computer Science*, pages 148–168. Springer, 2017.

[55] Arie Gurfinkel and Jorge A. Navas. Abstract interpretation of LLVM with a region-based memory model. In Roderick Bloem, Rayna Dimitrova, Chuchu Fan, and Natasha Sharygina, editors, *Software Verification - 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18-19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18-19, 2021, Revised Selected Papers*, volume 13124 of *Lecture Notes in Computer Science*, pages 122–144. Springer, 2021.

[56] Jacob M. Howe and Andy King. Logahedra: A new weakly relational domain. In Zhiming Liu and Anders P. Ravn, editors, *Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings*, volume 5799 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2009.

[57] Jing Hua and Ping Wang. Security vulnerabilities in facebook data breach. In Shahram Latifi, editor, *ITNG 2024: 21st International Conference on Information Technology-New Generations*, pages 159–166, Cham, 2024. Springer Nature Switzerland.

[58] Wei Huang, Yao Dong, Ana L. Milanova, and Julian Dolby. Scalable and precise taint analysis for android. In Michal Young and Tao Xie, editors, *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 106–117. ACM, 2015.

[59] Bill Huston. Single-chip microcomputers can be easy to program. In *American Federation of Information Processing Societies: 1982 National Computer Conference, 7-10 June, 1982, Houston, Texas, USA*, volume 51 of *AFIPS Conference Proceedings*, pages 85–93. AFIPS Press, 1982.

[60] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.

[61] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In Richard A. DeMillo, editor, *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982*, pages 66–74. ACM Press, 1982.

[62] Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. Modular static analysis of string manipulations in C programs. In Andreas Podelski, editor, *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 of *Lecture Notes in Computer Science*, pages 243–262. Springer, 2018.

[63] Temesghen Kahsai, Rody Kersten, Philipp Rümmer, and Martin Schäf. Quantified heap invariants for object-oriented programs. In Thomas Eiter and David Sands, editors, *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, volume 46 of *EPiC Series in Computing*, pages 368–384. EasyChair, 2017.

[64] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[65] Stephen Cole Kleene. *Introduction to Metamathematics.* The University Series in Higher Mathematics. D. Van Nostrand, Princeton, NJ, 1952. Cited on page 28.

[66] Jakub Kuderski, Jorge A. Navas, and Arie Gurfinkel. Unification-based pointer analysis without oversharing. In Clark W. Barrett and Jin Yang, editors, *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, pages 37–45. IEEE, 2019.

[67] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004.

[68] Chris Lattner and Vikram S. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 129–142. ACM, 2005.

[69] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. Verus: A practical foundation for systems verification. In Emmett Witchel, Christopher J. Rossbach, Andrea C. Arpaci-Dusseau, and Kimberly Keeton, editors, *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024*, pages 438–454. ACM, 2024.

[70] Vincent Laviron and Francesco Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In Neil D. Jones and Markus Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, volume 5403 of *Lecture Notes in Computer Science*, pages 229–244. Springer, 2009.

[71] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, Lecture Notes in Computer Science, pages 348–370. Springer, 2010.

[72] Jacques-Louis Lions, Lennart Luebeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran. Ariane 5 flight 501 failure report by the inquiry board, 1996.

[73] LLVM Project. libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed: 2026-04-23.

[74] Meta. The facts on news reports about facebook data. Meta Newsroom, 2021. Accessed: April 22, 2026.

[75] Bertrand Meyer. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., USA, 1997.

[76] Microsoft Security Response Center. We need a safer systems programming language, July 2019. Accessed: 2026-04-23.

[77] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects, Second Symposium, PADO 2001, Aarhus, Denmark, May 21-23, 2001, Proceedings*, volume 2053 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2001.

[78] Antoine Miné. The octagon abstract domain. In Elizabeth Burd, Peter Aiken, and Rainer Koschke, editors, *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001*, page 310. IEEE Computer Society, 2001.

[79] Antoine Miné. *Weakly Relational Numerical Abstract Domains. (Domaines numériques abstraits faiblement relationnels)*. PhD thesis, École Polytechnique, Palaiseau, France, 2004.

[80] Antoine Miné. The octagon abstract domain. *High. Order Symb. Comput.*, 19(1):31–100, 2006.

[81] Antoine Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Found. Trends Program. Lang.*, 4(3-4):120–372, 2017.

[82] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Mopsa-c: Modular domains and relational abstract interpretation for C programs (competition contribution). In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 565–570. Springer, 2023.

[83] National Institute of Standards and Technology (NIST). CVE-2025-14847 Detail: Mongodb heap-based buffer over-read, December 2025. Accessed: 2026-04-19.

[84] National Institute of Standards and Technology (NIST). CVE-2025-40551 Detail: Solarwinds web help desk untrusted data deserialization vulnerability. National Vulnerability Database, January 2026. Accessed: 2026-04-19.

[85] National Institute of Standards and Technology (NIST). CVE-2026-20128: Cisco nx-os software information disclosure, February 2026. Accessed: 2026-04-19.

[86] Charles G Nelson. An n^{logn} algorithm for the two-variable-per-constraint linear programming satisfiability problem. Technical report, Stanford University, Stanford, CA, USA, 1978.

[87] Chris Okasaki and Andy Gill. Fast mergeable integer maps. In *Notes of the ACM SIGPLAN Workshop on ML*, pages 77–86, 1998.

[88] Francesco Parolini and Antoine Miné. Sound abstract nonexploitability analysis. In Rayna Dimitrova, Ori Lahav, and Sebastian Wolff, editors, *Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part II*, volume 14500 of *Lecture Notes in Computer Science*, pages 314–337. Springer, 2024.

[89] Siddharth Priya, Yusen Su, Yuyan Bao, Xiang Zhou, Yakir Vizel, and Arie Gurfinkel. Bounded model checking for llvm. In *# PLACEHOLDER_PARENT_METADATA_VALUE#*, pages 214–224, 2022.

[90] Siddharth Priya, Yusen Su, Yuyan Bao, Xiang Zhou, Yakir Vizel, and Arie Gurfinkel. Bounded model checking for LLVM. In Alberto Griggio and Neha Rungta, editors, *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, pages 214–224. IEEE, 2022.

[91] Siddharth Priya, Xiang Zhou, Yusen Su, Yakir Vizel, Yuyan Bao, and Arie Gurfinkel. Verifying verified code. In Zhe Hou and Vijay Ganesh, editors, *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceedings*, volume 12971 of *Lecture Notes in Computer Science*, pages 187–202. Springer, 2021.

[92] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In Ron K. Cytron and Peter Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61. ACM Press, 1995.

[93] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[94] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Low-level liquid types. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 131–144. ACM, 2010.

[95] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 25–41. Springer, 2005.

[96] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Phasar: An inter-procedural static analysis framework for C/C++. In Tomás Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*, volume 11428 of *Lecture Notes in Computer Science*, pages 393–410. Springer, 2019.

[97] Axel Simon, Andy King, and Jacob M. Howe. Two variables per linear inequality as an abstract domain. In Michael Leuschel, editor, *Logic Based Program Synthesis and Tranformation, 12th International Workshop, LOPSTR 2002, Madrid, Spain, September 17-20,2002, Revised Selected Papers*, volume 2664 of *Lecture Notes in Computer Science*, pages 71–89. Springer, 2002.

[98] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. Making numerical program analysis fast. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 303–313. ACM, 2015.

[99] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. Fast polyhedra abstract domain. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 46–59. ACM, 2017.

[100] Bjarne Steensgaard. Points-to analysis in almost linear time. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 32–41. ACM Press, 1996.

[101] Yusen Su, Jorge A. Navas, Arie Gurfinkel, and Isabel Garcia-Contreras. Automatic inference of relational object invariants. In Shankaranarayanan Krishna, Sriram Sankaranarayanan, and Ashutosh Trivedi, editors, *Verification, Model Checking, and Abstract Interpretation - 26th International Conference, VMCAI 2025, Denver, CO, USA, January 20-21, 2025, Proceedings, Part I*, volume 15529 of *Lecture Notes in Computer Science*, pages 214–236. Springer, 2025.

[102] Chuyue Sun, Yican Sun, Daneshvar Amrollahi, Ethan Zhang, Shuvendu Lahiri, Shan Lu, David Dill, and Clark Barrett. Veristruct: Ai-assisted automated verification of data-structure modules in verus. In Sebastian Junges and Guy Katz, editors, *Tools*

and Algorithms for the Construction and Analysis of Systems, pages 109–128, Cham, 2026. Springer Nature Switzerland.

[103] Joseph Tafese, Siddharth Priya, Giuliano Losa, Arie Gurfinkel, and Graydon Hoare. A tale of two case studies: A unified exploration of rust verification with SEABMC. In Ahmed Irfan and Daniela Kaufmann, editors, *Proceedings of the 25th Conference on Formal Methods in Computer-Aided Design, FMCAD 2025, Menlo Park, CA, USA, October 6-10, 2025*. TU Wien Academic Press, 2025.

[104] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[105] Antoine Toubhans, Bor-Yuh Evan Chang, and Xavier Rival. An abstract domain combinator for separately conjoining memory abstractions. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, volume 8723 of *Lecture Notes in Computer Science*, pages 285–301. Springer, 2014.

[106] U.S. House of Representatives, Committee on Oversight and Government Reform. The Equifax Data Breach. Majority staff report, U.S. House of Representatives, December 2018. 115th Congress. Accessed: 2026-04-23.

[107] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. Verifying dynamic trait objects in rust. In *44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022*, pages 321–330. IEEE, 2022.

[108] Yayi Wang, Shenao Wang, Jian Zhao, Shaosen Shi, Ting Li, Yan Cheng, Lizhong Bian, Kan Yu, Yanjie Zhao, and Haoyu Wang. YASA: scalable multi-language taint analysis on the unified AST at ant group. *CoRR*, abs/2601.17390, 2026.

[109] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting program specification synthesis by large language models using static analysis and program verification. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II*, Lecture Notes in Computer Science, pages 302–328. Springer, 2024.

[110] Jie Zhou, John Criswell, and Michael Hicks. Fat pointers for temporal memory safety of C. *Proc. ACM Program. Lang.*, 7(OOPSLA1):316–347, 2023.