

# Abstract Interpretation with BDDs

A Gentle Guide to Program Verification

The [bdd-rs](#) contributors

December 19, 2025

# Preface



Welcome to this comprehensive guide on abstract interpretation combined with Binary Decision Diagrams (BDDs). Whether you're a student encountering formal methods for the first time, a practitioner seeking to apply verification techniques, or a researcher exploring new analysis approaches, this guide meets you where you are and takes you further.

## 1. Who This Guide Is For

This guide serves multiple audiences with diverse backgrounds and goals:

**Complete Beginners:** New to program verification and formal methods? Start with Part I. The guide builds intuition through concrete examples before introducing mathematical formalism. No prior knowledge of abstract interpretation or BDDs assumed — just basic programming experience and curiosity.

**Practitioners:** Software engineers interested in applying verification techniques? Focus on Part I for conceptual understanding, then jump to Part III (Section 16, Section 18) for implementation details and benchmarks.

**Researchers and Graduate Students:** Already familiar with program analysis basics? Skim Part I and dive into Part II, which provides complete mathematical foundations, formal proofs, and connections to research literature.

**Educators:** This guide is structured to support a one-semester graduate course on program analysis, with progressive exercises, worked examples, and discussion prompts throughout.

## 2. What Makes This Guide Different

Unlike traditional academic papers or reference manuals, this guide:

- **Tells a story:** Concepts build progressively, motivating each idea before formalizing it.
- **Shows real code:** Every concept is backed by working Rust implementations from the `bdd-rs` library.
- **Balances rigor and intuition:** Mathematical precision maintained while providing accessible explanations.
- **Connects to practice:** Real-world applications — from security analysis to smart contracts — demonstrate where these techniques shine.
- **Provides multiple paths:** Skip sections that don't match your goals, or deep-dive into formal theory where interested.

## 3. Structure of This Guide

This guide is organized into three parts, each serving a distinct purpose:

**Part I: Gentle Introduction** (Section 1 through Section 6) starts from first principles. We build intuition about program abstraction, control flow, and symbolic representations through concrete examples. Core concepts like abstract domains, lattice operations, and BDD-based path sensitivity emerge naturally from practical problems. This part is accessible to anyone with programming background.

**Part II: Deep Dive** (Section 7 through Section 15) provides rigorous mathematical foundations. We develop complete lattice theory, fixpoint theorems, Galois connections, and domain combination operators. Each chapter includes formal definitions, proofs, and implementation guidance. Topics include abstract transformers (Section 9), reduced products (Section 11), BDD path sensitivity (Section 12), and specialized domains for strings (Section 13), pointers (Section 14), and precision refinement (Section 15).

**Part III: Applications & Future Directions** (Section 16 through Section 21) bridges theory and practice. We explore security analysis with taint tracking, inter-procedural analysis with context sensitivity, performance optimization techniques, and the emerging integration of AI with formal methods. The guide concludes with a complete case study analyzing an access control system and directions for future research.

## 4. How to Use This Guide

### 4.1 For Learning (First Time Through)

1. Start with the **Prologue** to understand why static analysis matters.
2. Work through **Part I** sequentially — concepts build on each other.
3. Try the code examples from the `bdd-rs` repository as you go.
4. When encountering formalism in Part II, don't panic — intuition comes first, rigor follows.
5. Use the **exercises** (marked with difficulty levels) to test understanding.
6. Skip Part II proofs on first reading if they feel overwhelming — return when ready.

### 4.2 For Reference (Coming Back)

- Use the **table of contents** and **cross-references** to navigate directly to topics.
- **Info boxes** and **margin notes** provide quick refreshers on key concepts.
- **Example references** link to concrete code in the repository.
- The **index** (when complete) will enable rapid lookup of specific terms.

### 4.3 Reading Paths

We've marked sections with icons to help you navigate:



Must-read for everyone



Gentler content for newcomers



Advanced

Formal details and proofs



Implementation Focus

Code and practical details

## 5. Prerequisites

### Assumed knowledge:

- Basic programming experience (preferably in Rust, but examples are explained).
- Undergraduate discrete mathematics (sets, functions, relations, basic logic).
- Familiarity with graphs and trees.

### Helpful but not required:

- Understanding of compilers or program analysis.
- Experience with functional programming patterns.
- Exposure to Boolean algebra or propositional logic.

### For Part II (advanced readers):

- Mathematical maturity for reading proofs.
- Comfort with order theory (partially ordered sets, lattices).
- Familiarity with fixpoint theorems.

Don't let prerequisites intimidate you — the guide builds concepts incrementally. When mathematical background is needed, we provide intuition before formalism.

## 6. Companion Resources

This guide is part of a comprehensive ecosystem:

- **bdd-rs library**: Open-source Rust implementation at [github.com/Lipen/bdd-rs](https://github.com/Lipen/bdd-rs).
- **Code examples**: All examples from this guide are in the [bdd-rs/examples/abstract-interpretation/](https://github.com/Lipen/bdd-rs/tree/main/examples/abstract-interpretation/) directory.
- **Test suite**: Comprehensive tests validate correctness of implementations.

All code is open-source and contributions are welcome.

## 7. Acknowledgments

This guide builds on decades of foundational research. Abstract interpretation was pioneered by Patrick and Radhia Cousot in their seminal work. Binary Decision Diagrams were introduced by Randal Bryant, revolutionizing symbolic verification. Countless researchers have extended these foundations — their work is cited throughout.

The [bdd-rs](https://github.com/Lipen/bdd-rs) library represents a modern synthesis of these ideas, optimized for Rust's ownership model and designed for practical deployment.

Thank you to the open-source community for feedback, bug reports, and contributions that have improved both the library and this guide.

## 8. Let's Begin

Program verification is not just an academic exercise — it's a necessity for building reliable, secure systems. Testing alone cannot guarantee correctness in modern software. We need tools that reason about **all** possible behaviors, not just sampled executions.

This guide equips you with the theory and practice of BDD-guided abstract interpretation:

- **Theory**: Understand why these techniques work and when they apply.
- **Practice**: Implement analyses using the `bdd-rs` library.
- **Applications**: Apply these tools to real security and verification problems.

Whether you're here to understand how verification works, to apply these techniques to your own code, or to advance the state of the art, this guide aims to be your companion.

Let's dive in!

# Contents

Preface .....	i
Part I: Gentle Introduction .....	1
Prologue: The Case for Static Analysis .....	2
0.1 The Illusion of Testing .....	2
0.2 The Abstract Approach .....	3
0.3 The Promise of Formal Methods .....	4
0.4 The Toolsmith’s Craft .....	5
1 Foundations of Abstraction .....	6
1.1 The Geometric Analogy .....	6
1.2 The Subject of Analysis: IMP .....	7
1.3 Designing an Abstract Domain .....	7
1.4 Formalizing Abstraction .....	8
1.5 Interactive Reasoning .....	8
1.6 The Challenge of Control Flow .....	9
2 Control Flow and Program Structure .....	12
2.1 From AST to CFG .....	12
2.2 Translating AST to CFG .....	13
2.3 The Path Explosion Problem .....	14
2.4 The Solution: Symbolic Representation .....	16
3 Symbolic Reasoning with BDDs .....	18
3.1 From Sets to Functions .....	18
3.2 Formal Definition of BDDs .....	19
3.3 ROBDD Invariants .....	20
3.4 Visualizing Reduction .....	21
3.5 Complexity of Core Operations .....	22
3.6 Research Spotlight: Variable Ordering Heuristics .....	23
3.7 Exercises .....	24
3.8 The <code>SymbolicManager</code> .....	24
3.9 Why BDDs Solve State Explosion .....	26
4 Implementing the Engine: The <code>AnalysisManager</code> .....	28
4.1 Setting Up the Project .....	28
4.2 The <code>bdd-rs</code> Crash Course .....	28
4.3 Defining the Input Language .....	30
4.4 Designing the <code>AnalysisManager</code> .....	30
4.5 Allocating Conditions .....	31
4.6 Exposing BDD Operations .....	32
4.7 Debugging with Graphviz .....	32
4.8 Putting It Together .....	32
4.9 Manager Internals Deep Dive .....	34
5 The Abstract Program State .....	39

5.1	The Core Idea: Trace Partitioning .....	39
5.2	Architecture .....	40
5.3	Implementation: PathSensitiveState .....	41
5.4	Advanced: Condition Management .....	42
5.5	Alternative Design: Partitioning .....	44
5.6	Refining Abstract Values: Bidirectional Flow .....	46
5.7	Putting It Together: The Interpreter Loop .....	47
5.8	Understanding the Product: How BDDs and Domains Cooperate .....	48
5.9	Managing Partition Growth .....	49
5.10	Complete Example: Temperature Controller Analysis .....	50
5.11	Beyond Single Domains: Combining Multiple Abstractions .....	52
5.12	Relational Domains and BDD Synergy .....	53
5.13	Research Spotlight: Trace Partitioning in Context .....	54
5.14	Engineering Perspective: Implementation Trade-offs .....	55
5.15	Summary .....	56
6	A Complete Example: Symbolic Execution Engine .....	58
6.1	What is Symbolic Execution? .....	58
6.2	Architecture Overview .....	59
6.3	Condition Language .....	59
6.4	Symbolic State .....	60
6.5	Modifications and Actions .....	60
6.6	Branching .....	61
6.7	Program Representation .....	61
6.8	Program Walker .....	62
6.9	Conflict Detection .....	63
6.10	Complete Example: Simple Program .....	63
6.11	Enhancements for Real Systems .....	64
6.12	Practical Considerations .....	65
6.13	Real-World Applications .....	66
6.14	Summary .....	66
	Part II: Deep Dive .....	68
7	Lattice Theory Foundations .....	69
7.1	Partial Orders and Lattices .....	69
7.2	Height and Chains .....	73
7.3	Monotone Functions .....	74
7.4	Fixpoints and Tarski's Theorem .....	75
7.5	Kleene Fixpoint Theorem and Iteration .....	76
7.6	Galois Connections .....	77
7.7	The Lattice of Boolean Functions .....	80
8	Fixpoint Algorithms .....	82
8.1	From Kleene to Chaotic Iteration .....	82
8.2	Worklist Algorithms .....	83
8.3	Iteration Strategies .....	85
8.4	Complexity Analysis .....	86

8.5	Widening with Worklist .....	87
8.6	Narrowing Iterations .....	88
8.7	Delayed Widening .....	89
9	Advanced Galois Connections .....	91
9.1	From Concrete to Abstract Transformers .....	91
9.2	Best Abstract Transformers .....	92
9.3	Completeness of Abstract Transformers .....	93
9.4	Combining Domains .....	94
10	The Theory of Approximation .....	96
10.1	Widening Operators .....	96
10.2	Narrowing Operators .....	97
10.3	The Analysis Loop Pattern .....	97
10.4	Designing Widening Operators .....	98
11	Algebraic Domain Combinations .....	99
11.1	The Direct Product .....	99
11.2	The Reduced Product .....	101
11.3	Trace Partitioning .....	105
11.4	Abstract Transformers for Products .....	106
11.5	Relational Domains .....	107
11.6	Precision vs. Cost Tradeoffs .....	107
11.7	Widening in Product Domains .....	108
12	BDD Path Sensitivity .....	110
12.1	The BDD Product Domain .....	110
12.2	Implementing Lattice Operations .....	110
12.3	The Transfer Function: Assume & Filter .....	111
12.4	Reduction: The “Killer” Interaction .....	112
12.5	Variable Ordering and Performance .....	112
12.6	Case Study: Array Access Safety .....	112
12.7	Performance Considerations .....	113
13	String and Automata Domains .....	116
13.1	Motivation and Threat Model .....	116
13.2	Scalar String Properties: Length and Charset .....	116
13.3	Regular-Language Abstraction via Automata .....	117
13.4	Transformers for Common Operations .....	117
13.5	Reduced Products and Cooperation .....	118
13.6	Validation and Normalization Pipelines .....	118
13.7	Widening/Narrowing Patterns .....	118
14	Points-to and Dynamic Type Domains .....	120
14.1	Program Model and Sensitivities .....	120
14.2	May/Must Aliasing and Allocation Sites .....	120
14.3	Dynamic Type Domain .....	121
14.4	Sound Transfer Functions .....	121
14.5	Cooperation with Numeric and Control Domains .....	122
14.6	Precision vs. Performance: What to Bound .....	122



14.7	Worked Micro-Example: Object Aliasing .....	122
14.8	Related Sensitivities and Variants .....	123
15	Precision Techniques and Design Patterns .....	125
15.1	The Power of Combination: Reduced Products .....	125
15.2	Divide and Conquer: State Partitioning .....	127
15.3	Accelerating Convergence: Widening and Narrowing .....	129
15.4	Engineering Heuristics .....	130
15.5	User Experience: Explaining the Result .....	131
	Part III: Applications & Future Directions .....	135
16	Security Analysis .....	136
16.1	Input Taint Analysis .....	136
16.2	BDD-Guided Taint Analysis .....	136
16.3	Implicit Flows and Side Channels .....	137
16.4	Implementation Strategy .....	138
17	Inter-Procedural Analysis .....	139
17.1	Call Graph and Summaries .....	139
17.2	The Challenge of Summaries with BDDs .....	139
17.3	Call-Strings (k-limited) .....	140
17.4	Handling Loops and Recursion .....	140
17.5	Modular vs Whole-Program Analysis .....	141
18	Performance & Debugging .....	142
18.1	The Three Pillars of BDD Performance .....	142
18.2	Debugging Techniques .....	143
18.3	Profiling .....	144
18.4	Tuning Widening .....	144
19	AI-Guided Analysis .....	146
19.1	The Wizard and the Clerk .....	146
19.2	Invariant Synthesis .....	147
19.3	Widening Oracles .....	147
19.4	Future Directions .....	147
20	Case Study: Access Control System .....	148
20.1	Problem Statement .....	148
20.2	BDD Encoding .....	148
20.3	Walkthrough .....	149
20.4	Detecting Misconfigurations .....	149
20.5	Performance Analysis .....	150
20.6	Code Snippet .....	150
21	Conclusion & Further Reading .....	151
21.1	The Big Picture .....	151
21.2	What We Built .....	151
21.3	Further Reading .....	151
21.4	Final Words .....	152

# Part I

## Gentle Introduction

This part starts from first principles, building intuition about program abstraction, control flow, and symbolic representations. Through running examples like heater controllers and traffic lights, we motivate why BDD-based path-sensitive analysis matters. This part is accessible to anyone with programming background.

# Prologue: The Case for Static Analysis



Civilization runs on abstraction. We navigate cities with maps, not by memorizing every brick. We predict weather with pressure and temperature, not by tracking every air molecule. To understand complexity, we must focus on structure and ignore noise.

Yet in software, we often abandon this wisdom. We treat programs as concrete machines to operate, not mathematical objects to understand. We rely on testing — running code with specific inputs — to guess at behavior. But modern software is extraordinarily complex. It has zero physical tolerance: one misplaced character in a smart contract or avionics controller can cause catastrophe.

As systems grow from thousands to millions of lines, concrete approaches collapse. Complexity is the silent killer. To tame it, we must rediscover abstraction. We need tools that reason about **infinite** program behaviors using **finite** representations. We need tools that don't just run code, but **understand** it.

## 0.1 The Illusion of Testing

Testing runs code and observes output to check correctness. But testing is fundamentally optimistic. It assumes the future resembles the past, and that chosen inputs represent reality. For modern software, this assumption fails under the scale of state space.

Consider a simple function that takes two 64-bit integers:

```
fn process(a: u64, b: u64) { ... }
```

How many possible input pairs exist? Each variable has  $2^{64}$  possibilities. Together, they form a state space of  $2^{128}$  combinations. That is approximately  $3.4 \times 10^{38}$  states.

To put this in perspective: if you had a supercomputer checking one trillion inputs per second, it would take **10 trillion trillion years** to test this single function exhaustively. The universe is only 13.8 billion years old.

But the problem runs deeper than size. Software is **discontinuous**. In physics, nature is smooth: a bridge that holds 10 tons likely holds 10.1 tons. In software, this intuition breaks. Specific values like `0`, `u64::MAX`, or particular bit patterns can trigger entirely different code paths. Testing `x = 5` reveals nothing about `x = 0`.

## 0.2 The Abstract Approach

We must stop simulating the machine point by point. We need to reason about **all** behaviors at once.

Testing is like fishing with a spear: you catch one input at a time. **Abstract Interpretation** is like fishing with a net: you capture entire regions of state space in one pass.

Instead of asking “What does the program do for input  $A$ ?”, we ask “What does it do for **all positive integers**?”. We replace precise concrete values with **abstract properties**.

1. Concrete:  $x = 5, y = 10$
2. Abstract:  $x > 0, y > 0$

If we know  $x$  is positive and  $y$  is positive, we know  $x + y$  is positive. We have proven a property for **infinite** concrete cases with a single abstract rule. We lose exact values but preserve safety properties (no overflow, no negative result).

Patrick and Radhia Cousot formalized this approach in 1977. Their insight was revolutionary: programs can be studied by approximating semantics in a mathematically controlled way. This unified dataflow analysis, type checking, and interval analysis into one rigorous framework.

### 0.2.1 The Art of Approximation

Consider: is  $x * x$  always non-negative? Proving this by testing requires checking every number from **Neg** infinity to **Pos** infinity. Proving this by abstract interpretation simply observes the rule of signs. We map infinite integers to finite abstract values:

Abstract Value	Meaning	Examples
<b>Pos</b>	Positive integers	1, 42, 5000
<b>Zero</b>	Zero	0
<b>Neg</b>	Negative integers	-1, -7, -100
<b>Top</b>	Unknown	Any integer

Table 1: The Sign Abstract Domain

By defining operations on these values, we can prove properties without running the code. Here is how we might express this in Rust:

```
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
enum Sign { Neg, Zero, Pos, Top }

impl Sign {
    fn mul(self, other: Sign) → Sign {
        use Sign::*;
        match (self, other) {
            (Zero, _) | (_, Zero) ⇒ Zero,
            (Pos, Pos) | (Neg, Neg) ⇒ Pos,
```

```

        (Pos, Neg) | (Neg, Pos) => Neg,
        - => Top,
    }
}

```

With this small abstraction, we can prove that `x * x` is always `Pos` or `Zero` (non-negative), covering infinite inputs instantly.

## 0.2.2 The Machinery: Lattices and Fixpoints

Two mathematical concepts make this rigorous: **Lattices** and **Fixpoints**.

A **Lattice** provides a standard way to combine information. If one code path says `x` is `Pos` and another says `x` is `Zero`, the lattice tells us how to merge these facts (into `NonNeg`). It defines precision ordering: some abstract values are more precise than others.

A **Fixpoint** is the stable state of analysis. When analyzing loops, we cannot unroll them infinitely. Instead, we iterate until abstract facts stop changing. Because our lattices have finite height, we reach this stable state in finite time.

This approach trades discrete for continuous:

1. A **state** is a point in 256-dimensional hyperspace.
2. A **property** (like “no overflow”) is a region in this space.
3. The **program** is a trajectory moving points through space.

Verification is no longer simulation. It is geometry. To prove safety, we ask: “Do reachable states intersect error states?” Converting logic to geometry lets us answer this definitively for **all** inputs.

## 0.3 The Promise of Formal Methods

Why does this matter? Because failure costs are no longer just crashed apps. They are crashed economies or lost lives. Formal methods, once purely academic, are now essential for securing our digital world.

1. **Hardware Verification**: Intel and AMD use these techniques to ensure CPUs add numbers correctly. A bug here means billion-dollar recalls.
2. **Smart Contracts**: Billions in DeFi are secured by symbolic analysis. Code is law here, and one reentrancy bug can drain a vault instantly.
3. **Safety-Critical Systems**: Mars Rover flight software and nuclear reactor control loops are verified, not just tested. Failures here are disasters. The Ariane 5 rocket exploded 40 seconds after launch from integer overflow. The Mars Climate Orbiter disintegrated from a unit mismatch. At 140 million miles from Earth, you cannot push a hotfix.

Mastering these tools means more than learning new algorithms. It means learning to tame the infinite.

## 0.4 The Toolsmith's Craft

This guide is for those who want to build such tools. We will not just study theory. We will forge a **Symbolic Analyzer** from scratch. Building precise, durable tools requires the right materials.

### 0.4.1 Why Rust?

Rust offers a combination rarely found in other languages: memory safety without garbage collection, an expressive type system, and algebraic data types. For abstract interpretation, this allows us to:

1. Prototype abstract domains using traits and enums.
2. Ensure memory safety in our analyzer without overhead.
3. Leverage a rich ecosystem of libraries.

### 0.4.2 Why BDDs?

Many program properties reduce to Boolean conditions (“Is this variable zero?”, “Is this path reachable?”). Binary Decision Diagrams provide compact, canonical representations of Boolean functions. They compress logic, letting us represent massive state spaces ( $2^{100}$  and beyond) in microseconds.

### 0.4.3 The Roadmap

By the end of this journey, you will have:

1. **Formal Foundations**: Complete lattices, Galois connections, and fixpoint theorems.
2. **Rust Implementations**: Abstract domains (intervals, signs), control-flow graphs, and fixpoint engines.
3. **Symbolic Analysis**: Using BDDs to encode complex logic and state spaces.
4. **Visualization**: Generating diagrams to see the geometry of your analysis.

You will see software not as a black box to poke and prod, but as a crystal structure to analyze and perfect.

Let us begin.

# Chapter 1

## Foundations of Abstraction



The Prologue showed we cannot test every possible input. We must reason about **sets** of inputs (abstract properties) rather than individual values. But how do we actually do this? How do we ensure approximation doesn't hide the bugs we seek?

This chapter introduces the core mechanism: **Abstract Domains**.

### 1.1 The Geometric Analogy

Imagine a cylinder floating in space. Describing its exact position and shape requires precise coordinates for every surface point. This is like a program's **concrete state**: complex, detailed (all variable values, memory, heap), hard to manipulate.

Now imagine shining light to cast shadows on walls.

- From the top, the shadow is a **circle**.
- From the side, the shadow is a **rectangle**.



#### Intuition

Neither shadow captures the full object. However, both shadows provide **sound constraints**.

- If the circular shadow has a diameter of 10cm, we know for a fact that the object fits within a 10cm width.
- If the rectangular shadow has a height of 20cm, we know the object is no taller than 20cm.

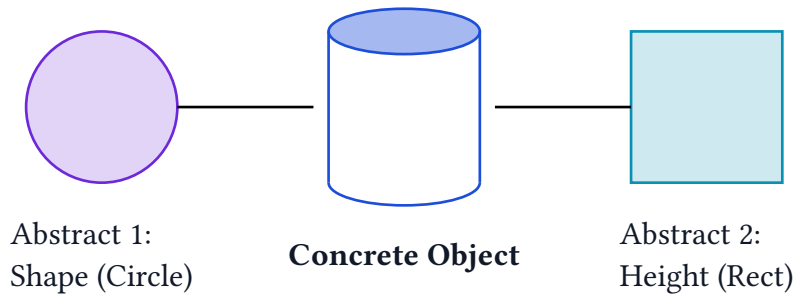


Figure 1: Abstraction as Projection. The concrete object (cylinder) is complex. Its shadows (circle, rectangle) are simple abstractions. Each shadow captures some truth but loses other details.

Abstract Interpretation is choosing the right “projection” (abstraction) for the property we want to prove.

- To prove a variable is positive, project to its **Sign**.
- To prove a variable is **Even**, project to its **Parity**.
- To prove a variable is within a range, project to an **Interval**.

## 1.2 The Subject of Analysis: IMP

We need a concrete subject to analyze. Throughout this guide, we build a **Static Analyzer** for a toy language called **IMP** (Minimal Imperative Language).

IMP is standard in verification textbooks. It supports:

- **Variables**: Integers (`x`, `y`, `z`).
- **Arithmetic**: `+`, `-`, `*`, `/`.
- **Control Flow**: `if-else`, `while`.
- **Assertions**: `assert(condition)`.

Here is a simple IMP program:

```
// Example IMP program
x = input();
if x > 0 {
    y = x + 1;
} else {
    y = 0;
}
assert(y > 0);
```

Our goal: verify properties like “Is the assertion always true?” or “Can `y` ever be negative?”.

## 1.3 Designing an Abstract Domain

We cannot simulate every possible integer input. Instead, we design an **Abstract Domain** capturing the properties we care about.

Focus on variable **sign**. Concrete variables hold specific integers (e.g., `5`, `-42`, `0`). For analysis, we often only care if a number is **Pos**, **Neg**, or **Zero**.



We define a set of abstract values  $D$ :

$$D = \{\perp, \text{Neg}, \text{Zero}, \text{Pos}, \top\} \quad (1)$$

- $\perp$  (Bottom): Represents the **empty set** (impossible / dead code).
- **Neg**: Represents the set of all **Neg** integers.
- **Zero**: Represents the singleton set  $\{0\}$ .
- **Pos**: Represents the set of all **Pos** integers.
- $\top$  (Top): Represents the **universal set** (unknown / any integer).

We can implement this in Rust:



### Hands-On Example

Complete implementation of the Sign domain with lattice operations. Shows how abstraction trades precision for tractability.

Source: `sign.rs`  
 ▶ Run: `cargo run --example sign_domain`

## 1.4 Formalizing Abstraction

With a domain defined, we can be rigorous. We define two functions connecting the concrete world (actual execution) and abstract world (properties).

**Definition 1 (Concretization Function)** The concretization function  $\gamma : D \rightarrow \mathcal{P}(\mathbb{Z})$  maps an abstract value to a set of concrete integers  $\mathbb{Z}$ . For our Sign domain:

$$\begin{aligned} \gamma(\text{Pos}) &= \{z \in \mathbb{Z} \mid z > 0\} \\ \gamma(\text{Neg}) &= \{z \in \mathbb{Z} \mid z < 0\} \\ \gamma(\text{Zero}) &= \{0\} \\ \gamma(\perp) &= \emptyset \\ \gamma(\top) &= \mathbb{Z} \end{aligned} \quad (2)$$

Analysis is **sound** if abstract results always cover concrete results. If a variable is actually **5**, analysis must return either **Pos** or  $\top$ , never **Neg**.

## 1.5 Interactive Reasoning

Let's build intuition by playing a game using this domain. I have two hidden program states (variables),  $A$  and  $B$ . I won't tell you their exact values, but I will tell you their **abstract signs**.

### 1.5.1 First Scenario: Both Pos

Suppose:

- $A$  has sign Pos
- $B$  has sign Pos

Question: If I pick one variable at random, what is its sign?

The answer is Pos. Reasoning: The union of two Pos value sets is still Pos. Both possibilities lead to Pos values, so we can be certain.

### 1.5.2 Second Scenario: Mixed Signs

Now suppose:

- $A$  has sign Pos
- $B$  has sign Neg

Question: If I pick one variable at random, what is its sign?

The answer is  $\top$  (Unknown). Here's why:

- Could be Pos or Neg.
- Domain  $D$  lacks a “Non-Zero” value.
- Smallest value covering both is  $\top$ .

Since the result depends on **which path** was taken, we cannot give a precise single sign. We return  $\top$  to remain sound.

### 1.5.3 Abstract Semantics

We define **abstract transfer functions** to execute code in this domain. For merging two paths:

```
impl AbstractDomain for Sign {
  fn join(&self, other: &Self) → Self {
    match (self, other) {
      (Sign::Pos, Sign::Pos) ⇒ Sign::Pos,
      (Sign::Neg, Sign::Neg) ⇒ Sign::Neg,
      (Sign::Zero, Sign::Zero) ⇒ Sign::Zero,
      (Sign::Bot, x) ⇒ x.clone(),
      (x, Sign::Bot) ⇒ x.clone(),
      // ... merging different signs returns Top
      _ ⇒ Sign::Top,
    }
  }
}
```

## 1.6 The Challenge of Control Flow

Straight-line code is easy. Branches are hard.

```
if input > 0 {
  x = 1; // x is "Pos"
```

```

} else {
    x = -1; // x is "Neg"
}
// At this point, what is x?

```

At the merge point, `x` could be `1` OR `-1`. In the Sign domain, we must find one value covering **both** possibilities. The smallest value covering both `Pos` and `Neg` is  $\top$ .

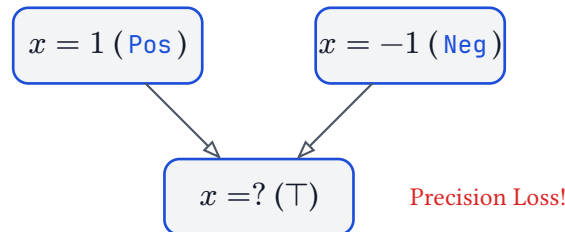


Figure 2: The Merge Problem. Merging two precise paths often results in information loss ( $\top$ ).

We lost the information that `x` is non-zero! We know it’s one of the two, but abstraction forces us to say “could be anything.”

This is where **BDDs** enter. Instead of merging everything into one abstract value (getting  $\top$ ), BDDs track **which path** leads to which value.

- Path 1 (`input > 0`): `x` is `Pos`
- Path 2 (`input ≤ 0`): `x` is `Neg`

This is called **Path Sensitivity**, and it is the main focus of this guide.

## Chapter Summary

This chapter established the foundational paradigm of abstract interpretation through concrete examples.

**Abstraction is projection** — like a shadow of a three-dimensional object, an abstract domain captures essential properties while intentionally discarding others. The choice of what to track (sign, interval, exact value) fundamentally determines analysis capability and cost.

**Soundness is the non-negotiable guarantee**: when the abstraction claims “safe”, the concrete program must indeed be safe. The abstraction may answer “unknown” for programs that are actually safe, but it must never claim safety for unsafe programs. This one-directional guarantee enables using abstract interpretation for verification rather than mere bug finding.

The **precision-versus-speed tradeoff** pervades all analysis design. Richer domains like intervals provide more detailed information than simple signs, but require more

computation per operation. Understanding this tradeoff guides domain selection for specific verification goals.

Finally, **the merge problem** reveals abstraction's fundamental challenge: when execution paths reconverge, we must combine possibly conflicting information. Naive joining often loses precision, motivating the path-sensitive techniques we'll explore in subsequent chapters.

# Chapter 2

## Control Flow and Program Structure

The previous chapter defined our Toy Language (IMP) using an Abstract Syntax Tree (AST). ASTs excel at representing code **structure** (how it's written), but analyzing **behavior** (how it executes) is awkward.

To analyze a program, we need to see it not as a hierarchy of rules, but as a network of execution paths. This representation is the **Control Flow Graph (CFG)**.

**Key Insight Analogy:** Think of an AST as a **Rule Book** — it lists regulations chapter by chapter. A CFG is like a **Traffic Map** — it shows how execution actually moves through the intersections and roads.

### 2.1 From AST to CFG

ASTs are recursive and hierarchical. Program execution is sequential (instruction-by-instruction) and sometimes jumps between blocks. A CFG “flattens” recursive structure into a graph.

**Definition 2 (Control Flow Graph)** A **Control Flow Graph** is a directed graph  $G = (V, E, v_{\text{entry}})$  where:

- $V$  is the set of **Basic Blocks** (nodes).
- $E \subseteq V \times V$  is the set of control flow transitions (edges).
- $v_{\text{entry}} \in V$  is the unique entry point of the program.

#### 2.1.1 The Basic Block

The fundamental unit of a CFG is the **Basic Block**.

**Definition 3 (Basic Block)** A **Basic Block** is a maximal sequence of instructions with:

1. **Single Entry:** Execution can only enter at the first instruction (the “leader”).
2. **Single Exit:** Execution can only leave from the last instruction (the “terminator”).
3. **Atomic Execution:** If the first instruction executes, all subsequent instructions in the block are guaranteed to execute (unless a crash occurs).

## 2.1.2 Implementing the CFG

In our Rust implementation, we represent the CFG as a collection of blocks, identified by unique integers (`BlockId`).

**Why not analyze the AST directly?** ASTs are trees, but control flow is a graph. Programs often have “jumps” (e.g., `goto`, `break`, `continue`). In an AST, the target is a separate subtree. In a CFG, it is just an edge to the target block. This “flattening” makes analysis much simpler, especially for loops.

```
type BlockId = usize;

#[derive(Clone, Debug)]
pub struct BasicBlock {
    pub id: BlockId,
    pub instructions: Vec<Instruction>, // The straight-line code
    pub terminator: Terminator,        // How we leave the block
}

#[derive(Clone, Debug)]
pub enum Terminator {
    // Unconditional jump to another block
    Jump(BlockId),
    // Conditional branch based on a condition
    Branch {
        condition: Match, // Uses the Match type from our AST
        true_target: BlockId,
        false_target: BlockId,
    },
    // Final verdict
    Return(Verdict),
}
```



### Hands-On Example

Implementation of a CFG builder that transforms a linear sequence of instructions into a graph of basic blocks. Shows how to handle labels, jumps, and branches.

**Source:** `cfg_builder.rs`  
**Run:** `cargo run --example cfg_builder`

## 2.2 Translating AST to CFG

The translation process involves breaking down complex AST nodes (like `if-else` trees) into simple blocks and edges. Let’s visualize how each structure is transformed.

### 2.2.1 Sequence (`s1; s2`)

Sequences are concatenated. If `s1` is just a modification, `s2` appends to it. If `s1` contains control flow, it ends the current block and `s2` starts a new one.

### 2.2.2 Branch (`if m { t } else { e }`)

A branch splits the execution flow into two paths that eventually merge.

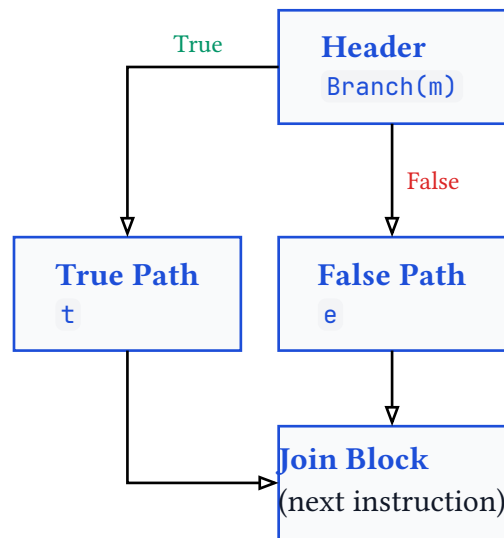


Figure 3: Translation of `if m { t } else { e }` into CFG blocks.

### 2.2.3 Loops

While our toy language PFL is loop-free, general programs have loops. We need a “Header” block to evaluate the condition each time execution loops.

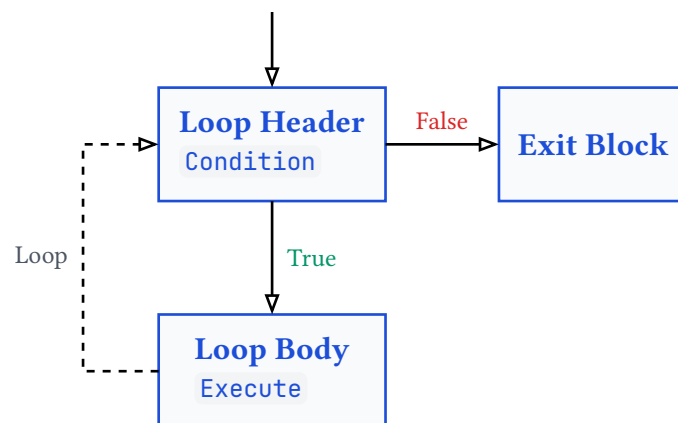


Figure 4: Translation of a `while` loop into CFG blocks.

## 2.3 The Path Explosion Problem

Why all this trouble? Why is program verification so hard?

The answer lies in CFG structure. Every branch multiplies the number of possible execution paths.

Consider a sequence of just 3 independent checks in IMP:

```

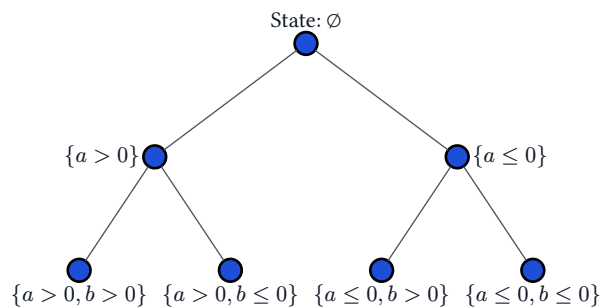
if a > 0 { ... }
if b > 0 { ... }
if c > 0 { ... }

```

This creates  $|\Pi| = 2 \times 2 \times 2 = 8$  paths. For a sequence of  $N$  branches, the size of the path set  $\Pi$  grows exponentially:

$$|\Pi| = 2^N \quad (3)$$

This is **exponential growth**.



**$2^N$  Paths!**

Figure 5: Visualizing Path Explosion. As execution branches, the number of distinct states we must track doubles at each step.

Analyzing a loop by unrolling it (simulating each iteration) treats 100 iterations like 100 nested `if` statements. Path count becomes astronomical ( $2^{100}$ ).

### 2.3.1 Path Sensitivity vs. Scalability

We want **path-sensitive** analysis — analysis that distinguishes between different execution histories.

### 2.3.2 Path-Insensitive Analysis

Path-insensitive analysis merges all incoming paths at join points. At any program point, it can only say “variable `x` could be anything.” This approach is fast because it only tracks one abstract state per program location. However, it loses precision by forgetting which path was taken.

### 2.3.3 Path-Sensitive Analysis

Path-sensitive analysis maintains separate information for each path. It can make precise statements like “if we took the true branch, `x` is **Pos**; if we took the false branch, `x` is **Neg**.” This provides much better precision, enabling detection of subtle bugs. The cost is potentially exponential state growth as paths multiply.



**The Dilemma** We want the precision of path sensitivity without the cost of enumerating exponential paths. Naive enumeration is impossible. Naive merging is imprecise.

## 2.4 The Solution: Symbolic Representation

We need a “Third Way.” We need a data structure representing **sets of states** compactly, without listing them individually.



### Intuition

**Extensional vs. Intensional Definition** There are two ways to define a set  $S$  within a universe  $\mathcal{U}$ :

- **Extensional:** Listing every member explicitly.

$$S = \{s_1, s_2, \dots, s_n\} \quad (4)$$

This is computationally infeasible when  $|S|$  is large (e.g.,  $|\mathcal{U}| \approx 2^{256}$ ).

- **Intensional:** Specifying a logical predicate  $P$  characterizing the set.

$$S = \{x \in \mathcal{U} \mid P(x)\} \quad (5)$$

Symbolic execution uses the **intensional** approach. A BDD encodes the **characteristic function**  $\chi_S : \mathcal{U} \rightarrow \{0, 1\}$ :

$$\chi_S(x) = \begin{cases} 1 & \text{if } P(x) \text{ is true} \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

This lets us manipulate the **logic** of the set (function  $\chi_S$ ) rather than enumerating elements.

In this guide, we use **Binary Decision Diagrams (BDDs)** as our symbolic representation. BDDs let us:

1. Encode program logic.
2. Represent huge state sets efficiently.
3. Perform operations (join, intersection) on these sets mathematically.

Next chapter: we dive into how BDDs work their magic.

### Chapter Summary

This chapter dissected program structure to reveal both the analysis challenge and its solution.

**Control Flow Graphs provide the analysis substrate** by transforming hierarchical program syntax into a flat graph structure. Each node represents a basic block —

a maximal sequence of statements that always execute together atomically, with control flow decisions occurring only at block boundaries. This representation exposes program execution topology explicitly, enabling systematic traversal and analysis.

The **path explosion problem** emerges as program analysis's central computational challenge. With  $n$  independent conditionals, a program contains  $2^n$  distinct execution paths. Loops introduce infinite path families, making exhaustive enumeration fundamentally intractable.

**Path-insensitive analysis** addresses this by merging states at control flow join points, sacrificing precision for tractability. While sufficient for many properties, it loses the ability to prove properties that depend on specific execution sequences.

**Symbolic execution with BDDs** provides the breakthrough: representing exponentially many paths implicitly through symbolic constraints. Rather than enumerating paths individually, we manipulate entire path sets using polynomial-time Boolean operations. This combination of precision and efficiency enables practical path-sensitive analysis.

# Chapter 3

## Symbolic Reasoning with BDDs

Section 2 showed that tracking every execution path individually leads to **path explosion**.

To build a scalable verifier, we need a mechanism representing **sets of states** efficiently.

This chapter introduces the **Binary Decision Diagram (BDD)**, the core data structure powering our Symbolic Analyzer.

Instead of enumerating states as lists, we represent them as **Boolean functions**.

### 3.1 From Sets to Functions

A fundamental insight: sets and functions correspond.

We represent any subset  $S$  of universe  $U$  using a **characteristic function**  $f_S : U \rightarrow \{0, 1\}$ :

$$f_S(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{if } x \notin S \end{cases} \quad (7)$$

If we can represent  $f_S$  compactly, we effectively represent the set  $S$  compactly.

#### 3.1.1 Constraints as Boolean Formulas

In a CFG, a path is determined by decisions made at branch points.

By assigning Boolean variables to branch conditions, we encode paths as logical formulas.

Consider the following IMP snippet:

```
if x > 0 { // Decision A
    // ...
} else {
    // ...
}
if y == 5 { // Decision B
    valid = true;
}
```

Let  $A$  represent the condition  $x > 0$  and  $B$  represent  $y = 5$ .

Each path through the program corresponds to a conjunction of these Boolean variables. We have four possible paths:

1. When both conditions are true ( $x > 0$  and  $y = 5$ ): represented by  $A \wedge B$
2. When the first is true but second is false ( $x > 0$  and  $y \neq 5$ ): represented by  $A \wedge \neg B$
3. When the first is false but second is true ( $x \leq 0$  and  $y = 5$ ): represented by  $\neg A \wedge B$
4. When both conditions are false ( $x \leq 0$  and  $y \neq 5$ ): represented by  $\neg A \wedge \neg B$

The set of all valid paths through the program is the disjunction (logical OR) of these four formulas.

To represent the set of states where `valid = true` (i.e., where decision  $B$  is true), we write:

$$(A \wedge B) \vee (\neg A \wedge B) \quad (8)$$

Using Boolean algebra, this simplifies to just  $B$ .

**Key Insight** Set operations on states correspond directly to Boolean operations on formulas:

- **Union** ( $\cup$ )  $\rightarrow$  Logical OR ( $\vee$ )
- **Intersection** ( $\cap$ )  $\rightarrow$  Logical AND ( $\wedge$ )
- **Empty Set** ( $\emptyset$ )  $\rightarrow$  False (0)
- **Universal Set** ( $U$ )  $\rightarrow$  True (1)

## 3.2 Formal Definition of BDDs

A **Binary Decision Diagram (BDD)** is a graph-based data structure representing Boolean function  $f : \{0, 1\}^n \rightarrow \{0, 1\}$ .

Construction relies on the **Shannon Expansion**.

**Definition 4 (Shannon Expansion)** Any Boolean function  $f(x_1, \dots, x_n)$  can be decomposed with respect to a variable  $x_i$ :

$$f = (x_i \wedge f_{x_i=1}) \vee (\neg x_i \wedge f_{x_i=0}) \quad (9)$$

where:

- $f_{x_i=1}$  is the function  $f$  with  $x_i$  set to True (Positive Cofactor).
- $f_{x_i=0}$  is the function  $f$  with  $x_i$  set to False (Negative Cofactor).

Recursively applying this expansion yields a **Decision Tree**.

Since trees grow exponentially with variable count ( $2^n$  leaves), we transform the tree into a **Directed Acyclic Graph (DAG)** for compactness.

### 3.2.1 Ordered Binary Decision Diagrams (OBDD)

A BDD is **Ordered** (OBDD) if variables appear in the same fixed order on all root-to-terminal paths.

For instance, given ordering  $A < B < C$ , every path tests  $A$  before  $B$ , and  $B$  before  $C$ .



### Common Pitfall

**Variable Ordering Matters!** The size of a BDD depends heavily on the variable order. A good order keeps related variables close together. A bad order can cause exponential blowup. For example, for the function  $(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee \dots$ , the order  $a_1, b_1, a_2, b_2, \dots$  is linear, while  $a_1, a_2, \dots, b_1, b_2, \dots$  is exponential.

## 3.2.2 Reduced BDDs

A BDD is **Reduced** (ROBDD) if it contains no redundant information.

We achieve this by repeatedly applying two reduction rules until the graph is minimal:

1. **Merge Isomorphic Nodes:** If two nodes  $u$  and  $v$  represent the same variable and have identical high and low children, they are equivalent. We keep one and redirect all edges pointing to the other.
2. **Eliminate Redundant Tests:** If a node  $u$  has identical high and low children ( $\text{high}(u) = \text{low}(u)$ ), the decision at  $u$  does not affect the outcome. We remove  $u$  and redirect incoming edges directly to its child.

**Canonicity Property** For a fixed variable ordering, the reduced OBDD for any Boolean function is **unique**.

This property is crucial for verification: checking the equivalence of two functions  $f \equiv g$  reduces to checking if their BDD root nodes are identical (pointer equality), which is an  $O(1)$  operation.

## 3.3 ROBDD Invariants

The reduced ordered BDD representation satisfies three core invariants that guarantee canonicity:

1. **Invariant 1 (Ordering):** Variables appear in strictly ascending order along every root-to-terminal path.
2. **Invariant 2 (No Redundant Tests):** No internal node has identical high and low children.
3. **Invariant 3 (No Duplicate Nodes):** No two distinct nodes share the same variable and identical child pair (“low”, “high”).

Together, these invariants ensure a one-to-one mapping between Boolean functions and their graph representation under a fixed order.

**Definition 5 (Canonicity of ROBDDs)** Let  $f$  be a Boolean function and let  $\pi$  be a fixed total order over its variables. There exists exactly one ROBDD  $B$  such that  $B$  represents  $f$  and respects  $\pi$ .



#### Intuition

The uniqueness result turns semantic equivalence checking into pointer equality, allowing  $O(1)$  functional equivalence tests and enabling aggressive memoization.

### 3.3.1 Practical Check List

Before committing a node to the unique table we must enforce:

- Ordering: Reject construction if a child violates variable order.
- Redundancy: Collapse a node whose children match.
- Duplication: Reuse existing canonical node if present.



#### Implementation Note

The `Bdd` manager uses a hash consing table keyed by `(var, low, high)`. After computing low and high references for an apply step, it calls an insertion routine returning an existing node reference when possible.

Before diving deeper into BDD theory, it's worth getting hands-on experience with basic BDD operations.

The following example demonstrates variable creation, boolean operations, and the canonicity property in action:



#### Hands-On Example

Hands-on introduction to BDD operations: creating variables, applying boolean operations (AND, OR, NOT, XOR), and observing canonicity. Perfect starting point for understanding BDDs practically.

Source: `basics.rs`  
 Run: `cargo run --example bdd_basics`

## 3.4 Visualizing Reduction

To illustrate these concepts, let us visualize the reduction of the decision tree for the function  $(A \wedge B) \vee C$ , using the variable ordering  $A < B < C$ .

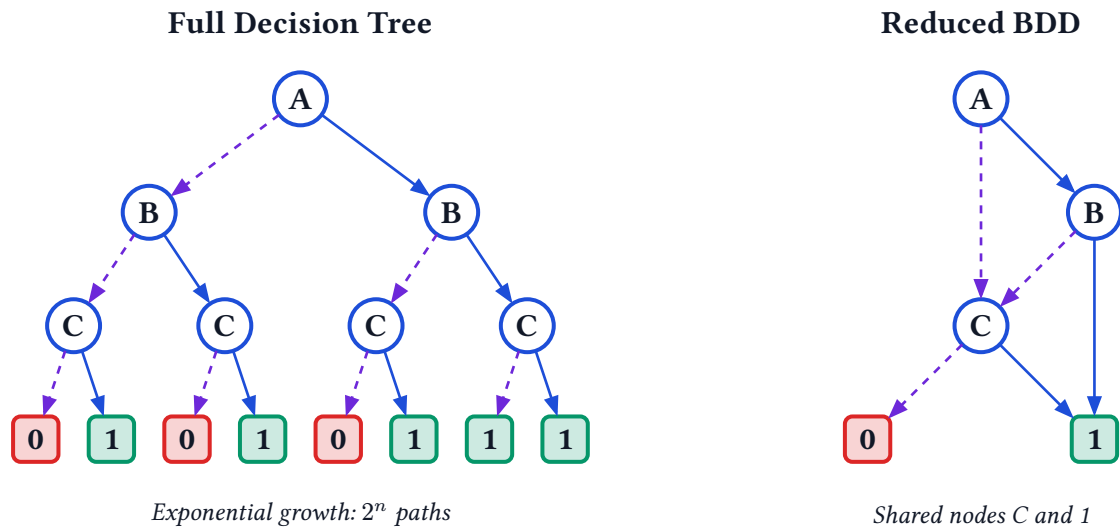


Figure 6: Comparison: Full Decision Tree vs. Reduced BDD for  $(A \wedge B) \vee C$

Figure 6 demonstrates the dramatic difference between the tree and graph representations:

- **Left (Decision Tree):** The tree explicitly enumerates all 8 paths. Note the redundancy: the subtrees for  $C$  are repeated multiple times.
- **Right (Reduced BDD):** The graph is significantly more compact due to reduction:
  - **Isomorphism:** The node  $C$  is shared by both branches of  $B$ .
  - **Redundant Test Elimination:** Notice the edge from  $A$  (low) directly to  $C$ . If  $A$  is false, the expression  $(A \wedge B) \vee C$  simplifies to  $(0 \wedge B) \vee C$ , which equals  $C$ . This means the value of  $B$  is irrelevant. The redundant  $B$  node is removed, and  $A$  connects directly to  $C$ .

## 3.5 Complexity of Core Operations

Operation cost depends on input BDD sizes and variable ordering quality, not raw path count. Let  $n$  be variable count and  $|B|$  denote node count.

Operation	Complexity
Apply (Binary Boolean)	$O( B_f  \times  B_g )$ worst case; typically near-linear with caching
Negation	$O( B )$ via complement edge toggling
Restriction / Cofactor	$O( B )$ traversing affected nodes
Quantification	Potentially $O( B  \times n)$ ; reduced by variable clustering
Variable Reordering	Heuristic sifting $O(n^2)$ swaps



### Common Pitfall

**Worst Case Alert:** Adversarial formulas destroy sharing and approach exponential size, e.g., integer multiplication bit level carry dependencies.



#### Implementation Note

Performance hotspots concentrate in apply and quantification routines. Instrument counters for cache hit ratio and node creation frequency to guide reordering heuristics.

## 3.6 Research Spotlight: Variable Ordering Heuristics

Variable ordering dominates BDD size, thus memory and time. Several approaches exist:

### 3.6.1 Static Heuristics

Apply domain-driven grouping once at initialization. For example, place related bitfields adjacent to each other. This leverages problem structure without runtime overhead.

### 3.6.2 Dynamic Sifting

Iteratively move a variable up and down through the ordering to minimize BDD size. At each position, measure the total node count and keep the best placement. This local optimization works well in practice.

### 3.6.3 Genetic and Annealing Approaches

Use global stochastic search to explore the space of variable permutations. Suitable for stable benchmarks where investment in finding good orderings pays off. These methods can escape local minima that trap greedy approaches.

### 3.6.4 Machine Learning Guidance

Extract features from BDD structure (node fanout, support size, *etc.*) and feed them to ranking models. The model predicts which variable swap is most promising. This combines the power of search with learned heuristics from previous problems.

#### R. E. Bryant (1986)

*Graph Based Algorithms for Boolean Function Manipulation*

Bryant's seminal work introduced reduced ordered BDDs and established foundational complexity tradeoffs still optimized with modern heuristics.



#### Implementation Note



Integrate a **lazy sifting trigger**: when node count crosses a threshold factor versus baseline, schedule a limited window reordering batch.

## 3.7 Exercises



### Exercise

*Easy*

Construct the full decision tree and reduced BDD for  $(A \vee B) \wedge (C \vee D)$  under order  $A < B < C < D$ . Count nodes and compare reduction ratio.



### Exercise

*Medium*

Show that two distinct Boolean formulas can share an identical ROBDD when variable ordering differs. Provide explicit counterexample and explain resolution.



### Exercise

*Medium*

Instrument the **apply** operation to record cache hits and misses for random 50 variable CNF instances. Plot hit ratio vs clause density.



### Exercise

*Hard*

Propose a scoring function for dynamic ordering decisions using features: variable level, node fanout, subtree size. Evaluate on synthetic benchmark set.



### Exercise

*Hard*

Implement existential quantification of a subset of variables and empirically measure node count delta for structured vs random formulas.

## 3.8 The SymbolicManager

In our Analyzer, we bridge the gap between the “semantic” AST world (nodes like `x > 0`) and the “numeric” BDD library world (integer variables like 1, 2, 3).

We implement a `SymbolicManager` to handle translation and ensure consistency.

**Design Pattern** The `SymbolicManager` acts as a facade over the raw BDD manager. It maintains a `HashMap<Condition, Ref>` to ensure that identical AST conditions always map to the same BDD variable.

The structure is defined as follows:

```
// Ensure Condition derives Hash and Eq!
#[derive(Clone, Debug, PartialEq, Eq, Hash)]
pub enum Condition {
    // ... variants ...
}

pub struct SymbolicManager {
    bdd: Bdd,
    mapping: HashMap<Condition, usize>, // Maps AST conditions to BDD variable IDs
    next_var_id: usize,
}

impl SymbolicManager {
    /// Get or create the BDD variable for a condition
    pub fn get_condition(&mut self, c: &Condition) → Ref {
        if let Some(&id) = self.mapping.get(c) {
            return self.bdd.mk_var(id as u32);
        }

        let id = self.next_var_id;
        self.next_var_id += 1;
        self.mapping.insert(c.clone(), id);
        self.bdd.mk_var(id as u32)
    }
}
```



### Intuition

**What is a Ref?** A `Ref` is just a lightweight integer handle (like a pointer or an ID). It has no meaning on its own. You must always pass it back to the `Bdd` manager to perform operations. Think of it like a “file descriptor” — you need the OS (Manager) to read the file.

The workflow:

1. When analyzer encounters a condition (e.g., `x > 0`) for the first time, manager allocates a new BDD variable (e.g., index 1) and stores the mapping.
2. When the same condition appears later, manager retrieves existing mapping and returns the same BDD variable.

This guarantees the **same** logical condition always gets the **same** BDD variable, preserving logical consistency.

## 3.9 Why BDDs Solve State Explosion

Section 2 showed:  $N$  branches can create  $2^N$  paths.

BDDs mitigate this by exploiting structure and independence.

When decisions don't interact (e.g., independent checks), BDD size grows **linearly** with variable count, not exponentially.

For example, consider 100 independent **if** statements: Consider the difference in representation size:

With explicit path enumeration, we would need to store  $2^{100}$  distinct paths. This number is astronomically large — more than the number of atoms in the observable universe. Storing this explicitly is computationally intractable.

With the BDD representation, we only need  $2 \times 100 = 200$  nodes. Each chain contributes exactly 2 nodes (one decision node pointing to True, one to False). This linear size makes the representation trivial to store and manipulate.

The BDD automatically factors out independence.

Explosion typically occurs only when variables are heavily correlated in complex ways (e.g., cryptographic hashes). This is less common in typical control logic.

### Chapter Summary

This chapter introduced Binary Decision Diagrams as the symbolic representation backbone for path-sensitive analysis.

The fundamental insight is that **sets of program states can be represented as Boolean functions**, mapping variable valuations to true (state included) or false (state excluded). **BDDs provide compact, canonical graph representations** of these functions through systematic sharing of common substructure.

Two **reduction rules** — eliminating isomorphic subgraphs and removing redundant tests — enable exponential compression while preserving canonicity. This means equivalent Boolean functions always produce identical BDD structures, making equality testing trivial.

The **SymbolicManager** bridges program syntax and BDD representation by mapping conditions to BDD variables consistently. This canonicalization ensures that identical program conditions always map to the same BDD variable, enabling efficient reuse and comparison.

With these foundations in place, we're ready to implement the concrete analysis infrastructure that leverages BDDs for symbolic program analysis.

**Explore BDD Operations** To see boolean operations in action, check out the example demonstrating AND, OR, XOR, and their algebraic properties like De Morgan's laws and absorption.



### Hands-On Example

Demonstration of standard boolean operations on BDDs. Includes verification of De Morgan's laws and other identities.



Source: `boolean_ops.rs`

► Run:

```
cargo run --example bdd_boolean_ops
```

# Chapter 4

## Implementing the Engine: The `AnalysisManager`

We have the theory: BDDs represent sets of program states. Now we build the engine.

Our first task is to implement the `AnalysisManager`. This component translates the language of our program conditions (strings like “`x > 0`”) into the language of BDDs (variables like 1, 2, 3).

### 4.1 Setting Up the Project

First, let's create a new Rust project for our IMP Analyzer.

```
cargo new imp-analyzer
cd imp-analyzer
# If using the local workspace:
# cargo add --path ../../bdd-rs
# If using crates.io (once published):
cargo add bdd-rs
```

We will use `bdd-rs` for all Boolean manipulation. This library is designed to be safe, fast, and easy to use, but it requires following a specific `manager-centric` design pattern.

### 4.2 The `bdd-rs` Crash Course

Before we build the manager, let's understand the tools we have. The `bdd-rs` library operates on a strict principle: **The Manager is King**.

**Why a Manager?** In a BDD, nodes are shared. If you have the condition `x > 0 AND y < 5` in two different parts of the program, they point to the **exact same memory address**.

To make this work, we need a central authority — the `Bdd` manager — to:

1. **Deduplicate**: Check if a node already exists before creating a new one (Hash Consing).
2. **Cache**: Remember the results of operations like `AND` and `OR` to avoid re-computing them (Computed Table).

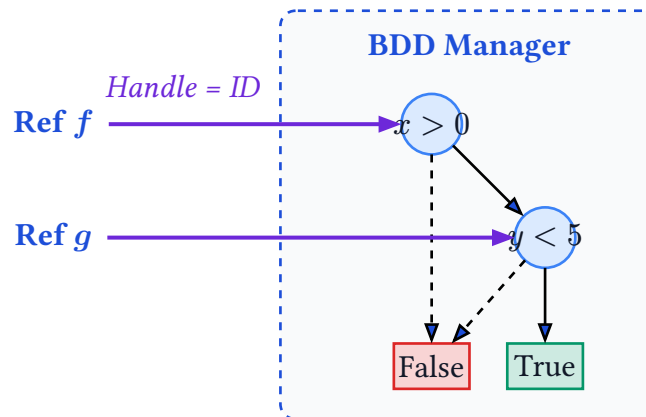


Figure 7: The Manager-Centric Model. The user holds `Ref` handles (integers), while the Manager stores the actual graph nodes. Multiple Refs can point to the same node.

Here is how you use it correctly:

```
use bdd_rs::bdd::Bdd; // Import the manager

fn main() {
    // 1. Initialize the manager
    let bdd = Bdd::default();

    // 2. Create variables (must be 1-indexed!)
    // Variable 0 is reserved for internal use.
    let x_gt_0 = bdd.mk_var(1);
    let y_lt_5 = bdd.mk_var(2);

    // 3. Combine them using the manager
    let both_true = bdd.apply_and(x_gt_0, y_lt_5); // (x > 0) AND (y < 5)
    let either_true = bdd.apply_or(x_gt_0, y_lt_5); // (x > 0) OR (y < 5)

    // 4. Check results
    println!("Both True: {:?}", both_true);
}
```

**Caution Important Invariant:** You never operate on `Ref` directly (e.g., `x.and(y)` is wrong). You always ask the manager to do it: `bdd.apply_and(x, y)`. The `Ref` is just a lightweight handle (a number); the Manager holds the actual graph. `Ref` implements `Copy`, so you can pass it around freely without worrying about ownership.

Understanding why this manager-centric design is essential requires looking at the internal mechanisms of hash consing and computed caches. The following example provides a deep dive into the manager's architecture:



### Hands-On Example

Deep dive into BDD manager architecture: hash consing, computed cache, and why all operations must go through the manager. Essential for understanding the performance characteristics of BDDs.

**Source:** `manager_demo.rs`  
**Run:** `cargo run --example bdd_manager`

## 4.3 Defining the Input Language

To build a verifier, we first need a language to verify. Let's define a minimal Abstract Syntax Tree (AST) for our IMP language conditions. This allows us to represent statements like `x > 0` or `y = 10` as data structures.

```
// src/ast.rs

#[derive(Debug, Clone, PartialEq, Eq, Hash)]
pub enum Var {
    X, Y, Z, // Simplified for example
    // In real code: Named(String)
}

#[derive(Debug, Clone, PartialEq, Eq, Hash)]
pub enum Op {
    Eq, // =
    Gt, // >
    Lt, // <
}

#[derive(Debug, Clone, PartialEq, Eq, Hash)]
pub struct Condition {
    pub var: Var,
    pub op: Op,
    pub val: i32,
}
```

## 4.4 Designing the AnalysisManager

Now, let's build our bridge. The BDD engine doesn't understand variables or integers. It only understands boolean variables 1, 2, 3, .... We need a component that maps our rich AST conditions (like `x > 0`) to these simple BDD variables.

We need a struct that holds:

1. The `Bdd` manager itself.
2. A mapping from `Condition` (our AST node) to BDD variable IDs.
3. A counter to assign new IDs.

```

use std::collections::HashMap;
use bdd_rs::bdd::Bdd;
use bdd_rs::reference::Ref;
// Assuming Condition is defined as in Chapter 1
use crate::ast::Condition;

pub struct AnalysisManager {
    bdd: Bdd,
    mapping: HashMap<Condition, usize>,
    next_var_id: usize,
}

impl AnalysisManager {
    pub fn new() → Self {
        Self {
            bdd: Bdd::default(), // Use default configuration
            mapping: HashMap::new(),
            next_var_id: 1, // Start at 1! 0 is reserved.
        }
    }
}

```

## 4.5 Allocating Conditions

`get_condition_var` takes a `Condition` and returns a BDD `Ref`. If we've seen this condition before, return the existing variable. Otherwise, allocate a new one.

```

impl AnalysisManager {
    pub fn get_condition_var(&mut self, c: &Condition) → Ref {
        if let Some(&id) = self.mapping.get(c) {
            // We've seen this condition before.
            // Return the BDD variable for it.
            return self.bdd.mk_var(id as u32);
        }

        // New condition!
        let id = self.next_var_id;
        self.next_var_id += 1;

        self.mapping.insert(c.clone(), id);
        self.bdd.mk_var(id as u32)
    }
}

```

**The Boolean Abstraction Gap** Our simple manager treats each `Condition` as an independent boolean variable.

If we encounter `x > 0` and `x > 5`, they get separate variables 1 and 2. The BDD allows  $1 \wedge 2$  (fine),  $1 \wedge \neg 2$  (fine), but also  $\neg 1 \wedge 2$  (impossible: `x > 5` implies `x > 0`).

This is **Boolean Abstraction**: we lose semantic relationships between arithmetic constraints. Fixing this requires SMT integration or domain refinement, but we accept this precision loss for now.

This guarantees `x > 0` always maps to the same BDD variable, ensuring consistency. Mismatching it to 1 in one place and 2 elsewhere would create independent facts.



## 4.6 Exposing BDD Operations

We expose AND, OR, NOT so the rest of the engine can use them without touching the raw `Bdd` field. This encapsulates the BDD logic.

```
impl AnalysisManager {
  pub fn and(&self, a: Ref, b: Ref) → Ref {
    self.bdd.apply_and(a, b)
  }

  pub fn or(&self, a: Ref, b: Ref) → Ref {
    self.bdd.apply_or(a, b)
  }

  pub fn not(&self, a: Ref) → Ref {
    self.bdd.apply_not(a)
  }

  pub fn true_ref(&self) → Ref {
    self.bdd.mk_true()
  }

  pub fn false_ref(&self) → Ref {
    self.bdd.mk_false()
  }

  // Helper to visualize the BDD (for debugging)
  pub fn to_dot(&self, r: Ref) → String {
    bdd_rs::dot::to_dot(&self.bdd, r)
  }
}
```

## 4.7 Debugging with Graphviz

BDDs are graphs — visualize them to debug. We exposed a `to_dot` method in our manager:

```
// Inside main()
let dot_graph = mgr.to_dot(state);
println!("{}", dot_graph);
```

You can save this output to a `.dot` file and render it using Graphviz:

```
dot -Tpng output.dot -o output.png
```

**Key Insight** Visualizing the BDD is the fastest way to spot if your variable ordering is inefficient or if your logic is incorrect. If the graph looks like a tangled mess for a simple program, check your variable ordering!

## 4.8 Putting It Together

Test our manager with a simple scenario using the `Var`, `Op`, and `Condition` types:

```
// Make sure to include the AST definition from above!
// mod ast; use ast::{Var, Op, Condition};

fn main() {
    let mut mgr = AnalysisManager::new();

    // Encounter "x > 0"
    let x_gt_0 = Condition { var: Var::X, op: Op::Gt, val: 0 };
    let c1 = mgr.get_condition_var(&x_gt_0);

    // Encounter "y < 5"
    let y_lt_5 = Condition { var: Var::Y, op: Op::Lt, val: 5 };
    let c2 = mgr.get_condition_var(&y_lt_5);

    // State: x > 0 AND y < 5
    let state = mgr.and(c1, c2);

    // Encounter "x > 0" again!
    // The manager should return the SAME variable ID.
    let c3 = mgr.get_condition_var(&x_gt_0);

    // Should be the same variable
    assert_eq!(c1, c3);

    println!("State BDD: {:?}", state);
}
```

`AnalysisManager` is the foundation of our symbolic execution engine. Next chapter, we use it to “execute” IMP programs and build BDDs automatically.

**Advanced BDD Topics** For production BDD engines, two advanced topics are critical:

**Quantification ( $\exists$ ,  $\forall$ ):** Existential and universal quantification allow projecting out variables from BDDs. For example, checking if any input causes an error involves existential quantification. See `quantification.rs` (`cargo run --example bdd_quantification`) for implementation.

**Variable Ordering:** The order in which variables appear in the BDD is the single most important factor affecting size. A good ordering can result in linear node count, while a bad one causes exponential blowup. See `variable_ordering.rs` (`cargo run --example bdd_variable_ordering`) for strategies.

Variable ordering can make the difference between tractable and intractable analysis for the same formula!



### Exercise 1

Medium

#### Derived Operations:

1. Implement `implies(&self, a: Ref, b: Ref) → Ref` using `apply_not` and `apply_or`. Use this to check for **Redundant Checks** (if  $\text{Check}_A \Rightarrow \text{Check}_B$ , then  $\text{Check}_B$  might be redundant).

2. Implement `are_mutually_exclusive(&self, a: Ref, b: Ref) → bool`. Use this to check for **Unreachable Code** (if path condition  $P$  and branch condition  $C$  are mutually exclusive, the branch is dead).

## 4.9 Manager Internals Deep Dive

The `Bdd` manager maintains three critical components:

1. **Unique Table**: Hash map keyed by `(var, low, high)` ensuring canonical node reuse.
2. **Computed Cache**: Memoization table keyed by `(op, left, right)` returning previously computed results.
3. **Variable Metadata**: Ordering array plus auxiliary stats (usage counts, last reorder timestamp).



### Implementation Note

Use a fixed capacity hash table with load factor monitoring to trigger resizing. Instrument node insertions to gather a histogram of variable distribution.



### Common Pitfall

**Hash Explosion Risk**. Poor hashing of `(var, low, high)` triples can degrade performance by increasing collisions. Always combine fields with a stable mixing function.

### 4.9.1 Apply Operation Workflow

Every binary boolean operation follows a recursive pattern. Given `(f, g)` and operator `op`, the algorithm returns a canonical `Ref` through six steps:

- Step 1:** Check trivial cases (identity, annihilators, complements).
- Step 2:** Consult computed cache.
- Step 3:** Align top variables according to global ordering.
- Step 4:** Recurse on cofactors producing provisional children.
- Step 5:** Reduce by eliminating redundant tests.
- Step 6:** Insert or reuse via unique table.

#### Algorithm 1: Generic Apply

**Input:** BDD nodes  $f, g$ ; operator `op`.

**Output:** BDD node representing  $f \text{ op } g$ .

```

1  if trivial(f, g, op) then
2    return simplify(f, g, op)
3  if cache.contains(op, f, g) then
4    return cache.get(op, f, g)
5  v ← select_top_var(f, g)    // Choose root variable.
6  (f0, f1) ← cofactor(f, v)  // Split on low/high.
7  (g0, g1) ← cofactor(g, v)
8  low ← apply(op, f0, g0)    // Recursive calls.
9  high ← apply(op, f1, g1)
10 if low = high then
11   return low    // Redundant test elimination.
12 node ← unique_table.intern(v, low, high)  // Insert.
13 cache.put(op, f, g, node)  // Memoize.
14 return node

```

## 4.9.2 Instrumentation and Metrics

Add lightweight counters to monitor performance throughout BDD operations. Here are the most useful metrics to track:

**Node creations:** Count the total number of unique table insertions. High values indicate many new nodes are being created, suggesting complex formulas or poor sharing.

**Cache hits vs. misses:** Track the ratio of cached results to recomputations in apply operations. A high hit rate indicates good memoization; low rates suggest cache eviction or poor locality.

**Reductions applied:** Count how many times redundant test elimination fires. This shows how much simplification is happening during construction.

**Peak node count:** Track the maximum number of live nodes at any point. This high-water mark indicates memory pressure and helps tune garbage collection.



### Implementation Note

Expose a `struct BddStats` to hold these counters:

```

struct BddStats {
    node_creations: u64,
    cache_hits: u64,
    cache_misses: u64,
    reductions: u64,
}

```

```
peak_nodes: u64,
}
```

Update statistics at key points in the apply algorithm.



### Exercise 2

Medium

Add `BddStats` to your local fork and print a summary after constructing random conjunctions of 100 variables. Compare cache hit ratio before and after enabling complement edge optimization.

## 4.9.3 Concurrency and Thread Safety

BDD managers rely on global uniqueness, complicating multi-threaded usage. Parallel apply operations can race during node creation, violating canonicity.

Safe strategies:

- **Sharding**: Partition variable sets and build partial BDDs, then combine sequentially.
- **Task Queues**: Serialize unique table insertions while allowing parallel cofactor recursion.
- **Read-Mostly Locking**: Use an `RwLock` for unique table with short write locks on insertion.

**Caution Do Not Clone Managers Freely.** Cloning without deep copy of tables breaks pointer equality assumptions across instances.



### Exercise 3

Hard

Sketch a design for parallel apply using a work stealing deque for recursion tasks and a centralized insertion channel. Analyze potential contention points.

## 4.9.4 Advanced Quantification Example

Existential quantification removes variables by abstracting over both branches. The formula is:


$$\exists x. f(x) = f_{x=0} \vee f_{x=1} \quad (10)$$

Implementation merges cofactors then reduces.



### Hands-On Example

Hands on quantification implementation illustrating Shannon expansion reduction integration.

 **Source:** `quantification.rs`  
**Run:** `cargo run --example bdd_quantification`



### Exercise 4

Medium

Implement universal quantification  $\forall x. f(x) = f_{x=0} \wedge f_{x=1}$  and measure node count changes on structured vs random formulas.

## 4.9.5 Research Spotlight: Incremental Reordering

Incremental reordering seeks to relocate only a small subset of variables whose local neighborhoods grew disproportionately.

The approach consists of:

1. **Detect Hot Spots:** Variables with large subtree size growth since last snapshot.
2. **Local Sift Window:** Attempt swaps within a bounded index range.
3. **Rollback:** Revert if node count fails to improve beyond threshold.

Emerging work couples ML models predicting beneficial swap candidates using feature vectors (subtree size, fanout, clustering coefficient).

**P. C. McGeer (1993)**

*Dynamic Variable Reordering*

Early work formalized sifting heuristics establishing practical viability of dynamic ordering in industrial circuits.



### Exercise 5

Hard

Prototype a hot spot detector recording relative growth percentages per variable after each batch of 100 apply calls. Trigger a local reorder when any variable exceeds 20% growth.

## Chapter Summary

This chapter transformed BDD theory into concrete Rust implementation, establishing the analysis infrastructure.

We built the **AnalysisManager** as the central coordinator that maps program conditions to BDD variables. Its **canonicalization mechanism** ensures identical conditions consistently map to identical variables, enabling efficient reuse and preventing redundant variable allocation.

The implementation exposes **core Boolean operations** — AND, OR, NOT, ITE — that will serve as primitives for symbolic execution. By wrapping the **bdd-rs** library's low-level operations, we provide a clean interface for building complex symbolic state manipulations.

With this foundation established, we're equipped to implement the symbolic executor that automatically constructs BDDs while traversing program control flow.

# Chapter 5

## The Abstract Program State

We have the engine (BDDs) and the fuel (Abstract Domains). Now we build the vehicle that brings them together.

This chapter shows how to combine BDDs with abstract domains to track program state. Instead of storing a single abstract value per variable, we maintain **many** possibilities. Each possibility is guarded by a BDD representing which execution paths lead to it.

This combination is called **path-sensitive abstract interpretation**.

### 5.1 The Core Idea: Trace Partitioning

When analysis ignores which path was taken, it loses precision. Consider this simple branch:

```
let mut y = 0;
if x > 0 {
    y = 1;      // Path A: x > 0 implies y = 1
} else {
    y = 2;      // Path B: x ≤ 0 implies y = 2
}
// After merge point:
// Path-insensitive: y = {1} ∪ {2} = {1, 2}
// Lost information: (x > 0 ⇔ y = 1) and (x ≤ 0 ⇔ y = 2)
```

The problem is that path-insensitive analysis merges all incoming states regardless of how they were reached. The join operation  $y = 1 \sqcup y = 2$  produces  $y \in \{1, 2\}$ , forgetting which value corresponds to which input condition. We lost the correlation between the condition and the value.

Path-sensitive analysis maintains separate states:

```
// State 1: (path_condition: x > 0, environment: {y ↦ 1})
// State 2: (path_condition: x ≤ 0, environment: {y ↦ 2})
```

Now we preserve the correlation: for any concrete input  $x$ , we can determine the exact value of  $y$ .

However, maintaining separate states for each path faces a scaling challenge. With  $n$  independent conditions, explicit enumeration requires  $2^n$  states. A program with 30 boolean variables would need over a billion states, making explicit enumeration impractical.



### 5.1.1 BDD-Based Solution

Use BDDs to represent path conditions **symbolically**. Instead of enumerating  $2^n$  states explicitly, we maintain a set of pairs:

$$S = \{(b_1, \rho_1), (b_2, \rho_2), \dots, (b_k, \rho_k)\} \quad (11)$$

where:

- Each  $b_i$  is a BDD encoding a **set** of execution traces
- Each  $\rho_i$  is the abstract environment for variables along those traces
- The BDD operations (AND, OR, NOT) manipulate exponentially many paths in polynomial time

This technique is called **Trace Partitioning**.

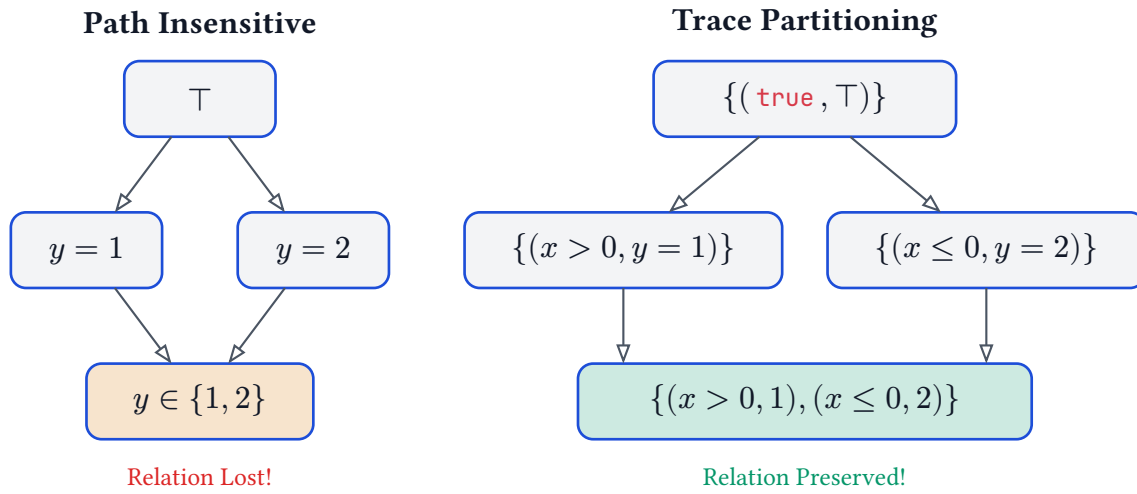


Figure 8: Trace Partitioning vs. Naive Merge

## 5.2 Architecture

Our design has three layers:

1. **BDD Path Tracker**: Uses BDD operations to track which execution paths are feasible.
2. **Abstract Environment**: Maps each variable to an abstract value (sign, interval, *etc.*).
3. **Combined State**: Pairs a BDD path condition with an abstract environment.

**Definition 6 (BDD-based Path-Sensitive Abstract State)** A state is a pair  $(b, \rho)$  where:

- $b$  is a BDD representing the path condition (which inputs reach here)
- $\rho$  is an abstract environment mapping variables to abstract values

The state represents: “For inputs satisfying  $b$ , the variables have values given by  $\rho$ .”

## 5.3 Implementation: PathSensitiveState

Here's the complete structure:

```
use std::collections::HashMap;
use std::rc::Rc;
use bdd_rs::bdd::Bdd;
use bdd_rs::reference::Ref;

struct PathSensitiveState {
    bdd: Rc<Bdd>,
    path: Ref,           // BDD representing path condition
    env: HashMap<String, Sign>, // Variable → Sign mapping
    next_var: u32,       // Next BDD variable to allocate
}
```

Each state tracks:

1. `path`: BDD formula describing which paths reach this state
2. `env`: Abstract environment mapping variables to values
3. `next_var`: Counter for fresh BDD variable allocation

### 5.3.1 Core Operations

Creating initial state:

```
impl PathSensitiveState {
    fn new(bdd: Rc<Bdd>) → Self {
        Self {
            bdd: bdd.clone(),
            path: bdd.mk_true(), // Initially: all paths feasible
            env: HashMap::new(),
            next_var: 1,
        }
    }
}
```

Checking feasibility:

```
fn is_feasible(&self) → bool {
    !self.bdd.is_zero(self.path)
}
```

When a path's BDD becomes False, that path is impossible.

### 5.3.2 Branching: Splitting Paths

At a branch point, we create two states with refined path conditions:

```
fn branch(&self) → (Self, Self) {
    // Allocate fresh BDD variable for this condition
    let cond_var = self.next_var;
    let cond_bdd = self.bdd.mk_var(cond_var);

    // True branch: path ∧ condition
    let true_path = self.bdd.apply_and(self.path, cond_bdd);
```

```

// False branch: path ∧ ¬condition
let not_cond = self.bdd.apply_not(cond_bdd);
let false_path = self.bdd.apply_and(self.path, not_cond);

// Return both states (details omitted)
(true_state, false_state)
}

```

Each branch gets a copy of the environment with a refined path condition.

Example usage:

```

let (mut true_state, mut false_state) = state.branch();

// True branch: x = 5
true_state.assign("x", Sign::Pos);

// False branch: x = -3
false_state.assign("x", Sign::Neg);

```

### 5.3.3 Joining: Merging Paths

When paths reconverge, we join them:

```

fn join(&self, other: &Self) → Self {
    // Merge path conditions: path1 ∨ path2
    let merged_path = self.bdd.apply_or(self.path, other.path);

    // Join data environments
    let mut merged_env = HashMap::new();
    for var in all_variables {
        let val1 = self.get(var);
        let val2 = other.get(var);
        merged_env.insert(var, val1.join(&val2));
    }
    // ... return merged state
}

```

The join operation combines path conditions with OR and abstract values with domain join ( $\sqcup$ ).

## 5.4 Advanced: Condition Management

Production analyzers optimize BDD variable allocation by recognizing negated predicates.

### 5.4.1 Condition Representation

Define an enum for program predicates:

```

#[derive(Debug, Clone, Hash, Eq, PartialEq)]
enum Condition {
    Greater(String, i32),    // x > c
    GreaterEq(String, i32),  // x ≥ c
    Less(String, i32),       // x < c
    LessEq(String, i32),     // x ≤ c
}

```

```

    Equals(String, i32),    // x = c
    NotEquals(String, i32), // x ≠ c
}

impl Condition {
    fn negate(&self) → Self {
        match self {
            Condition::Greater(var, val) ⇒ Condition::LessEq(var.clone(), *val),
            Condition::GreaterEq(var, val) ⇒ Condition::Less(var.clone(), *val),
            Condition::Less(var, val) ⇒ Condition::GreaterEq(var.clone(), *val),
            Condition::LessEq(var, val) ⇒ Condition::Greater(var.clone(), *val),
            Condition::Equals(var, val) ⇒ Condition::NotEquals(var.clone(), *val),
            Condition::NotEquals(var, val) ⇒ Condition::Equals(var.clone(), *val),
        }
    }
}

```

## 5.4.2 Negation-Aware BDD Allocation

The `AnalysisManager` caches conditions and reuses BDD variables for negations:

```

struct AnalysisManager {
    bdd: Bdd,
    mapping: HashMap<Condition, usize>, // Condition → BDD variable ID
    next_var_id: usize,
}

impl AnalysisManager {
    pub fn get_bdd(&mut self, c: &Condition) → Ref {
        // 1. Check if condition already has a variable
        if let Some(&id) = self.mapping.get(c) {
            return self.bdd.mk_var(id as u32);
        }

        // 2. Check if negation has a variable (reuse with NOT)
        let neg_c = c.negate();
        if let Some(&id) = self.mapping.get(&neg_c) {
            let var = self.bdd.mk_var(id as u32);
            return self.bdd.mk_not(var);
        }

        // 3. Allocate new variable
        let id = self.next_var_id;
        self.next_var_id += 1;
        self.mapping.insert(c.clone(), id);
        self.bdd.mk_var(id as u32)
    }
}

```

Step 2 halves variable usage for predicate pairs like `x > 0` and `x ≤ 0`.

## 5.4.3 Automatic Contradiction Detection

Consider sequential branches:

```

if x > 0 {           // Allocates BDD variable v1
    // ...
}
if x ≤ 0 {           // Recognizes negation, returns !v1

```

```
// ...
}
```

The BDD for the second branch is automatically  $\neg v_1$ . If both branches are taken (infeasible path), the conjunction becomes:

$$v_1 \wedge \neg v_1 = F \quad (12)$$

The BDD library detects this contradiction immediately, pruning the impossible path.

## 5.5 Alternative Design: Partitioning

The `PathSensitiveState` shown earlier clones states at each branch. An alternative design maintains **all** path-environment pairs in a single structure. This section sketches this approach to illustrate the design space.

Instead of cloning, we store multiple path, data pairs in one container. Each pair represents a partition of the program's execution space.

### 5.5.1 Partitioned State Representation

The core data structure:

```
struct PartitionedState<D: AbstractDomain> {
    partitions: Vec<Ref, D>, // List of (path, environment) pairs
    control: Rc<RefCell<AnalysisManager>>,
}
```

Invariant: path conditions should be mutually disjoint. That is, for  $i \neq j$ :  $\text{path}_i \wedge \text{path}_j = F$ .

### 5.5.2 Semantic Interpretation

The partitioned state  $S = \{(b_1, \rho_1), \dots, (b_k, \rho_k)\}$  represents a **disjunction**:

$$S \equiv (b_1 \wedge \rho_1) \vee (b_2 \wedge \rho_2) \vee \dots \vee (b_k \wedge \rho_k) \quad (13)$$

Reading: “Either the execution satisfies path condition  $b_1$  with variable environment  $\rho_1$ , OR it satisfies  $b_2$  with environment  $\rho_2, \dots$ ”

### 5.5.3 Matching Concrete Executions

Given a concrete input  $\sigma$ , we can determine which partition corresponds to that execution:

1. Search for partition  $i$  where the concrete input  $\sigma$  satisfies the BDD condition  $b_i$ . Since partitions are disjoint, at most one will match.
2. Once found, check whether the actual variable values match the abstract environment  $\rho_i$ .

This allows us to verify that our abstract analysis correctly covers all concrete behaviors.

#### State Evolution Example

Consider program:

```
let mut y = 0;
if x > 0 { y = 1; }
else { y = 2; }
```

State evolution:

1. **Initial:**  $\{(T, \{y \mapsto \perp\})\}$
2. **After  $y = 0$ :**  $\{(T, \{y \mapsto 0\})\}$
3. **After split:**
  - True branch:  $\{(x > 0, \{y \mapsto 1\})\}$
  - False branch:  $\{(\neg(x > 0), \{y \mapsto 2\})\}$
4. **After join:**  $\{(x > 0, \{y \mapsto 1\}), (\neg(x > 0), \{y \mapsto 2\})\}$

Final state has two partitions preserving exact correlation between  $x$  and  $y$ .

## 5.5.4 Smart Joining Strategy

When two control flow paths converge, we must join their states. Naive approach: create a single partition with joined environments. Smart approach: preserve distinctions when possible.

When two partitions have **identical** abstract environments, merge their path conditions:

$$(b_1, \rho) \sqcup (b_2, \rho) = (b_1 \vee b_2, \rho) \quad (14)$$

### Merge Opportunity

Two branches assign the same value:

```
if x > 0 { y = 5; }      // Partition: (x > 0, {y ↦ [5,5]})
else { y = 5; }         // Partition: (x ≤ 0, {y ↦ [5,5]})
```

After join: single partition  $(T, \{y \mapsto [5, 5]\})$  because environments are identical.

Join algorithm:

```
fn join(&self, other: &Self) → Self {
    let mut result = self.partitions.clone();

    for (path2, env2) in &other.partitions {
        // Try to find matching environment
        if let Some((path1, _)) = result.iter_mut().find(|(_, e)| e == env2) {
            *path1 = bdd.apply_or(*path1, *path2); // Merge paths
        } else {
            result.push((*path2, env2.clone())); // Add new partition
        }
    }
}
```

```
// ...
}
```

Without smart joining:  $n$  branches  $\Rightarrow 2^n$  partitions. With smart joining: common case  $O(k)$  partitions where  $k \ll 2^n$ .

## 5.6 Refining Abstract Values: Bidirectional Flow

BDDs track control flow predicates. Abstract domains track data value properties. Key insight: these two components can **inform each other**.

### 5.6.1 The Refinement Problem

Consider:

```
let mut x = read_input(); // x: [-∞, +∞]
if x < 10 {
    // What do we know about x here?
}
```

After the branch, the BDD path condition records  $x < 10$ , but the interval domain still has  $x \in [-\infty, +\infty]$  unless we explicitly communicate this constraint.

To solve this, we extract the numeric constraint from the branch condition and use it to **refine** the abstract environment. This ensures both components stay synchronized.

### 5.6.2 Generic Refinement Interface

Define a trait for domains that support constraint-based refinement:

```
trait Refineable {
    fn refine(&mut self, constraint: &Condition);
}

impl Refineable for IntervalDomain {
    fn refine(&mut self, constraint: &Condition) {
        match constraint {
            Condition::Less(var, val) => /* tighten upper bound */,
            Condition::Greater(var, val) => /* tighten lower bound */,
            // ...
        }
    }
}
```

### 5.6.3 Assume Operation: Coordinating Both Layers

The **assume** operation updates both control (BDD) and data (domain) components:

```
fn assume(&mut self, c: &Condition) {
    let bdd_cond = self.control.get_bdd(c);
```

```

for (path, env) in &mut self.partitions {
    *path = bdd.apply_and(*path, bdd_cond); // Update control
    env.refine(c);                          // Update data
}

// Remove infeasible partitions (path = False)
self.partitions.retain(|(p, _)| !bdd.is_zero(*p));
}

```

## 5.6.4 Example: Interval Refinement in Action

```

// Initial state: {(True, {x ↦ [-∞, +∞]})}
let mut state = PartitionedState::new(control);

// After: if x < 10
state.assume(&Condition::Less("x", 10));
// State: {(x < 10, {x ↦ [-∞, 9]})}
//          ^^^^^^      ^^^^^^^^^^^^^^
//          BDD         Interval refined!

// After: if x > 5 (nested condition)
state.assume(&Condition::Greater("x", 5));
// State: {(x < 10) ∧ (x > 5), {x ↦ [6, 9]})}
//          ^^^^^^^^^^^^^^^^^^^^^^^^^^      ^^^^^^^^^^^^^^
//          BDD tracks both      Interval refined twice!

```

The interval domain automatically tightens bounds based on branch conditions, even though these constraints come from control flow, not assignments.

**Key Insight** Refinement creates a **feedback loop** between control and data:

- BDD operations determine path feasibility
- Constraints extracted from paths refine data abstractions
- Refined data can rule out additional paths

## 5.7 Putting It Together: The Interpreter Loop

Now that we have all the pieces — branching, joining, refinement — let's see how they fit together. The abstract interpreter walks through program statements, applying these operations:

```

fn eval_stmt(stmt: &Stmt, state: &mut State) {
    match stmt {
        Stmt::Assign(var, expr) => {
            state.assign(var, expr);
        }
        Stmt::If(cond, then_block, else_block) => {
            let mut true_state = state.clone();
            let mut false_state = state.clone();

            true_state.assume(cond);
            false_state.assume(&cond.negate());
        }
    }
}

```



```

    eval_block(then_block, &mut true_state);
    eval_block(else_block, &mut false_state);

    *state = true_state.join(&false_state);
  }
  Stmt::While(cond, body) => {
    // Fixpoint iteration (see Chapter 10)
  }
}
}

```

The interpreter handles three types of statements, each with different effects on state:

**Assignment statements:** Update the variable environment with new values. The path condition remains unchanged since assignments don't affect control flow.

**Branch statements:** Split the state into two copies, one for each branch direction. Refine both path conditions and environments using the `assume` operation. After analyzing both branches, join the results back together.

**Loop statements:** Require fixpoint iteration to handle potential unbounded execution. Chapter 10 covers the widening operators needed to ensure termination.

## 5.8 Understanding the Product: How BDDs and Domains Cooperate

Let's step back and understand the fundamental mechanism at work. The key insight: BDDs and abstract domains form a **feedback loop**.

Each state  $(b, \rho)$  combines:

- $b$ : BDD tracking which paths are feasible
- $\rho$ : abstract environment with variable values

These two components inform each other. This is an instance of **product domain** construction (see formal definition in Section 11).

### 5.8.1 Control Flow Refines Data Values

When we take a branch, the control flow condition provides information about variable values. Consider:

```

if x > 0 {
  // BDD records: path ∧ (x > 0)
  // Domain refines: x ∈ [1, +∞]
}

```

The branch condition `x > 0` not only updates the BDD path condition, but also allows the interval domain to tighten its bounds. Inside the branch, we know  $x > 0$ , so the interval can be refined from  $[-\infty, +\infty]$  to  $[1, +\infty]$ .

## 5.8.2 Data Values Eliminate Infeasible Paths

Conversely, when the abstract domain has precise information, it can prove that certain paths are impossible. Consider:

```
x = 5;           // Domain: x ∈ [5, 5]
if x < 0 {       // BDD would add: path ∧ (x < 0)
    // Unreachable! // But [5,5] ∩ (-∞,0) = ∅
}               // Drop this partition
```

Here, the interval domain knows  $x \in [5, 5]$ . When we encounter the condition `x < 0`, the intersection  $[5, 5] \cap (-\infty, 0) = \emptyset$  is empty. This proves the path is infeasible, so we can drop this partition entirely.

This bidirectional refinement is called **reduction**. The process continues until neither component can tighten further. For formal reduction operators with soundness proofs, see Section 11.

**Key Insight** Constraints flow between domains until stabilization. When neither the BDD control layer nor the data domain can further refine the other, the system has reached a stable reduced state.

## 5.9 Managing Partition Growth

As programs execute, partitions multiply. Without management, memory exhausts quickly. We need strategies to keep the number of partitions under control while preserving as much precision as possible.

### 5.9.1 Strategy 1: Remove Infeasible Partitions

The simplest strategy is to remove partitions whose BDD becomes False. These represent impossible execution paths and contribute nothing to the analysis.

```
partitions.retain(|(path, _)| !bdd.is_zero(*path));
```

This is essentially free — we’re discarding partitions that are already proven unreachable.

### 5.9.2 Strategy 2: Merge Partitions with Identical Environments

When two partitions have the same abstract values for all variables, we can merge their path conditions. For example:

```
// (x > 0, {y ↦ 5}) and (x < 10, {y ↦ 5})
// → ((x > 0) ∨ (x < 10), {y ↦ 5})
```

This reduces the number of partitions without losing precision, since the abstract environments are identical.

### 5.9.3 Strategy 3: Cap Partition Count (k-limiting)

When all else fails, we can impose a hard limit on the number of partitions. For example, set a maximum of  $k = 10$  partitions.

```
const MAX_PARTITIONS: usize = 10;
if partitions.len() > MAX_PARTITIONS {
    merge_most_similar(&mut partitions);
}
```

When this limit is exceeded, we force-merge the most similar partitions (those with closest abstract values). This trades precision for performance, but ensures the analysis remains tractable.

With these management strategies in place, let's see path-sensitive analysis in action.

## 5.10 Complete Example: Temperature Controller Analysis

Let's apply everything we've learned to a realistic embedded system. Consider a temperature controller with safety-critical bounds:

```
fn control_temp(sensor: i32) → i32 {
    let mut heater_power = 0;
    if sensor < 15 {
        heater_power = 100; // Full power: cold room
    } else if sensor < 20 {
        heater_power = 50; // Half power: moderate
    } else {
        heater_power = 0; // Off: warm enough
    }
    heater_power
}
```

**Safety property:** `heater_power` must always be in  $[0, 100]$  (hardware constraint).

### 5.10.1 Path-Insensitive Interval Analysis

Uses a single interval per variable, merging all paths.

Analysis trace:

1. Entry: `sensor` =  $\top$  (unknown), `heater_power` =  $\perp$
2. After `heater_power = 0`:
  - `heater_power` =  $[0, 0]$
3. First branch (`sensor < 15`):
  - True: `heater_power` =  $[100, 100]$
  - False: `heater_power` =  $[0, 0]$
4. Merge after first `if`:
  - `heater_power` =  $[0, 0] \sqcup [100, 100] = [0, 100]$

5. Second branch (`sensor < 20` on false path):
  - True: `heater_power` = [50, 50]
  - False: `heater_power` = [0, 0]
6. Final merge:
  - `heater_power` = [0, 100]  $\sqcup$  [50, 50]  $\sqcup$  [0, 0] = [0, 100]

The analysis concludes that `heater_power`  $\in$  [0, 100], so the safety property holds.

However, this path-insensitive analysis has lost significant information. It cannot answer important questions:

- “Under what conditions is heater at full power?”
- “Is `heater_power = 75` ever possible?”
- “What sensor range triggers half power?”

Path-sensitivity will let us answer these questions precisely.

## 5.10.2 Path-Sensitive BDD Analysis

Maintains separate partitions for each control flow path.

Analysis trace:

1. **Entry:**

$$\{(T, \{\text{heater\_power} \mapsto \perp, \text{sensor} \mapsto \top\})\} \quad (15)$$

2. **After `heater_power = 0`:**

$$\{(T, \{\text{heater\_power} \mapsto [0, 0], \text{sensor} \mapsto \top\})\} \quad (16)$$

3. **First split (`sensor < 15`):**

- **True partition:**

$$(\text{sensor} < 15, \{\text{heater\_power} \mapsto [100, 100], \text{sensor} \mapsto [-\infty, 14]\}) \quad (17)$$

- **False partition:**

$$(\neg(\text{sensor} < 15), \{\text{heater\_power} \mapsto [0, 0], \text{sensor} \mapsto [15, +\infty]\}) \quad (18)$$

4. **Second split on false partition (`sensor < 20`):**

- **True partition (from false):**

$$((\neg(s < 15)) \wedge (s < 20), \{\text{heater\_power} \mapsto [50, 50], s \mapsto [15, 19]\}) \quad (19)$$

- **False partition (from false):**

$$((\neg(s < 15)) \wedge \neg(s < 20), \{\text{heater\_power} \mapsto [0, 0], s \mapsto [20, +\infty]\}) \quad (20)$$

5. **Final state (three partitions):**

$$\begin{aligned} S = \{ & (\text{sensor} < 15, \{\text{hp} \mapsto 100, \text{sensor} \mapsto [-\infty, 14]\}), \\ & ((\text{sensor} \geq 15) \wedge (\text{sensor} < 20), \{\text{hp} \mapsto 50, \text{sensor} \mapsto [15, 19]\}), \\ & (\text{sensor} \geq 20, \{\text{hp} \mapsto 0, \text{sensor} \mapsto [20, +\infty]\}) \} \end{aligned} \quad (21)$$

Now we can verify the safety property on each partition separately:

- Partition 1:  $100 \in [0, 100]$  ✓
- Partition 2:  $50 \in [0, 100]$  ✓
- Partition 3:  $0 \in [0, 100]$  ✓

The property holds on all paths, confirming the system is safe.

### 5.10.3 Advanced Queries Enabled by Path-Sensitivity

Path-sensitivity enables queries about the relationship between inputs and outputs:

- “When is heater at full power?”  $\rightarrow$  `sensor < 15`
- “Is `heater_power = 75` possible?”  $\rightarrow$  No (only 0, 50, 100)
- “What sensor values cause half power?”  $\rightarrow$  `sensor`  $\in [15, 19]$

This transforms analysis from “what values are possible” into “under which conditions do these values occur”.

## 5.11 Beyond Single Domains: Combining Multiple Abstractions

So far we’ve paired BDDs with a single abstract domain (sign, interval, *etc.*). But we can go further: combine **multiple** data abstractions simultaneously. Each domain contributes different information, and reduction lets them refine each other. See `combined.rs` (`cargo run --example combined`) for runnable examples.

### 5.11.1 Sign $\times$ Interval Product

The sign domain detects zero division and distinguishes `Pos` from `Neg` values. The interval domain tracks numeric bounds. Together, they can refine each other:

- If the sign domain determines a value is `Zero`, the interval domain can refine its bounds to  $[0, 0]$ .
- Conversely, if the interval domain has bounds  $[5, 10]$ , the sign domain can refine its value to `Pos`.

### 5.11.2 Interval $\times$ Congruence Product

Intervals provide bounds, while congruences track divisibility properties (e.g., `Even` or “divisible by 4”). When combined, they can significantly reduce the set of possible values:

- Interval  $[8, 12]$  with congruence  $\equiv 0 \bmod 4$  refines to  $\{8, 12\}$ . Only 8 and 12 in this range satisfy the divisibility constraint.

### 5.11.3 Polyhedra $\times$ Interval Product

Polyhedra capture linear relations between variables (e.g.,  $x - y \leq 5$ ). Intervals provide quick bounds for individual variables. Reduction works by projecting the polyhedron:

- Project the polyhedron onto a single variable to extract tighter bounds.

- Use interval bounds to simplify polyhedron constraints.



### Hands-On Example

Implements a reduced product of sign and parity domains, demonstrating how reduction eliminates impossible combinations like (Zero, Odd).

Source: `combined.rs`  
 ▶ Run: `cargo run --example combined_domain`



### Exercise

*Hard*

Design a reduced product of interval and parity (`Even/Odd`) domains. Implement reduction: interval  $[2, 5]$  with parity `Even` refines to  $\{2, 4\}$ . Measure overhead versus precision gain.

## 5.12 Relational Domains and BDD Synergy

**Caution Advanced Content:** Relational domains are covered in detail in Part II, Chapter 14. This section is optional for introductory readers.

Relational domains (octagons, polyhedra) track relationships between variables, like  $x - y \leq 5$ . Combining them with BDDs enables **conditional** relations: “ $x - y \leq 5$  when  $z > 0$ ”.

This requires careful design:

1. BDD tracks control predicates (boolean).
2. Relational domain tracks numeric relations.
3. Reduction extracts numeric bounds from BDD (when possible) and injects into polyhedron.

**Caution** Relational domains have cubic complexity in variable count. Partition explosion with relational domains can become intractable quickly. Consider simpler domains (intervals) for control-heavy code and relational domains for data-heavy loops.



### Exercise

*Hard*

Sketch integration of octagon domain with BDD control. Propose reduction operator extracting difference bounds from path predicates. Discuss complexity trade offs.

## 5.13 Research Spotlight: Trace Partitioning in Context

**Research Context** This section discusses the theoretical foundations and research literature. Useful for readers interested in the formal underpinnings, but not essential for practical implementation.

The trace partitioning approach we've used throughout this chapter has deep theoretical roots. Trace partitioning maintains disjoint path sets, each with its own abstract value. The **powerset** domain allows arbitrary sets of abstract elements without disjointness constraints. Trace partitioning is a restricted powerset where partitions align with control flow splits.

**Mauborgne and Rival (2005)**

*Trace Partitioning*

Formalized trace partitioning as a systematic framework for path-sensitive analysis distinguishing syntactic control predicates from semantic abstract values.

Recent work explores several promising directions:

**Dynamic partitioning:** Adjust partition granularity dynamically based on precision needs. Maintain fine-grained partitions where precision matters, coarser partitions elsewhere.

**Predicate abstraction:** Automatically choose relevant predicates to track using counterexample-guided refinement (CEGAR). Start with coarse abstraction and refine only when necessary to eliminate false alarms.

**Hybrid partitioning:** Combine trace partitioning with relational domains for specific variable clusters. Use path-sensitivity for control-heavy code, relational domains for data-intensive loops.



### Exercise

*Hard*

Compare trace partitioning (our approach) with full powerset on a loop with nested conditionals. Measure state count, analysis time, and precision (count false alarms).

## 5.14 Engineering Perspective: Implementation Trade-offs

**Engineering Considerations** This section discusses practical engineering trade-offs in production analyzers. Part III (Performance and Optimization) covers these topics in depth with benchmarks.

We've focused on correctness and precision, but production analyzers must also consider performance. Path-sensitive analysis can be expensive, so real systems balance precision against speed.

### 5.14.1 Eager Merge Strategy

Join states at every merge point in the control flow graph. This loses path-sensitivity early but bounds the number of states. Suitable for programs where path-sensitivity provides little benefit.

### 5.14.2 Lazy Merge Strategy

Maintain partitions as long as possible, only merging when absolutely necessary. Merging is forced at loop convergence points or when hitting state limits. Provides maximum precision at the cost of potentially exponential state growth.

### 5.14.3 Selective Sensitivity Strategy

Partition only on predicates likely to improve precision. For example, track null checks and array bounds precisely, but merge on other conditions. Requires heuristics or annotations to identify important predicates.

### 5.14.4 Predicate Sampling Strategy

Allocate BDD variables for only a subset of program conditions. Other conditions are handled by merging their states. Balances precision and performance by focusing resources on critical predicates.



#### Implementation Note

Expose a configuration parameter `sensitivity_mode: Enum { Full, Limited(k), Selective }`. Profile analysis runs measuring partition counts, memory usage, and precision metrics.



#### Exercise 5

*Medium*



Implement selective sensitivity: partition only on predicates marked `@sensitive` in annotations. Compare precision and performance versus full partitioning on annotated programs.

## 5.15 Summary

Path-sensitive analysis combines BDDs with abstract domains to track precise program state.

Core concepts:

- States are pairs  $(b, \rho)$ : BDD path condition + abstract environment
- BDDs compactly represent exponentially many paths
- Abstract domains track variable properties on each path

Key operations:

**Branching splits states:** Create two refined path conditions:  $\text{path} \wedge \text{cond}$  for true branch,  $\text{path} \wedge \neg \text{cond}$  for false branch. Each branch gets its own copy of the environment to track independently.

**Assignment updates values:** Modify the variable environment with new abstract values. The path condition stays the same since assignments don't create new paths.

**Joining merges paths:** Combine path conditions with logical OR. Join data environments using the domain's join operation ( $\sqcup$ ).

**Feasibility checking prunes paths:** When a BDD becomes False, that path is impossible and can be discarded immediately.

Implementation patterns:

```
// Basic: PathSensitiveState (code-examples/integration/sign_with_bdd.rs)
struct PathSensitiveState {
    path: Ref, // BDD path condition
    env: HashMap<String, Sign>, // Variable → abstract value
}

// Advanced: PartitionedState (production)
struct PartitionedState {
    partitions: Vec<(Ref, Domain)>, // Multiple (path, env) pairs
}
```

Trade-offs:

- Join early: fast, loses precision
- Join late: precise, exponential blowup
- $k$ -limiting: cap partitions, balance both



### Hands-On Example

Complete working implementation of basic path-sensitive analysis.

```
Source:  sign_with_bdd.rs
Run:    cargo run --example sign_with_bdd
```

In the next chapter, we build a symbolic executor using these techniques.

## Chapter Summary

This chapter synthesized control flow and data abstraction into a unified path-sensitive analysis framework.

The key architectural insight is **layered state representation**:  $(b, \rho)$  pairs a BDD tracking feasible execution paths with an abstract domain tracking variable values. This separation enables **orthogonal composition** — the BDD layer handles control flow sensitivity independently of the chosen data abstraction (signs, intervals, polyhedra, *etc.*).

**Path condition refinement** occurs automatically at branches. The true branch conjoins the condition to the path BDD ( $\text{path} \wedge \text{cond}$ ), while the false branch conjoins its negation. When a path BDD becomes False, that execution path has been proven infeasible through contradiction, enabling **automatic infeasible path pruning**.

This architecture **represents  $2^n$  paths symbolically** while performing operations in polynomial time. Rather than explicitly enumerating execution paths, Boolean operations on BDDs manipulate entire path families implicitly.

The **control-data cooperation** reveals its power when path conditions constrain data domains. If a path BDD encodes  $x = 5$ , the data domain can strengthen its approximation accordingly, recovering precision lost to path-insensitive merging.

This framework's **genericity** enables experimenting with different partitioning strategies, trading between path sensitivity (precision) and state space size (performance).

# Chapter 6

## A Complete Example: Symbolic Execution Engine

Theory and fragments are valuable, but nothing beats a complete working example. This chapter implements a simple symbolic execution engine for IMP programs using BDDs, analyzing real programs, tracking execution flows symbolically, and detecting bugs like unreachable code.

### 6.1 What is Symbolic Execution?

Symbolic execution runs programs with *symbolic* inputs instead of concrete values.

Traditional execution:

```
fn check(x: i32) → Status {  
  if x > 0 { Safe } else { Error }  
}
```

Symbolic execution:

```
// Symbolic: x is symbol a  
// Path 1: a > 0 → Safe  
// Path 2: a ≤ 0 → Error
```

Symbolic execution generates path conditions (Boolean formulas) and symbolic states.

**Definition 7 (Symbolic Program Execution)** Symbolic execution simulates program flows with symbolic variables, maintaining:

1. **Path condition:** Boolean formula describing constraints on inputs to reach this point.
2. **Symbolic state:** Maps each variable to a symbolic expression or value set.
3. **Flow exploration:** Fork at branches to explore both true and false paths.

This implementation takes a hybrid approach: path conditions are represented compactly as BDDs instead of explicit formulas or constraint solvers. Symbolic values remain as expressions, but path feasibility is tracked through BDD operations. This combines symbolic execution precision with BDD-based path representation efficiency.

## 6.2 Architecture Overview

1. **Condition Language:** Represent program conditions
2. **Symbolic State:** Path condition (BDD) + variable environment
3. **Program Walker:** Execute statements, update state
4. **Path Explorer:** Manage multiple paths, detect conflicts

This brings together everything we've learned — abstract domains, BDDs, control flow, and path-sensitive analysis. Study this implementation to see all the pieces working in harmony:



### Hands-On Example

Complete symbolic execution engine implementation with statement evaluation, path branching, conflict checking, and bug detection. This is the culmination of all concepts from previous chapters working together in a production-ready system.

**Source:** `executor.rs`  
**Run:** `cargo run --example symbolic_executor`

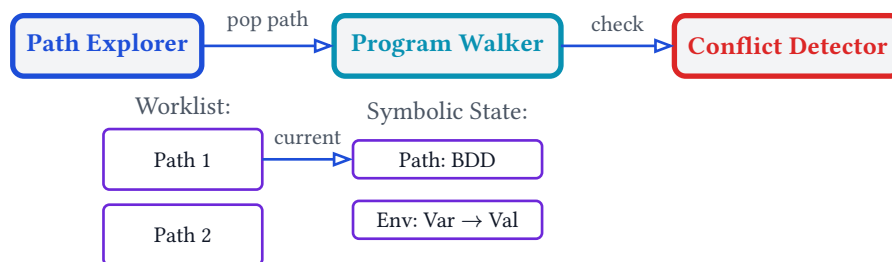


Figure 9: Symbolic executor architecture. The path explorer manages a worklist of symbolic states, each containing a BDD path condition and symbolic variable environment. The program walker processes statements, forking states at branches. Conflict detectors check for unreachable code by querying path feasibility.

## 6.3 Condition Language

We use the AST defined in Section 1.

```
// Recall from Chapter 1:
// pub enum Var { X, Y, ... }
// pub struct Condition { var: Var, op: Op, val: i32 }
```

Example:

```
// x > 0
let c = Condition { var: Var::X, op: Op::Gt, val: 0 };
```

## 6.4 Symbolic State

We reuse the `AnalysisManager` we built in Section 5 to manage our BDD variables. This ensures that if we encounter `x > 0` in different parts of the program, they map to the same BDD variable.

```
use bdd_rs::{Bdd, Ref};
use std::rc::Rc;
use std::cell::RefCell;
use std::collections::HashMap;

// Recall AnalysisManager from Chapter 5
struct SymbolicState {
    ctx: Rc<RefCell<AnalysisManager>>,
    path: Ref, // Path condition (BDD)
    env: HashMap<Var, i32>, // Var → Symbolic Value (simplified)
}

impl SymbolicState {
    fn new(ctx: Rc<RefCell<AnalysisManager>>) → Self {
        let true_path = ctx.borrow().bdd.mk_true();
        Self {
            ctx,
            path: true_path,
            env: HashMap::new(),
        }
    }

    fn is_feasible(&self) → bool {
        let bdd = &self.ctx.borrow().bdd;
        self.path != bdd.mk_false()
    }
}
```

## 6.5 Modifications and Actions

Evaluate modifications symbolically (e.g., assignments):

```
impl SymbolicState {
    fn assign(&mut self, var: Var, value: i32) {
        // In a real symbolic executor, value would be an expression
        self.env.insert(var, value);
    }
}
```

Example:

```
let mut state = SymbolicState::new(Rc::new(Bdd::default()));

// x = 10
state.assign(Var::X, 10);
```

## 6.6 Branching

When encountering an `If` statement, we split execution. We first evaluate the condition to its symbolic form, then check if we've seen it before.

```
impl SymbolicState {
  fn branch(&mut self, c: &Condition) → SymbolicState {
    // 1. Get canonical BDD from manager
    // The manager ensures that identical conditions map to the same BDD node
    let cond_bdd = self.ctx.borrow_mut().get_bdd(c);
    let bdd = &self.ctx.borrow().bdd;

    // True path: path ∧ cond
    let true_path = bdd.apply_and(self.path, cond_bdd);
    let mut true_state = self.clone();
    true_state.path = true_path;

    // False path: path ∧ ¬cond
    let false_path = bdd.apply_and(self.path, bdd.apply_not(cond_bdd));
    let mut false_state = self.clone();
    false_state.path = false_path;

    // Update self to true branch, return false branch
    *self = true_state;
    false_state
  }
}
```

By using `AnalysisManager`, we ensure identical conditions map to the same BDD variable. This lets the BDD deduce that a repeated check (e.g., `if x > 0` twice) is redundant.

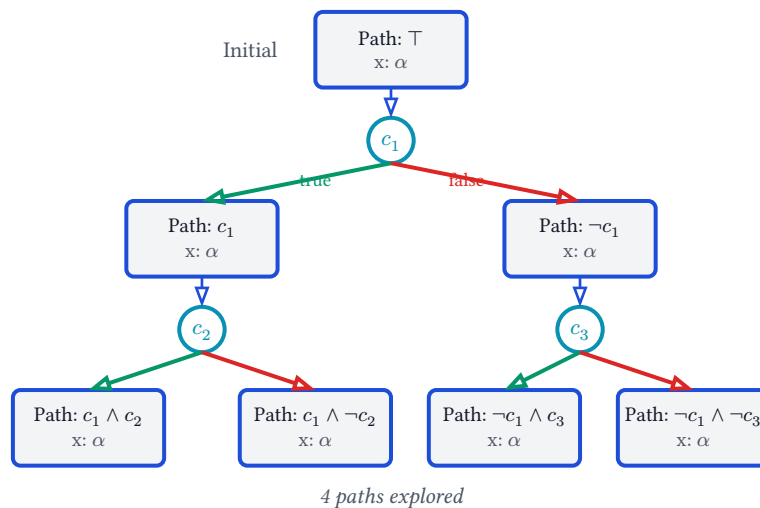


Figure 10: Flow forking at branches. Starting from an initial state, each `Condition` allocates a fresh BDD variable and splits into two states. The true branch updates the path with  $(p \wedge c)$ , the false branch with  $(p \wedge \neg c)$  where  $p$  is the current path condition. Both branches inherit the symbolic environment.

## 6.7 Program Representation

We use the `Stmt` enum from Section 1.

```
// Recall from Chapter 1:
// pub enum Stmt {
//     If(Condition, Box<Stmt>, Box<Stmt>), // if cond then ... else ...
//     Assign(Var, i32),
//     Seq(Box<Stmt>, Box<Stmt>),
//     ...
// }
```

## 6.8 Program Walker

Execute statements, tracking symbolic state:

```
struct ProgramWalker {
    ctx: Rc<RefCell<AnalysisManager>>,
}

impl ProgramWalker {
    fn new() → Self {
        Self {
            ctx: Rc::new(RefCell::new(AnalysisManager::new())),
        }
    }

    fn execute(&self, stmt: &Stmt) → Vec<SymbolicState> {
        let initial = SymbolicState::new(self.ctx.clone());
        self.execute_stmt(stmt, initial)
    }

    fn execute_stmt(&self, stmt: &Stmt, mut state: SymbolicState) → Vec<SymbolicState> {
        if !state.is_feasible() {
            return vec![]; // Dead path
        }

        match stmt {
            Stmt::Assign(var, val) => {
                state.assign(*var, *val);
                vec![state]
            }

            Stmt::Seq(s1, s2) => {
                let states = self.execute_stmt(s1, state);
                states.into_iter()
                    .flat_map(|s| self.execute_stmt(s2, s))
                    .collect()
            }

            Stmt::If(c, then_branch, else_branch) => {
                let false_state = state.branch(c);

                // Execute both branches
                let mut then_states = self.execute_stmt(then_branch, state);
                let mut else_states = self.execute_stmt(else_branch, false_state);

                then_states.append(&mut else_states);
                then_states
            }
        }
    }
}
```

## 6.9 Conflict Detection

To detect bugs, we check for **Unreachable Code**. If a statement is unreachable (its path condition is False), it is dead code.

```
impl ProgramWalker {
    // In a real implementation, this would be part of execute_stmt
    fn check_reachability(&self, state: &SymbolicState) → Option<String> {
        if !state.is_feasible() {
            Some("Code is unreachable (Dead Code)".to_string())
        } else {
            None
        }
    }
}
```

### Example 6.1: Dead Code Detection

```
// Buggy program: Second check is redundant/dead
let buggy_prog = Stmt::Seq(
    Box::new(Stmt::If(
        Condition { var: Var::X, op: Op::Gt, val: 0 }, // x > 0
        Box::new(Stmt::Skip),
        Box::new(Stmt::Skip),
    )),
    Box::new(Stmt::If(
        Condition { var: Var::X, op: Op::Gt, val: 0 }, // x > 0 again!
        Box::new(Stmt::Error), // Unreachable in else branch of first check!
        Box::new(Stmt::Skip),
    )),
);

// The walker would find that the path to the second check's branches is constrained by
the first check.
```

## 6.10 Complete Example: Simple Program

Analyze a simple program end-to-end:

```
fn program_example() {
    // Program:
    // if x > 0 {
    //     y = 1;
    // } else {
    //     y = 2;
    // }

    let prog = Stmt::If(
        Condition { var: Var::X, op: Op::Gt, val: 0 },
        Box::new(Stmt::Assign(Var::Y, 1)),
        Box::new(Stmt::Assign(Var::Y, 2)),
    );

    let walker = ProgramWalker::new();
    let final_states = walker.execute(&prog);
}
```



```
println!("Number of final paths: {}", final_states.len());

for (i, state) in final_states.iter().enumerate() {
    println!("\nPath {}: ", i);
    println!("  Condition: {:?}", state.path);
    println!("  Variables:");
    for (var, val) in &state.env {
        println!("    {:?} = {}", var, val);
    }
}
}
```

Output:

```
Number of final paths: 2

Path 0:
  Condition: [BDD node representing x > 0]
  Variables: {Y: 1}

Path 1:
  Condition: [BDD node representing x ≤ 0]
  Variables: {Y: 2}
```

Two paths, both feasible, covering the entire input space.

## 6.11 Enhancements for Real Systems

This toy executor lacks features needed for production. **Interval Analysis** handles ranges efficiently. **Abstract domain integration** uses intervals or signs to refine paths. **SMT solver integration** checks complex arithmetic constraints (e.g., `x + y > 10`). **Loop handling** requires fixpoint iteration. **Function calls** demand inter-procedural analysis.

For production:

```
// Integration with abstract domain
struct RefinedState<D: AbstractDomain> {
    path: Ref,
    symbolic_env: HashMap<Var, Expr>,
    abstract_env: HashMap<Var, D>,
}
```



### Hands-On Example

A complete verifier implementation that integrates the Interval abstract domain with BDD-based path sensitivity. This example demonstrates how to prove numeric properties (like bounds and safety) that the basic symbolic executor cannot handle.

**Source:** `verifier.rs`  
**Run:** `cargo run --example verifier`

## Symbolic Execution vs Abstract Interpretation

### Symbolic Execution

- Explores paths explicitly (or with BDDs)
- Maintains precise symbolic variables
- Uses SMT solvers for feasibility
- Goal: Find specific input that crashes program

### Abstract Interpretation

- Over-approximates all paths
- Uses abstract domains (Intervals, Signs)
- Guaranteed termination
- Goal: Prove program correctness for **all** inputs

### Hybrid approach

- BDDs for control flow
- Abstract domains for data
- Best of both worlds

## 6.12 Practical Considerations

### 6.12.1 Path Explosion

Even with BDDs, deeply nested branches create explosion. Mitigation strategies:

- Bound exploration depth
- Prioritize paths (heuristics)
- Merge similar paths aggressively (*e.g.*, merge all error paths)

### 6.12.2 Variable Ordering

BDD size depends critically on the ordering of condition variables.

Strategies:

- Allocate variables in execution order
- Group related variables
- Use heuristics based on dependency graph

### 6.12.3 Performance

Symbolic execution is inherently expensive.

Optimization tips:

- Cache BDD operations (built-in)
- Prune infeasible paths early
- Use abstract domains to eliminate paths (*e.g.*, if range analysis proves `x > 10`, don't explore `x < 5` branch)

## 6.13 Real-World Applications

Symbolic execution sees widespread use in software verification.

Tools like **KLEE** and **Angr** use symbolic techniques to verify C/C++ binaries. They can detect buffer overflows, null pointer dereferences, and dead code. **Infer** uses abstract interpretation to find bugs in Java/C++ codebases at scale.

## 6.14 Summary

Built a simple symbolic execution engine:

- Condition and Statement language
- Symbolic state with BDD path conditions
- Program Walker exploring all paths
- Basic conflict detection

Key takeaways:

- BDDs compactly represent path conditions
- Symbolic values track variable modifications
- Branching creates multiple states
- Real systems integrate abstract domains and SMT solvers

Part I covered abstract interpretation, BDDs, and their combination.

**Path Exploration Strategies** The path explosion problem is fundamental. Different exploration strategies offer different trade-offs. See `path_exploration.rs` (`cargo run --example path_exploration`) for comparisons of DFS, BFS, and bounded depth strategies.

Part II dives deeper into mathematical foundations and advanced techniques.

### Chapter Summary

This chapter completed the journey from abstract theory to concrete implementation by building a full symbolic executor.

The **executor architecture** integrates control flow traversal with symbolic state manipulation. It maintains a worklist of basic blocks awaiting analysis, processing each by applying transfer functions that update both path conditions (BDDs) and data values (abstract domains).

At control flow **merge points**, the executor performs sound joins that preserve path sensitivity. Rather than collapsing distinct paths into a single approximate state,

it maintains the BDD-based partition that distinguishes execution contexts. This preserves precision while ensuring termination through widening operators.

The analysis **detects assertion violations** by checking whether the path condition at an assertion point allows states violating the assertion. When such states exist, the BDD encoding the infeasible paths proves the assertion must hold. When satisfiable states remain, we've found a potential bug with its exact path conditions.

**Practical challenges** include path explosion (exponential state space growth), variable ordering effects on BDD size, and engineering tradeoffs between precision and performance. Production systems address these through bounded unrolling, aggressive widening, and hybrid approaches combining symbolic execution with other techniques.

The fundamental insight remains: **BDDs enable tractable path-sensitivity** by compactly representing exponentially many execution paths. This bridges the theoretical ideal of complete precision with the practical necessity of polynomial-time operations.

# Part II

## Deep Dive



Part II provides rigorous mathematical foundations for abstract interpretation. We develop complete lattice theory, fixpoint theorems, Galois connections, and advanced analysis techniques with formal proofs and implementation guidance.

# Chapter 7

## Lattice Theory

## Foundations



Advanced

We develop the theory of complete lattices, fixpoint theorems, and Galois connections — the essential mathematical foundations for understanding program analysis rigorously.

### 7.1 Partial Orders and Lattices



#### Intuition

#### Ordering as Information

In daily life, “order” usually means size (bigger vs. smaller). In abstract interpretation, order means **precision** or **information**.

- $x \leq y$  means “ $x$  is more precise than  $y$ ”.
- $x$  contains **more specific information** than  $y$ .
- $y$  represents a **larger set of possibilities** (more uncertainty) than  $x$ .

Think of it as:

- $\perp$  (Bottom): “Impossible” (Perfect information, but contradictory).
- $x$ : “ $x$  is 5” (Very precise).
- $y$ : “ $x$  is Pos” (Less precise).
- $\top$  (Top): “ $x$  is any integer” (No information).

**Definition 8 (Partial Order)** A **partial order** is a relation  $\leq$  on a set  $L$  that is:

- **Reflexive**:  $\forall x \in L : x \leq x$
- **Transitive**:  $\forall x, y, z \in L : x \leq y \wedge y \leq z \Rightarrow x \leq z$
- **Antisymmetric**:  $\forall x, y \in L : x \leq y \wedge y \leq x \Rightarrow x = y$

We write  $(L, \leq)$  for a set  $L$  equipped with a partial order  $\leq$ . This is called a **partially ordered set** or **poset**.

The partial order is the **precision ordering** in abstract interpretation:  $x \leq y$  means “ $x$  is more precise than  $y$ ”, or “ $x$  approximates fewer concrete behaviors”.

### Example

#### Real-World Example: Integer Intervals

Consider the set of integer intervals. The ordering is subset inclusion (reversed for precision):  $A \leq B$  if the set of integers in  $A$  is a subset of  $B$ .

- $[5, 5]$  (Single value) is very precise.
- $[0, 10]$  (Small range) is less precise.
- $[-\infty, +\infty]$  (All integers) is the least precise ( $\top$ ).
- $\emptyset$  (No value) is the most precise ( $\perp$ ).

$$\emptyset \leq [5, 5] \leq [0, 10] \leq [-\infty, +\infty] \quad (22)$$

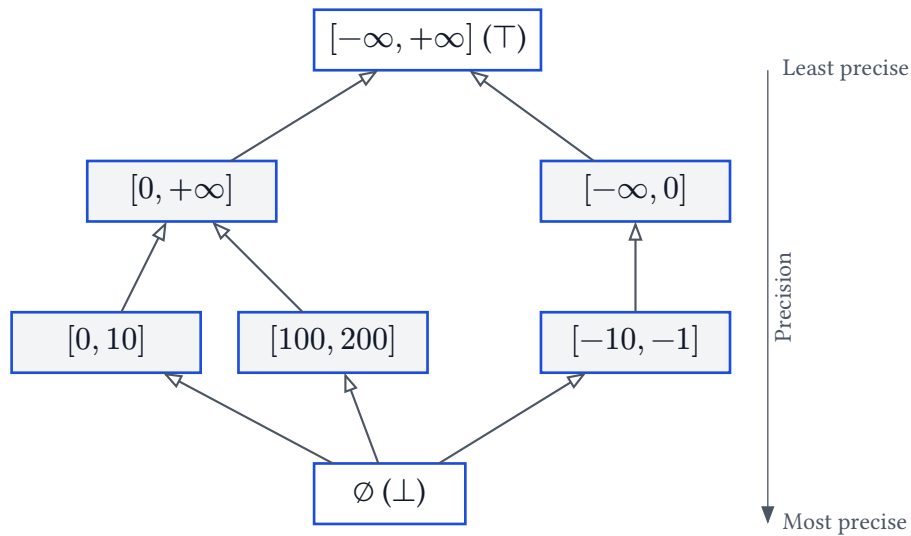


Figure 11: Lattice of Integer Intervals

### Example

#### Alternative Perspective: IP Subnets

For readers with a networking background, IP address blocks (CIDR) provide another intuitive example. The ordering is subset inclusion:

$$\emptyset \leq 192.168.1.5 \leq 192.168.1.0/24 \leq 0.0.0.0/0 \quad (23)$$

- 192.168.1.5 (Single IP) is very precise.
- 192.168.1.0/24 (Subnet) is less precise.
- 0.0.0.0/0 (The Internet) is the least precise ( $\top$ ).

**Definition 9 (Upper and Lower Bounds)** Let  $(L, \leq)$  be a poset and  $S \subseteq L$  be a subset.

- An element  $u \in L$  is an **upper bound** of  $S$  if  $\forall x \in S : x \leq u$ .
- An element  $l \in L$  is a **lower bound** of  $S$  if  $\forall x \in S : l \leq x$ .
- The **least upper bound** (lub) or **supremum** of  $S$ , denoted  $\sup S$  or  $\sqcup S$ , is the smallest element that is an upper bound of  $S$  (if it exists).
- The **greatest lower bound** (glb) or **infimum** of  $S$ , denoted  $\inf S$  or  $\sqcap S$ , is the largest element that is a lower bound of  $S$  (if it exists).

These operations correspond to **join** ( $\sqcup$ ) and **meet** ( $\sqcap$ ) in abstract interpretation. Join computes the least precise abstraction containing both inputs (over-approximation). Meet finds the most precise common refinement.

**Definition 10 (Lattice)** A poset  $(L, \leq)$  is a **lattice** if every pair of elements  $x, y \in L$  has both:

- A least upper bound:  $x \sqcup y$
- A greatest lower bound:  $x \sqcap y$

We write  $(L, \leq, \sqcup, \sqcap)$  or simply  $L$  when the operations are clear.

In Rust, we capture this structure with the `AbstractDomain` trait:

```
pub trait AbstractDomain: Clone + PartialEq + Debug {
    // The lattice operations
    fn bottom() → Self;
    fn top() → Self;
    fn join(&self, other: &Self) → Self;
    fn meet(&self, other: &Self) → Self;

    // The partial order
    fn is_less_or_equal(&self, other: &Self) → bool {
        //  $x \leq y$  iff  $x \sqcup y = y$ 
        self.join(other) == *other
    }
}
```

## Example

### Sign Lattice Join Table:

Consider the Sign lattice. The table below shows the result of the join operation ( $\sqcup$ ) for every pair of elements:

$\sqcup$	$\perp$	Pos	Neg	$\top$
$\perp$	$\perp$	Pos	Neg	$\top$



<b>Pos</b>	Pos	Pos	$\top$	$\top$
<b>Neg</b>	Neg	$\top$	Neg	$\top$
<b><math>\top</math></b>	$\top$	$\top$	$\top$	$\top$

Note that  $\text{Pos} \sqcup \text{Neg} = \top$  because there's no single sign that is both **Pos** and **Neg** (except  $\top$  which covers both).

### Example

#### Alternative: Protocol Lattice

In network analysis, protocols form a similar flat lattice:

- $\perp$  (No protocol)
- TCP, UDP, ICMP (Concrete protocols)
- $\top$  (Any protocol)

Since a packet cannot be both TCP and UDP simultaneously,  $\text{TCP} \sqcup \text{UDP} = \top$ .

**Definition 11 (Complete Lattice)** A lattice  $(L, \leq, \sqcup, \sqcap)$  is **complete** if every subset  $S \subseteq L$  (including infinite subsets) has both:

- A least upper bound:  $\bigsqcup_{x \in S} x$
- A greatest lower bound:  $\bigsqcap_{x \in S} x$

In particular, a complete lattice has:

- A **least element**  $\perp = \bigsqcap_{x \in L} x$  (bottom).
- A **greatest element**  $\top = \bigsqcup_{x \in L} x$  (top).

Complete lattices are the fundamental structure for abstract interpretation. Program analysis must handle unbounded sets of states and infinite chains during iteration.

**Theorem 1 (Properties of Complete Lattices)** Let  $(L, \leq, \sqcup, \sqcap, \perp, \top)$  be a complete lattice.

1. **Idempotence:**  $x \sqcup x = x$  and  $x \sqcap x = x$ .
2. **Commutativity:**  $x \sqcup y = y \sqcup x$  and  $x \sqcap y = y \sqcap x$ .
3. **Associativity:**  $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$ .
4. **Absorption:**  $x \sqcup (x \sqcap y) = x$  and  $x \sqcap (x \sqcup y) = x$ .
5. **Identity:**  $x \sqcup \perp = x$  and  $x \sqcap \top = x$ .
6. **Annihilation:**  $x \sqcup \top = \top$  and  $x \sqcap \perp = \perp$ .

**Proof.** We prove a representative subset.

**Idempotence of  $\sqcup$ :** Since  $x \leq x$ , we have  $x$  is an upper bound of  $\{x, x\}$ . For any other upper bound  $u$  with  $x \leq u$ , we have  $x \leq u$ . Thus  $x$  is the least upper bound, so  $x \sqcup x = x$ .

**Absorption of  $\sqcup$ :** Since  $x \sqcap y \leq x$ , we have  $x$  is an upper bound of  $\{x, x \sqcap y\}$ . For any upper bound  $u$  with  $x \leq u$  and  $x \sqcap y \leq u$ , we have  $x \leq u$ . Thus  $x \sqcup (x \sqcap y) = x$ .

Other properties follow similarly from the definitions. ■

## 7.2 Height and Chains

**Definition 12 (Chain)** A subset  $C \subseteq L$  is a **chain** if every two elements are comparable:

$$\forall x, y \in C : x \leq y \vee y \leq x \quad (24)$$

The **length** of a finite chain  $x_0 < x_1 < \dots < x_n$  is  $n$  (number of strict comparisons).

Chains are important because program analysis iterates along chains in the lattice. We refine approximations until reaching a fixpoint.

**Definition 13 (Height)** The **height** of a poset  $(L, \leq)$ , denoted  $\text{height}(L)$ , is the length of the longest chain in  $L$ . If arbitrarily long chains exist, the height is infinite.

### Example

#### Heights of common lattices:

- Sign lattice:  $\text{height} = 2$  (chain:  $\perp < \text{Pos} < \top$ ).
- Boolean lattice:  $\text{height}(\{\perp, \top\}) = 1$ .
- Powerset lattice:  $\text{height}(\mathcal{P}(S)) = |S|$  for finite set  $S$ .
- Interval lattice over  $\mathbb{Z}$ :  $\text{height}(\text{Interval}) = \infty$  (unbounded chains).

**Theorem 2 (Ascending Chain Condition)** A poset  $(L, \leq)$  satisfies the **ascending chain condition** (ACC) if every increasing chain

$$x_0 \leq x_1 \leq x_2 \leq \dots \quad (25)$$

eventually stabilizes. There exists  $N$  such that  $x_N = x_{N+1} = x_{N+2} = \dots$

**Why ACC Matters** The ascending chain condition ensures that fixpoint iteration terminates. Without ACC, analysis might iterate forever, refining approximations without converging.

When ACC doesn't hold (infinite-height lattices), we need **widening** operators to enforce convergence.

## 7.3 Monotone Functions

**Definition 14 (Monotone Function)** A function  $f : L \rightarrow M$  between posets is **monotone** (or **order-preserving**) if:

$$\forall x, y \in L : x \leq y \Rightarrow f(x) \leq f(y) \quad (26)$$

Intuitively, increasing precision in the input increases precision in the output.

Abstract operations in program analysis must be monotone to ensure sound approximation. If analysis loses precision when given more precise inputs, something is wrong.

### Example

#### Monotone functions on Intervals:

Abstract “absolute value” is monotone:

- If  $x_1 \leq x_2$  (e.g.,  $[0, 5] \leq [0, 10]$ ), then  $\text{abs}(x_1) \leq \text{abs}(x_2)$ .
- Example:  $\text{abs}([0, 5]) = [0, 5] \leq \text{abs}([0, 10]) = [0, 10]$ .

But consider a **non-monotone** function that returns 0 for  $\perp$  and  $[0, 5]$ , but 100 for  $[0, 10]$ . Then  $[0, 5] \leq [0, 10]$  but  $f([0, 5]) = 0 \not\leq 100 = f([0, 10])$ , violating monotonicity.

**Theorem 3 (Composition Preserves Monotonicity)** If  $f : L \rightarrow M$  and  $g : M \rightarrow N$  are monotone, then  $g \circ f : L \rightarrow N$  is monotone.

**Proof.** Let  $x \leq y$  in  $L$ . Since  $f$  is monotone,  $f(x) \leq f(y)$  in  $M$ . Since  $g$  is monotone,  $g(f(x)) \leq g(f(y))$  in  $N$ . Thus  $(g \circ f)(x) \leq (g \circ f)(y)$ . ■

## 7.4 Fixpoints and Tarski's Theorem

**Definition 15 (Fixpoint)** Let  $f : L \rightarrow L$  be a function on a poset. An element  $x \in L$  is a **fixpoint** of  $f$  if  $f(x) = x$ .

The set of all fixpoints is  $\text{Fix}(f) = \{x \in L \mid f(x) = x\}$ .

- A **least fixpoint** is  $\text{lfp}(f) = \bigcap_{x \in \text{Fix}(f)} x$ .
- A **greatest fixpoint** is  $\text{gfp}(f) = \bigcup_{x \in \text{Fix}(f)} x$ .

Fixpoints represent stable abstract states: applying the transfer function doesn't change the approximation.

**Theorem 4 (Tarski's Fixpoint Theorem)** Let  $(L, \leq, \sqcup, \sqcap, \perp, \top)$  be a complete lattice and  $f : L \rightarrow L$  be monotone. Then:

1.  $f$  has a least fixpoint:  $\text{lfp}(f) = \bigcap_{f(x) \leq x} x$ .
2.  $f$  has a greatest fixpoint:  $\text{gfp}(f) = \bigcup_{x \leq f(x)} x$ .
3. The set of all fixpoints  $\text{Fix}(f)$  forms a complete lattice.

This is the foundation of fixpoint iteration in abstract interpretation!

**Proof.** We prove existence of the least fixpoint (greatest is dual).

Let  $P = \{x \in L \mid f(x) \leq x\}$  be the set of **post-fixpoints**.

$P$  is non-empty:  $L$  has a top element  $\top$ , and  $f(\top) \leq \top$  (anything is below  $\top$ ), so  $\top \in P$ .

Let  $\mu = \bigcap_{x \in P} x$  be the greatest lower bound of  $P$ . We show  $f(\mu) = \mu$ .

First, show  $f(\mu) \leq \mu$ : For all  $x \in P$ , we have  $\mu \leq x$ . By monotonicity of  $f$ :  $f(\mu) \leq f(x)$ . Since  $x \in P$ , we have  $f(x) \leq x$ . Thus  $f(\mu) \leq x$  for all  $x \in P$ . Therefore  $f(\mu)$  is a lower bound of  $P$ , so  $f(\mu) \leq \bigcap_{x \in P} x = \mu$ .

Next, show  $\mu \leq f(\mu)$ : From  $f(\mu) \leq \mu$ , we have  $\mu \in P$  (it's a post-fixpoint). Applying  $f$ :  $f(\mu) \leq f(\mu)$  (trivially). By monotonicity:  $f(f(\mu)) \leq f(\mu)$ , so  $f(\mu) \in P$ . Since  $\mu = \bigcap_{x \in P} x$  is the greatest lower bound,  $\mu \leq f(\mu)$ .

By antisymmetry,  $f(\mu) = \mu$ , so  $\mu$  is a fixpoint.

$\mu$  is the least fixpoint: any fixpoint  $x$  satisfies  $f(x) = x$ , hence  $f(x) \leq x$ , so  $x \in P$ . Thus  $\mu = \bigcap_{x \in P} x \leq x$  for all fixpoints  $x$ . ■

**Constructive Fixpoint** Tarski's theorem tells us the fixpoint exists, but doesn't give an algorithm to compute it. The **Kleene fixpoint theorem** (next section) provides a constructive iteration method.

## 7.5 Kleene Fixpoint Theorem and Iteration

**Theorem 5 (Kleene Fixpoint Theorem)** Let  $(L, \leq)$  be a complete lattice with finite height, and  $f : L \rightarrow L$  be monotone. Then the least fixpoint can be computed by iteration:

$$\text{lfp}(f) = \bigsqcup_{i \geq 0} f^i(\perp) \quad (27)$$

where  $f^0(\perp) = \perp$  and  $f^{i+1}(\perp) = f(f^i(\perp))$ .

Moreover, the sequence stabilizes after finitely many steps. There exists  $N$  such that  $f^N(\perp) = f^{N+1}(\perp) = \text{lfp}(f)$ .

This is the standard fixpoint iteration algorithm used in dataflow analysis.

**Proof.** Define the **Kleene sequence**:  $x_0 = \perp, x_{i+1} = f(x_i)$ .

**Step 1:** The sequence is increasing. By induction on  $i$ :

- Base:  $x_0 = \perp \leq f(\perp) = x_1$  (bottom is below everything).
- Step: Assume  $x_i \leq x_{i+1}$ . By monotonicity:  $f(x_i) \leq f(x_{i+1})$ , i.e.,  $x_{i+1} \leq x_{i+2}$ .

**Step 2:** The sequence stabilizes. Since  $L$  has finite height and the sequence is increasing, it must eventually stabilize at some  $x_N = x_{N+1}$ .

**Step 3:**  $x_N$  is a fixpoint.  $f(x_N) = f(x_N) = x_{N+1} = x_N$ .

**Step 4:**  $x_N$  is the least fixpoint. Let  $\mu$  be any fixpoint. By induction,  $x_i \leq \mu$  for all  $i$ :

- Base:  $x_0 = \perp \leq \mu$ .
- Step: Assume  $x_i \leq \mu$ . Then  $x_{i+1} = f(x_i) \leq f(\mu) = \mu$  (by monotonicity).

Thus  $x_N \leq \mu$ , so  $x_N$  is the least fixpoint. ■

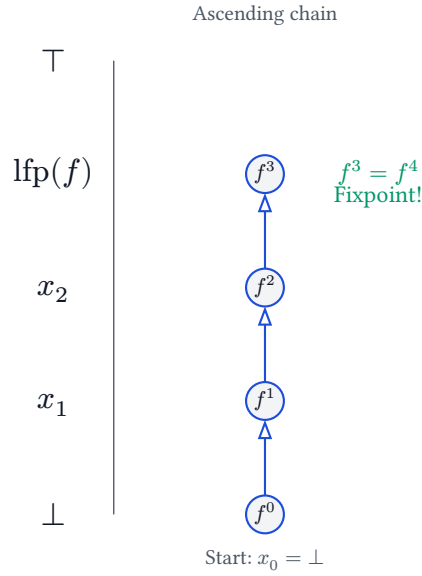


Figure 12: Kleene iteration ascending from bottom to least fixpoint

**Example****Computing reachable program states:**

Consider a program with states  $\{\text{Init}, \text{Loop}, \text{Exit}\}$ . Let  $f(S) = S \cup \text{next\_state}(S)$ .

Starting from  $S_0 = \{\text{Init}\}$ :

- $f^0(\{\text{Init}\}) = \{\text{Init}\}$ .
- $f^1(\{\text{Init}\}) = \{\text{Init}, \text{Loop}\}$ .
- $f^2(\{\text{Init}\}) = \{\text{Init}, \text{Loop}, \text{Exit}\}$ .
- $f^3(\{\text{Init}\}) = \{\text{Init}, \text{Loop}, \text{Exit}\}$  (no new states).

Fixpoint reached after 3 iterations!

## 7.6 Galois Connections

Galois connections relate concrete and abstract domains, ensuring abstraction and concretization are consistent.

**Definition 16 (Galois Connection)** Let  $(C, \leq_C)$  and  $(A, \leq_A)$  be posets. A pair of monotone functions

$$\alpha : C \rightarrow A \quad \text{and} \quad \gamma : A \rightarrow C \quad (28)$$

forms a **Galois connection**, written  $(C, \alpha, \gamma, A)$  or  $C \xrightarrow[\gamma]{\alpha} A$ , if:

$$\forall c \in C, \forall a \in A : \alpha(c) \leq_A a \iff c \leq_C \gamma(a) \quad (29)$$

- $\alpha$  is the **abstraction function** (concrete  $\rightarrow$  abstract).
- $\gamma$  is the **concretization function** (abstract  $\rightarrow$  concrete).

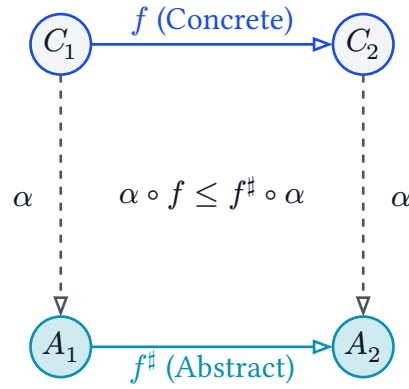


Figure 13: Concrete vs. Abstract Execution: The Commuting Diagram

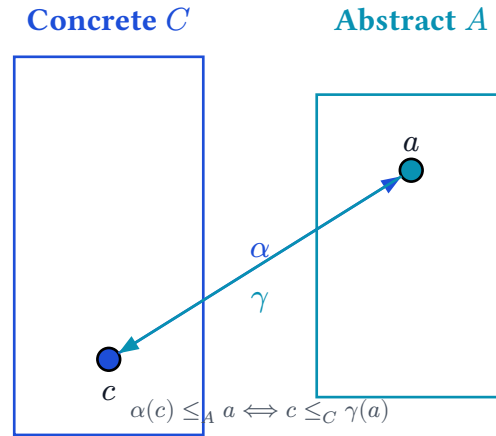


Figure 14: Galois connection between concrete and abstract domains

**Theorem 6 (Properties of Galois Connections)** Let  $(C, \alpha, \gamma, A)$  be a Galois connection.

1.  $\alpha$  and  $\gamma$  are monotone.
2.  $\alpha \circ \gamma$  is **reductive**:  $\alpha(\gamma(a)) \leq_A a$  for all  $a \in A$ .
3.  $\gamma \circ \alpha$  is **extensive**:  $c \leq_C \gamma(\alpha(c))$  for all  $c \in C$ .
4.  $\alpha(\gamma(\alpha(c))) = \alpha(c)$  (abstraction idempotent).
5.  $\gamma(\alpha(\gamma(a))) = \gamma(a)$  (concretization idempotent).

**Proof.** We prove properties 2 and 3 (others are exercises).

**Property 2** ( $\alpha \circ \gamma$  reductive): By definition of Galois connection with  $c = \gamma(a)$  and  $a = a$ :

$$\alpha(\gamma(a)) \leq_A a \iff \gamma(a) \leq_C \gamma(a) \quad (30)$$

The right side is trivially true, so  $\alpha(\gamma(a)) \leq_A a$ .

**Property 3** ( $\gamma \circ \alpha$  extensive): By definition with  $c = c$  and  $a = \alpha(c)$ :

$$\alpha(c) \leq_A \alpha(c) \iff c \leq_C \gamma(\alpha(c)) \quad (31)$$

The left side is trivially true, so  $c \leq_C \gamma(\alpha(c))$ . ■

### Example

#### Sign abstraction as Galois connection:

Concrete domain:  $C = \mathcal{P}(\mathbb{Z})$  (powersets of integers). Abstract domain:  $A = \{\text{Bot}, \text{Neg}, \text{Zero}, \text{Pos}, \text{Top}\}$ .

Abstraction  $\alpha : \mathcal{P}(\mathbb{Z}) \rightarrow A$ :

$$\alpha(S) = \begin{cases} \text{Bot} & \text{if } S = \emptyset \\ \text{Neg} & \text{if } S \subseteq \mathbb{Z}^- \\ \text{Zero} & \text{if } S = \{0\} \\ \text{Pos} & \text{if } S \subseteq \mathbb{Z}^+ \\ \text{Top} & \text{otherwise} \end{cases} \quad (32)$$

Concretization  $\gamma : A \rightarrow \mathcal{P}(\mathbb{Z})$ :

$$\gamma(a) = \begin{cases} \emptyset & \text{if } a = \text{Bot} \\ \mathbb{Z}^- & \text{if } a = \text{Neg} \\ \{0\} & \text{if } a = \text{Zero} \\ \mathbb{Z}^+ & \text{if } a = \text{Pos} \\ \mathbb{Z} & \text{if } a = \text{Top} \end{cases} \quad (33)$$



Verify adjunction:  $\alpha(S) \leq_A a \iff S \subseteq \gamma(a)$ .

## 7.7 The Lattice of Boolean Functions

Since we use BDDs to represent sets of paths, it is crucial to understand that Boolean functions form a lattice.

Let  $B_n$  be the set of all Boolean functions  $f : \{0, 1\}^n \rightarrow \{0, 1\}$ . We define the ordering  $f \leq g$  as logical implication:

$$f \leq g \iff \forall x : f(x) \Rightarrow g(x) \quad (34)$$

This forms a complete lattice  $(B_n, \leq, \vee, \wedge, \text{false}, \text{true})$ .

- Join ( $\sqcup$ ) is logical OR ( $\vee$ ).
- Meet ( $\sqcap$ ) is logical AND ( $\wedge$ ).
- Bottom ( $\perp$ ) is the constant function `false` (empty set of paths).
- Top ( $\top$ ) is the constant function `true` (all paths).

**Key Insight** BDDs implement this lattice. The BDD operations `apply_or` and `apply_and` correspond exactly to the lattice join and meet operations. This is why we can use BDDs directly in our abstract interpretation framework.

### Chapter Summary

This chapter established the mathematical foundations that make abstract interpretation both rigorous and practical.

**Complete lattices** provide the algebraic structure underlying abstract domains, ensuring every set of approximations has both a least upper bound (join) and greatest lower bound (meet). This structure enables systematic combination of information from multiple execution paths.

**Monotone functions** preserve the precision ordering, ensuring that abstract transformers never lose information in unexpected ways. This monotonicity property is essential for soundness: if we start with a safe approximation, applying a monotone function maintains that safety.

**Tarski's fixpoint theorem** provides the existence guarantee for program analysis, proving that every monotone function over a complete lattice has both a least and greatest fixpoint. This theorem justifies the search for least fixpoints as optimal solutions to dataflow equations.

**Kleene iteration** transforms existence into construction, providing a concrete algorithm that computes least fixpoints by iterating from the bottom element. The ascending chain condition ensures this iteration terminates after finitely many steps for well-designed abstract domains.

**Galois connections** formalize the relationship between concrete and abstract semantics through adjoint abstraction and concretization functions. This framework enables proving soundness compositionally and reasoning about precision loss systematically. Together, these tools enable rigorous program analysis with guaranteed termination, soundness, and predictable precision characteristics.

# Chapter 8

## Fixpoint Algorithms



Advanced

Tarski's theorem guarantees fixpoint existence and Kleene's theorem provides a basic iteration method, but practical program analysis requires more sophisticated algorithms. This chapter explores chaotic iteration, worklist algorithms, and strategies for computing fixpoints efficiently.

### 8.1 From Kleene to Chaotic Iteration

Kleene iteration computes  $f^0(\perp), f^1(\perp), f^2(\perp), \dots$  sequentially. For compositional problems (multiple equations), we can interleave updates — this is **chaotic iteration**.

**Definition 17 (Equation System)** A **system of equations** over a complete lattice  $L$  consists of:

- Variables  $X = \{x_1, \dots, x_n\}$ .
- Equations  $x_i = f_i(x_1, \dots, x_n)$  where  $f_i : L^n \rightarrow L$  are monotone.

A **solution** is an assignment  $\sigma : X \rightarrow L$  satisfying all equations. The **least solution** is the least fixpoint of the combined function  $F : L^n \rightarrow L^n$  defined by  $F(\vec{x}) = (f_1(\vec{x}), \dots, f_n(\vec{x}))$ .

#### Example

##### Real-World Example: Basic Block Reachability

Consider a Control Flow Graph (CFG) with basic blocks. For each block  $B$ , let  $\text{Reachable}[B]$  be the set of program states that can reach it.

$$\text{Reachable}[B] = \bigcup_{P \in \text{pred}(B)} (\text{Reachable}[P] \cap \text{Edge}[P \rightarrow B]) \quad (35)$$

This gives one equation per block.

- Kleene iteration updates all blocks in lockstep.
- Chaotic iteration updates blocks as soon as their predecessors change.
- If Block A feeds Block B, updating A then B is faster than updating B then A.

**Theorem 7 (Chaotic Iteration Convergence)** Let  $X = \{x_1, \dots, x_n\}$  be variables with equations  $x_i = f_i(\vec{x})$  over a complete lattice with ascending chain condition. Let  $\sigma^0, \sigma^1, \sigma^2, \dots$  be a sequence of assignments where:

- $\sigma^0(x_i) = \perp$  for all  $i$ .
- $\sigma^{k+1}$  updates **some** variable:  $\sigma^{k+1}(x_i) = f_i(\sigma^k(x_1), \dots, \sigma^k(x_n))$  for some  $i$ , and  $\sigma^{k+1}(x_j) = \sigma^k(x_j)$  for  $j \neq i$ .
- Every variable is updated **infinitely often**.

Then the sequence converges to the least fixpoint.

As long as we keep updating variables (fairness), the order doesn't matter.

**Proof.** Define the **product lattice**  $L^n$  with ordering  $(x_1, \dots, x_n) \leq (y_1, \dots, y_n)$  iff  $x_i \leq y_i$  for all  $i$ .

**Step 1:** The sequence is increasing. Each update  $\sigma^{k+1}(x_i) = f_i(\sigma^k)$  satisfies:

- $\sigma^{k+1}(x_i) \leq f_i(\sigma^k)$  (by induction: initially  $\perp \leq f_i(\perp)$ , and if all components increased, monotonicity gives  $f_i(\sigma^k) \leq f_i(\sigma^{k+1})$ ).
- Thus  $\sigma^k \leq \sigma^{k+1}$  component-wise.

**Step 2:** The sequence stabilizes. Since  $L^n$  has ACC (product of ACC lattices), the increasing sequence stabilizes at some  $\sigma^*$ .

**Step 3:**  $\sigma^*$  is a fixpoint. For each  $i$ , since  $x_i$  is updated infinitely often and the sequence stabilized:

$$\sigma^*(x_i) = f_i(\sigma^*) \quad (36)$$

**Step 4:**  $\sigma^*$  is the least fixpoint. Any fixpoint  $\mu$  satisfies  $\sigma^0 \leq \mu$  (base case). If  $\sigma^k \leq \mu$ , then  $\sigma^{k+1}(x_i) = f_i(\sigma^k) \leq f_i(\mu) = \mu(x_i)$  by monotonicity. Thus  $\sigma^* \leq \mu$ . ■

## 8.2 Worklist Algorithms

Chaotic iteration can be wasteful: updating a variable when its inputs haven't changed accomplishes nothing. **Worklist algorithms** track dependencies and only recompute when necessary.

### Algorithm 2: Basic Worklist

**Input:** Equation system  $x_i = f_i(\vec{x})$  with dependency graph  $G$ .

**Output:** Least fixpoint solution.

- 1  $W \leftarrow \{x_1, \dots, x_n\}$  // Initialize worklist with all variables.
- 2  $\sigma \leftarrow \lambda i. \perp$  // Initialize solution to bottom.

```

3   while  $W \neq \emptyset$  do
4        $x \leftarrow \text{remove}(W)$     // Pick a variable from worklist.
5        $\text{old} \leftarrow \sigma(x)$ 
6        $\text{new} \leftarrow f_x(\sigma)$  // Recompute using current solution.
7       if  $\text{new} \neq \text{old}$  then
8            $\sigma(x) \leftarrow \text{new}$  // Update solution.
9           for each  $y$  where  $x \rightarrow y \in G$  do
10               $\text{add}(W, y)$  // Re-examine dependents.
11 return  $\sigma$ 

```

A generic worklist solver in Rust:

```

use std::collections::{VecDeque, HashSet};
use std::hash::Hash;

pub fn solve_worklist<K, D, F>(
    initial_worklist: Vec<K>,
    mut get_deps: F,
    mut transfer: impl FnMut(&K, &D) → D,
    bottom: D
) → HashMap<K, D>
where
    K: Eq + Hash + Clone,
    D: AbstractDomain,
    F: FnMut(&K) → Vec<K>, // Returns dependents of a node
{
    let mut state: HashMap<K, D> = HashMap::new();
    let mut worklist: VecDeque<K> = initial_worklist.into();
    let mut in_worklist: HashSet<K> = worklist.iter().cloned().collect();

    while let Some(node) = worklist.pop_front() {
        in_worklist.remove(&node);

        let old_val = state.get(&node).unwrap_or(&bottom).clone();
        let new_val = transfer(&node, &old_val);

        if new_val != old_val {
            // State changed! Update and notify dependents
            state.insert(node.clone(), new_val);

            for dep in get_deps(&node) {
                if !in_worklist.contains(&dep) {
                    worklist.push_back(dep.clone());
                    in_worklist.insert(dep);
                }
            }
        }
    }
    state
}

```

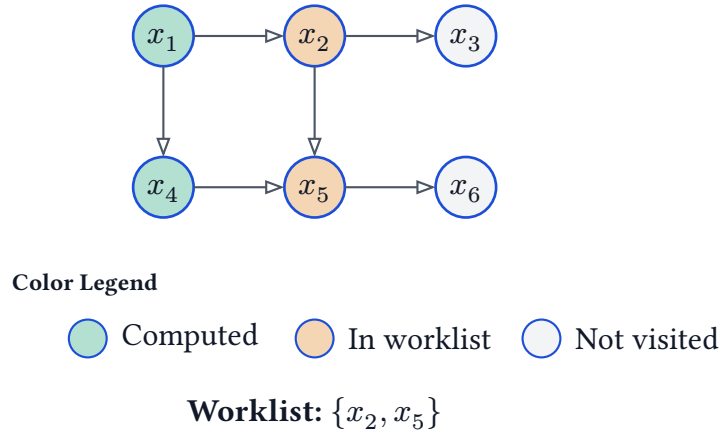


Figure 15: Worklist algorithm tracking dependencies

**Theorem 8 (Worklist Correctness)** The worklist algorithm computes the least fixpoint if:

1. Each variable is processed when added to the worklist.
2. Dependencies are correctly tracked.
3. The lattice satisfies ACC.

Proof follows chaotic iteration: only variables whose inputs changed need recomputation.

## 8.3 Iteration Strategies

The order of processing the worklist affects performance significantly.

**Definition 18 (Iteration Order)** Common strategies for choosing the next variable:

- **FIFO** (breadth-first): Process in insertion order.
- **LIFO** (depth-first): Process most recently added first.
- **RPO** (reverse postorder): Process following CFG structure.
- **Priority queue**: Process by estimated impact.

### Example

#### Reverse postorder for forward dataflow:

For forward analyses (information flows with CFG edges), reverse postorder minimizes recomputation.

Given CFG with postorder  $p_n, \dots, p_1$ , process in order  $p_1, \dots, p_n$ . This ensures each node is processed after most of its predecessors.

For reducible CFGs (no irreducible loops), this converges in  $O(d)$  passes where  $d$  is loop nesting depth.

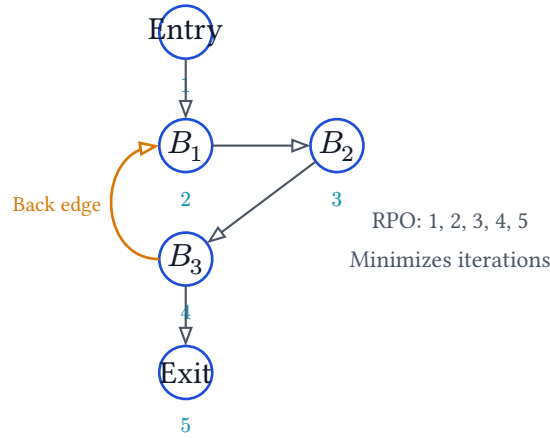


Figure 16: Iteration orders on control flow graph

## 8.4 Complexity Analysis

**Theorem 9 (Worklist Complexity)** For a system with  $n$  variables and maximum dependency degree  $d$ :

- **Time per iteration:**  $O(nd)$  (evaluate each function once).
- **Number of iterations:**  $O(h \cdot n)$  where  $h$  is lattice height.
- **Total time:**  $O(h \cdot n^2 \cdot d)$ .

For **forward dataflow on reducible CFGs** with RPO:

- Iterations:  $O(\text{depth of loops})$ .
- Total:  $O(\text{loop depth} \cdot n \cdot d)$ .

Good iteration order reduces the constant factor dramatically.

**Proof.** Each variable can increase at most  $h$  times (lattice height). With  $n$  variables, total number of increases is  $\leq h \cdot n$ .

Each increase causes at most  $d$  dependents to be re-examined. Thus total work is  $O(h \cdot n \cdot d)$  updates.

Each update costs  $O(n)$  to evaluate the function (in general). Total:  $O(h \cdot n^2 \cdot d)$ .

For RPO on reducible graphs, structural properties ensure  $O(\text{loop depth})$  passes suffice, improving the constant. ■

## 8.5 Widening with Worklist

For infinite-height lattices, combine worklist with widening.

### Algorithm 3: Worklist with Widening

```

1  $W \leftarrow \{x_1, \dots, x_n\}$ 
2  $\sigma \leftarrow \lambda i. \perp$ 
3  $\text{count} \leftarrow \lambda i. 0$  // Track update count per variable.
4 while  $W \neq \emptyset$  do
5    $x \leftarrow \text{remove}(W)$ 
6    $\text{old} \leftarrow \sigma(x)$ 
7    $\text{new} \leftarrow f_x(\sigma)$ 
8    $\text{count}(x) \leftarrow \text{count}(x) + 1$ 
9   if  $\text{count}(x) > \text{threshold}$  then
10     $\sqcup \text{new} \leftarrow \text{old} \nabla \text{new}$  // Apply widening after threshold.
11    if  $\text{new} \neq \text{old}$  then
12       $\sigma(x) \leftarrow \text{new}$ 
13     $\sqcup$  add dependents to  $W$ 
14 return  $\sigma$ 

```

Extend the `AbstractDomain` trait with a `widen` method:

```

pub trait AbstractDomain: Clone + PartialEq + Debug {
  // ... existing methods ...

  // Default implementation: just join (for finite height lattices)
  fn widen(&self, other: &Self) → Self {
    self.join(other)
  }
}

// Example: Interval widening
impl AbstractDomain for Interval {
  fn widen(&self, other: &Self) → Self {
    // If bound is unstable, jump to infinity
    let new_min = if other.min < self.min { NegInf } else { self.min };
    let new_max = if other.max > self.max { PosInf } else { self.max };
    Interval::new(new_min, new_max)
  }
}

```

Typical threshold: 2-3 iterations before widening kicks in.

### Example

#### Loop Counter Analysis with Widening:



Consider a loop where a counter is decremented: `i = 64; while (i > 0) { i = i - 1; }`

Abstract domain: Intervals for `i`. Without widening:

- Iteration 1:  $i \in [64, 64]$
- Iteration 2:  $i \in [63, 64]$
- Iteration 3:  $i \in [62, 64]$
- ... (64 iterations)

With widening (threshold = 2):

- Iteration 1:  $i \in [64, 64]$
- Iteration 2:  $i \in [63, 64]$
- Iteration 3:  $[63, 64] \nabla [62, 64] = [-\infty, 64]$  (stabilized!)

## 8.6 Narrowing Iterations

After widening converges to a post-fixpoint, narrow to recover precision.

### Algorithm 4: Two-Phase Worklist

```

1 // Phase 1: Widening (ascending).
2  $\sigma_{\text{up}} \leftarrow \text{widening\_iteration}()$ 
3 // Phase 2: Narrowing (descending).
4  $W \leftarrow \{x_1, \dots, x_n\}$ 
5  $\sigma \leftarrow \sigma_{\text{up}}$ 
6 while  $W \neq \emptyset$  and not\_too\_many\_iterations() do
7    $x \leftarrow \text{remove}(W)$ 
8    $\text{old} \leftarrow \sigma(x)$ 
9    $\text{new} \leftarrow f_x(\sigma)$ 
10   $\text{narrow} \leftarrow \text{old} \triangle \text{new}$  // Apply narrowing.
11  if  $\text{narrow} \neq \text{old}$  then
12     $\sigma(x) \leftarrow \text{narrow}$ 
13   $\text{add dependents to } W$ 
14 return  $\sigma$ 

```

Narrowing is optional but often improves precision significantly.

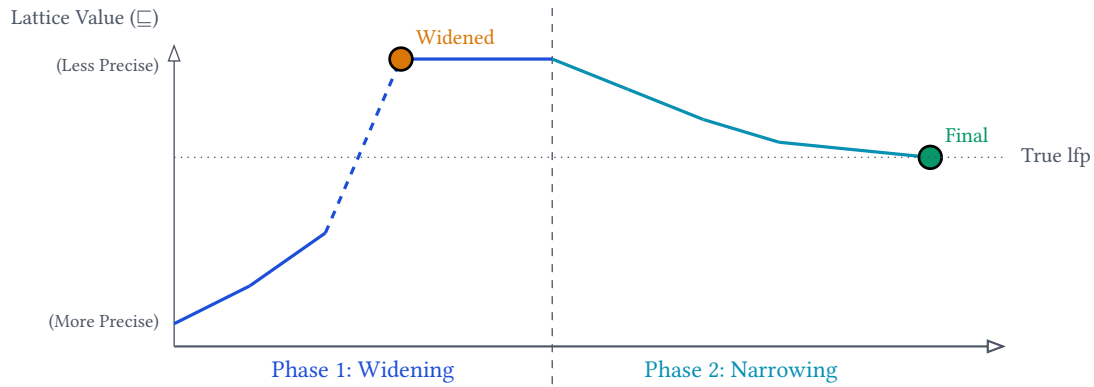


Figure 17: Two-phase fixpoint computation with widening and narrowing

## 8.7 Delayed Widening

Applying widening too early loses precision unnecessarily. **Delayed widening** waits for the analysis to explore “natural” convergence first.

**Definition 19 (Widening Delay)** Strategies for delaying widening:

- **Threshold-based:** Apply widening only after  $k$  updates to a variable.
- **Loop-based:** Apply widening only at loop heads.
- **Heuristic:** Use domain-specific knowledge to identify widening points.

### Example

#### Bounded Loop Processing:

```
// Process up to 10 items
for i in 0..10 {
  process_item(data, i);
}
```

Without delay: After 2 iterations, widen  $i$  to  $[0, +\infty]$ , losing the bound information ( $i < 10$ ).

With delay (threshold = 5): Natural convergence to  $[0, 10]$  before widening triggers, preserving the precise bound.

## Chapter Summary

This chapter bridged the gap between fixpoint theory and practical implementation by developing algorithms that compute least fixpoints efficiently on real programs.

**Chaotic iteration** generalizes Kleene’s sequential approach by allowing flexible update orders while preserving convergence guarantees through fairness conditions. This freedom enables **worklist algorithms** that exploit program structure by tracking dependencies explicitly, ensuring each computation step processes only variables whose inputs have changed.

**Iteration strategies** determine the order of worklist processing, with dramatic performance implications. Reverse postorder traversal aligns with program control flow, enabling rapid propagation through acyclic regions. Queue-based (FIFO) and stack-based (LIFO) strategies offer simpler alternatives with different propagation characteristics.

For infinite-height lattices where natural convergence is impossible, **widening operators** force termination by extrapolating ascending chains to stable upper bounds. The complementary **narrowing phase** refines these approximations in a descending iteration, recovering precision lost to aggressive widening. **Delayed widening** further improves results by deferring extrapolation until natural convergence stalls, preserving precise invariants when they arise quickly.

These techniques form the algorithmic backbone of practical program analyzers, transforming abstract interpretation from theoretical framework to deployable verification technology.

# Chapter 9

## Advanced Galois Connections



Advanced

Section 7 introduced Galois connections as the mathematical foundation relating concrete and abstract domains. We now explore advanced topics: abstract transformers, best transformers, and completeness — essential tools for designing precise and efficient analyses. For domain combination techniques including products and reduction operators, see Section 11.

### 9.1 From Concrete to Abstract Transformers

**Definition 20 (Concrete Transformer)** Given a program statement  $s$  and concrete domain  $C$ , the **concrete transformer**  $\llbracket s \rrbracket_C : C \rightarrow C$  computes the effect of executing  $s$ :

$$\llbracket s \rrbracket_C(c) = \left\{ \sigma' \mid \exists \sigma \in c : \sigma \xrightarrow{s} \sigma' \right\} \quad (37)$$

For a concrete state set  $c$ , this is the set of all states reachable by executing  $s$  from any state in  $c$ .

#### Example

##### Variable Modification:

For  $x := x - 1$ , the concrete transformer is:

$$\llbracket x := x - 1 \rrbracket_C(c) = \{ \sigma[x \mapsto \sigma(x) - 1] \mid \sigma \in c \} \quad (38)$$

If  $c = \{(x = 64, y = 100), (x = 32, y = 50)\}$ :

$$\llbracket x := x - 1 \rrbracket_C(c) = \{(x = 63, y = 100), (x = 31, y = 50)\} \quad (39)$$

**Definition 21 (Abstract Transformer)** Given Galois connection  $(C, \alpha, \gamma, A)$  and statement  $s$ , an **abstract transformer**  $\llbracket s \rrbracket^\# : A \rightarrow A$  is **sound** if:

$$\alpha(\llbracket s \rrbracket_C(\gamma(a))) \sqsubseteq \llbracket s \rrbracket^\#(a) \quad (40)$$

for all  $a \in A$ . Equivalently, by the adjunction property:

$$\llbracket s \rrbracket_C(\gamma(a)) \subseteq \gamma(\llbracket s \rrbracket^\#(a)) \quad (41)$$

The soundness condition ensures the abstract transformer over-approximates the concrete behavior: any concrete state reachable by executing  $s$  is represented in the abstract result.

**Key Insight** Soundness is the cornerstone of abstract interpretation. We may lose precision (over-approximate), but we never claim a property holds when it doesn't (under-approximate). This makes the analysis safe for verification.

## 9.2 Best Abstract Transformers

Not all sound abstract transformers are equally precise. The **best** (most precise) transformer exists when the Galois connection has certain properties.

**Definition 22 (Best Abstract Transformer)** The **best abstract transformer** for statement  $s$  is:

$$\llbracket s \rrbracket_{\text{best}}^\#(a) = \alpha(\llbracket s \rrbracket_C(\gamma(a))) \quad (42)$$

This is the most precise sound transformer. It computes exactly the abstraction of the concrete result.

**Theorem 10 (Best Transformer Characterization)** Let  $(C, \alpha, \gamma, A)$  be a Galois connection.

1.  $\llbracket s \rrbracket_{\text{best}}^\#$  is sound (by definition).
2.  $\llbracket s \rrbracket_{\text{best}}^\#$  is the **most precise**: for any sound transformer  $\llbracket s \rrbracket^\#$ ,

$$\llbracket s \rrbracket_{\text{best}}^\#(a) \sqsubseteq \llbracket s \rrbracket^\#(a) \quad (43)$$

**Proof. Soundness:** Follows directly from the definition and Galois connection properties.

**Most precise:** Let  $\llbracket s \rrbracket^\#$  be any sound transformer. By soundness:

$$\alpha(\llbracket s \rrbracket_C(\gamma(a))) \sqsubseteq \llbracket s \rrbracket^\#(a) \quad (44)$$

Since  $\llbracket s \rrbracket_{\text{best}}^\#(a) = \alpha(\llbracket s \rrbracket_C(\gamma(a)))$  by definition, we have:

$$\llbracket s \rrbracket_{\text{best}}^\#(a) \sqsubseteq \llbracket s \rrbracket^\#(a) \quad (45)$$

■

### Example

#### Variable Decrement in Sign Domain:

For  $x := x - 1$  with sign domain, consider  $a = \text{Pos}$  (Pos element).

Concrete:  $\gamma(\text{Pos}) = \mathbb{Z}^+ = \{1, 2, \dots\}$ .

After  $x := x - 1$ :  $\llbracket x := x - 1 \rrbracket_C(\mathbb{Z}^+) = \{0, 1, \dots\}$ .

Best transformer:

- In basic sign domain  $\{\perp, \text{Neg}, \text{Zero}, \text{Pos}, \top\}$ , we get  $\alpha(\{0, 1, \dots\}) = \text{Zero} \sqcup \text{Pos}$  (since the set spans both). Without a “NonNeg” element, this join yields  $\top$ .
- In extended sign domain with “NonNeg”, the best transformer would return NonNeg (more precise).

A sound but imprecise alternative: A conservative implementation might always return  $\top$  for any subtraction, sacrificing precision for simplicity.

## 9.3 Completeness of Abstract Transformers

Best transformers are most precise, but computing them may be expensive or impossible.

**Completeness** characterizes when a practical transformer achieves the precision of the best transformer.

**Definition 23 (Completeness)** An abstract transformer  $\llbracket s \rrbracket^\#$  is **complete** (or **exact**) if:

$$\llbracket s \rrbracket^\#(a) = \alpha(\llbracket s \rrbracket_C(\gamma(a))) \quad (46)$$

for all  $a \in A$ . Equivalently, it coincides with the best transformer.

**Theorem 11 (Completeness Condition)** Let  $(C, \alpha, \gamma, A)$  be a Galois connection. Abstract transformer  $\llbracket s \rrbracket^\#$  is complete if and only if:

$$\llbracket s \rrbracket^\# \circ \alpha = \alpha \circ \llbracket s \rrbracket_C \quad (47)$$

That is, abstraction commutes with the concrete transformer.

### Example

**Variable Adjustment:**

Consider  $x := x + 4$  on intervals.

- Input:  $a = [60, 1500]$ .
- Concrete:  $\llbracket x := x + 4 \rrbracket_C(\{60, \dots, 1500\}) = \{64, \dots, 1504\}$ .
- Best:  $\alpha(\{64, \dots, 1504\}) = [64, 1504]$ .

A practical interval addition:  $[60, 1500] + 4 = [64, 1504]$  – complete!

However, for  $x := x * x$  (non-linear operation):

- Input:  $a = [-2, 2]$ .
- Concrete:  $\gamma([-2, 2]) \rightarrow \{0, 1, 4\}$  after squaring.
- Best:  $\alpha(\{0, 1, 4\}) = [0, 4]$ .
- Practical:  $[-2, 2] \times [-2, 2] = [-4, 4]$  – incomplete (loses precision at  $-4$ ).

**Precision vs. Efficiency** Complete transformers are ideal but not always feasible:

- **Best transformer:** Expensive (requires  $\gamma$ , concrete computation, then  $\alpha$ ).
- **Complete practical transformer:** Efficient direct abstract computation, same precision.
- **Incomplete transformer:** Fast but loses precision.

Analysis design balances these tradeoffs.

## 9.4 Combining Domains

Analyses often require tracking multiple properties simultaneously. While individual domains like Signs or Intervals track specific properties, combining them enables richer analysis.

The theory of domain combinations appears in Section 11, which covers:

- **Direct products:** Independent parallel analysis
- **Reduced products:** Coordination through reduction operators
- **Trace partitioning:** Path-sensitive analysis foundations
- **Formal Galois connection treatment:** Soundness proofs and precision theorems

This separation allows us to focus here on how transformers interact with single domains, while Section 11 provides the complete algebraic framework for combining them.

### Chapter Summary

This chapter completed the theoretical foundation for designing precise abstract analyses by exploring the transformation pipeline from concrete to abstract semantics.

**Abstract transformers** provide the mechanism for computing statement effects within abstract domains, replacing expensive concrete execution with tractable approximations. The notion of **best transformers** establishes an optimality criterion, defining the most precise sound approximation achievable through the abstraction function.

**Completeness** characterizes the ideal scenario where practical transformers achieve best transformer precision without computing through the costly  $\alpha \circ \llbracket s \rrbracket_C \circ \gamma$  composition. This theoretical benchmark guides the design of efficient transformers that preserve maximum precision.

These principles form the foundation for designing abstract interpretations that balance soundness with efficiency. For techniques combining multiple domains including product constructions and reduction operators, see Section 11.



# Chapter 10

## The Theory of Approximation



Advanced

Section 8 saw that Kleene iteration computes the least fixpoint of a monotone function. This guarantee comes with a catch: it only terminates if the lattice has finite height or if the function has specific properties. Many useful domains (like Intervals or Polyhedra) have infinite height, where naive iteration  $x := x \text{ join } f(x)$  might climb forever without reaching a fixpoint.

We introduce **Widening** and **Narrowing** to solve this. These operators trade precision for termination, ensuring we can analyze even infinite-state systems in finite time.

### 10.1 Widening Operators

A **widening operator**  $\nabla$  (nabla) is a binary operator  $A \times A \rightarrow A$  that accelerates convergence, acting like a “join” but guessing a stable upper bound.

**Definition 24 (Widening Operator)** An operator  $\nabla: A \times A \rightarrow A$  is a widening if:

1. **Upper Bound:** For all  $x, y \in A$ ,  $x \sqcup y \sqsubseteq x \nabla y$ .
2. **Termination:** For any ascending chain  $x_0 \sqsubseteq x_1 \sqsubseteq \dots$ , the sequence defined by  $y_0 = x_0$  and  $y_{n+1} = y_n \nabla x_{n+1}$  stabilizes in finite time.

Even if the underlying sequence  $x_n$  grows forever, the widened sequence  $y_n$  will hit a stable value (a post-fixpoint) in finite steps.

#### 10.1.1 Visualizing Widening

Consider the Interval domain: the chain  $[0, 1], [0, 2], [0, 3], \dots$  never stabilizes. A widening operator observes the growth and jumps to infinity.

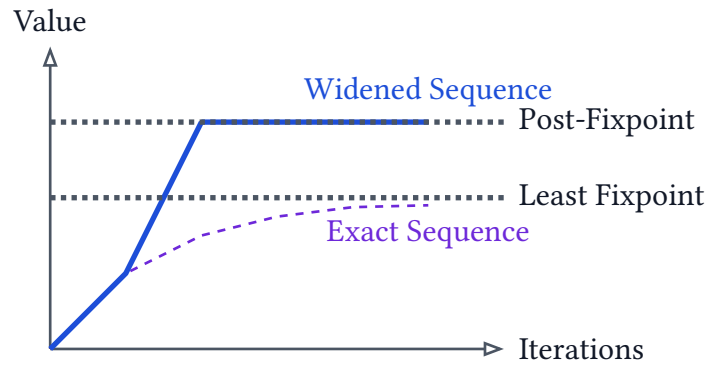


Figure 18: Widening accelerates convergence by jumping over the limit

The exact sequence approaches the Least Fixpoint (LFP) asymptotically but never reaches it. The widened sequence jumps **above** the LFP to a “Post-Fixpoint” — sound (over-approximates the true behavior) but less precise.

## 10.2 Narrowing Operators

Widening often overshoots, jumping to  $+\infty$  when the true bound is 100. Once we have a safe post-fixpoint  $x^\#$  (where  $f(x^\#) \sqsubseteq x^\#$ ), we refine it. We can’t just iterate  $f(x^\#)$  (might re-enter an infinite descending chain). We need a **narrowing operator**  $\triangle$  (Delta).

**Definition 25 (Narrowing Operator)** An operator  $\triangle: A \times A \rightarrow A$  is a narrowing if:

1. **Refinement:** For all  $y \sqsubseteq x$ ,  $y \sqsubseteq x \triangle y \sqsubseteq x$ .
2. **Termination:** For any sequence  $y_0 = x^\#$ ,  $y_{n+1} = y_n \triangle f(y_n)$ , the sequence stabilizes in finite time.

Narrowing allows us to step down from the coarse post-fixpoint towards the least fixpoint, stopping safely before we run out of time.

## 10.3 The Analysis Loop Pattern

Analyzing loops with infinite domains:

1. **Upward Iteration (Widening):** Compute  $x_0 = \perp$ ,  $x_{i+1} = x_i \nabla f(x_i)$ . Repeat until convergence  $x_n = x_{n+1}$ . Let  $x^\# = x_n$ . **Result:**  $x^\#$  is a sound over-approximation.
2. **Downward Iteration (Narrowing):** Compute  $y_0 = x^\#$ ,  $y_{i+1} = y_i \triangle f(y_i)$ . Repeat for a fixed number of steps or until convergence. **Result:**  $y_k$  is a more precise sound over-approximation.

### Real-World Example: Loop Analysis

Consider a loop incrementing a counter: `i = 80; while i < 100 { process(i); i++ }`

**Widening Phase:**

- Iter 0:  $[80, 80]$
- Iter 1:  $[80, 80] \nabla ([80, 80] \sqcup [81, 81]) = [80, 80] \nabla [80, 81]$ 
  - Standard widening jumps to  $+\infty$  if bound is unstable.
  - Result:  $[80, +\infty]$
- Iter 2: Check stability.  $f([80, +\infty]) = [80, 100]$ .
  - $[80, +\infty]$  is a post-fixpoint (since  $[80, 100] \subseteq [80, +\infty]$ ).

**Narrowing Phase:**

- Start:  $[80, +\infty]$
- Apply  $f$ :  $f([80, +\infty]) = [80, 100]$  (effect of loop guard `i < 100`).
- Narrow:  $[80, +\infty] \triangle [80, 100] = [80, 100]$ .
- Result:  $[80, 100]$ . Perfect precision!

## 10.4 Designing Widening Operators

Good widening operators exploit domain-specific structure to balance termination guarantees with precision preservation.

For **interval domains**, the standard widening strategy detects bound changes: if either endpoint shifts between iterations, that bound extrapolates to the corresponding infinity ( $+\infty$  for upper bounds,  $-\infty$  for lower bounds).

**Threshold-based widening** refines this approach by jumping to values in a predetermined set  $T = \{0, 1, 2, 4, 8, \dots\}$  rather than immediately reaching infinity. This technique captures common program constants and loop bounds, significantly improving precision for loops with small iteration counts.

**Delayed widening** defers extrapolation by performing  $N$  normal join operations before activating the widening operator. This strategy handles short loops precisely, avoiding premature over-approximation when natural convergence would succeed within a few iterations.

**Widening and BDDs** BDD domains usually have finite height (for a fixed number of variables), so strictly speaking, they don't **need** widening to terminate. However, the lattice height is  $2^N$ , which is practically infinite. We apply “structural widening” to BDDs (e.g., limiting node count or merging paths) to ensure **efficient** termination, as discussed in Section 12.

# Chapter 11

## Algebraic Domain Combinations



Previous chapters explored individual abstract domains like Intervals and Signs. Real-world analysis often requires tracking multiple properties simultaneously or capturing relationships between them. We formalize the algebra of **combining** abstract domains.

This chapter develops the theory of domain combinations, progressing from simple parallel execution to sophisticated coordination mechanisms. We begin with **direct products**, which run multiple analyses independently by pairing their results. Next, we introduce **reduced products**, which enable domains to exchange information and refine each other's precision. We then examine **trace partitioning**, the theoretical foundation that enables path-sensitive analysis by distinguishing abstract states based on control flow history. Finally, we explore **relational domains**, which transcend per-variable analysis by tracking correlations between multiple variables simultaneously.

### 11.1 The Direct Product

Combine two domains  $A$  and  $B$  via the **direct product**: run two independent analyses and pair their results.

**Definition 26 (Direct Product Domain)** Given domains  $(C, \alpha_1, \gamma_1, A_1)$  and  $(C, \alpha_2, \gamma_2, A_2)$  over the same concrete domain, their **direct product**  $A_1 \times A_2$  is defined as:

**Elements:** Pairs  $(a_1, a_2)$  where  $a_1 \in A_1$  and  $a_2 \in A_2$ .

**Ordering:**  $(a_1, b_1) \sqsubseteq (a_2, b_2) \iff a_1 \sqsubseteq_{A_1} a_2 \wedge b_1 \sqsubseteq_{A_2} b_2$ .

**Lattice operations (component-wise):**

- Join:  $(a_1, b_1) \sqcup (a_2, b_2) = (a_1 \sqcup_{A_1} a_2, b_1 \sqcup_{A_2} b_2)$
- Meet:  $(a_1, b_1) \sqcap (a_2, b_2) = (a_1 \sqcap_{A_1} a_2, b_1 \sqcap_{A_2} b_2)$

**Galois connection:**

- Abstraction:  $\alpha(c) = (\alpha_1(c), \alpha_2(c))$
- Concretization:  $\gamma((a_1, a_2)) = \gamma_1(a_1) \cap \gamma_2(a_2)$

The direct product answers questions neither domain could answer alone, but domains cannot **help** each other.

The direct product is sound but may contain **unrealizable** elements: pairs  $(a_1, a_2)$  where  $\gamma_1(a_1) \cap \gamma_2(a_2) = \emptyset$ . These represent inconsistent abstract states that correspond to no concrete execution.

### Direct Product: Precision Loss and Unrealizability

#### Precision loss from independence:

Consider the product of **Signs** and **Parity** domains. Let  $x$  be **Pos** in Signs and **Even** in Parity. State: (Pos, Even).

Now, assume we execute  $x = x / 2$ .

- Signs:  $\text{Pos} / 2 \rightarrow \text{Pos}$ .
- Parity:  $\text{Even} / 2 \rightarrow \top$  (could be **Even** or **Odd**).
- Result: (Pos,  $\top$ ).

We lost the information that  $x$  was **Even**! If we knew  $x = 6$ , then  $x/2 = 3$  (**Odd**). The domains operated independently and failed to refine the result.

#### Unrealizable states:

Consider **Type**  $\times$  **Value** product:

- **Type**:  $\{\perp, \text{Int}, \text{Float}, \text{Bool}, \top\}$
- **Value**: Interval domain  $[0, 100]$

Product element  $(\text{Int}, [5, 5])$  represents integer 5 — realizable.

Product element  $(\text{Bool}, [5, 5])$  – **unrealizable!** Booleans are 0 or 1. The value 5 contradicts the type `Bool`. The concrete meaning is  $\gamma(\text{Bool}) \cap \gamma(\text{Value}[5]) = \emptyset$ .

## 11.2 The Reduced Product

The **reduced product** improves upon the direct product by allowing information exchange (reduction) between domains.

**Definition 27 (Reduction Operator)** A **reduction operator**  $\rho : A_1 \times A_2 \rightarrow A_1 \times A_2$  transforms a pair  $(a_1, a_2)$  into a more precise pair  $(a'_1, a'_2)$  satisfying:

1. **Soundness:**  $\gamma(\rho(a_1, a_2)) = \gamma(a_1, a_2)$  (no concrete states lost)
2. **Improvement:**  $\rho(a_1, a_2) \sqsubseteq (a_1, a_2)$  (result is more precise)
3. **Idempotence:**  $\rho(\rho(a_1, a_2)) = \rho(a_1, a_2)$  (repeated application stabilizes)
4. **Monotonicity:**  $(a_1, a_2) \sqsubseteq (b_1, b_2) \Rightarrow \rho(a_1, a_2) \sqsubseteq \rho(b_1, b_2)$  (preserves ordering)

The **reduced product** applies  $\rho$  after each operation to maintain a canonical form.

Reduction propagates constraints discovered by one domain to the other.

### Reduction Examples

#### Intervals $\times$ Congruence:

State:  $x \in [10, 12]$  AND  $x \equiv 0 \pmod{5}$ .

- Intervals alone: 10, 11, 12.
- Congruence alone: ..., 5, 10, 15, ...
- Intersection: {10}.

**Reduction:** The congruence domain tells the interval domain: “The only valid value in  $[10, 12]$  is 10.” Refined Interval:  $[10, 10]$ .

The interval domain tells the congruence domain: “The value is exactly 10.” Refined Congruence:  $x \equiv 0 \pmod{10}$  (if supported).

#### Type $\times$ Value:

```
fn reduce(ty: Type, val: Interval) → (Type, Interval) {
  match ty {
    Type::Bool => {
      // Bool implies value is 0 or 1
      let bool_range = Interval::new(0, 1);
      let refined = val.intersect(bool_range);
      if refined.is_bottom() {
        (Type::Bottom, Interval::bottom())
      } else {

```

```

        (ty, refined)
      }
    },
    Type::Int | Type::Float => (ty, val),
    _ => (ty, val),
  }
}

```

After reduction,  $(\text{Bool}, [5, 5])$  becomes  $(\perp, \perp)$ , making the inconsistency explicit.

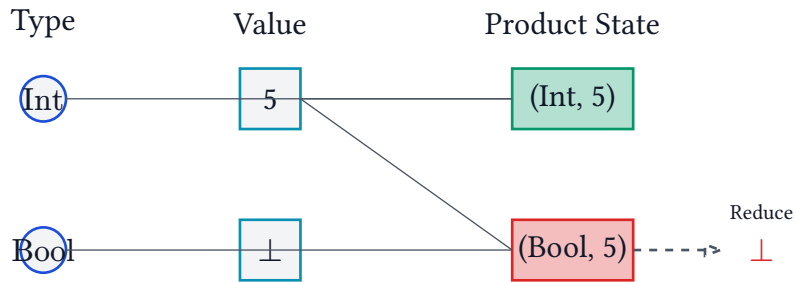


Figure 19: Reduced product eliminates unrealizable states

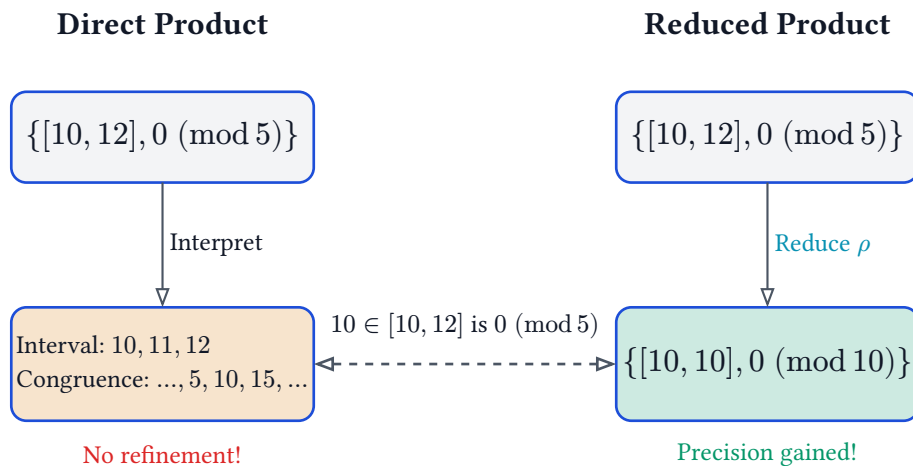


Figure 20: Direct vs. Reduced Product: Intervals  $\times$  Congruence

In practice, computing the **optimal** reduction (the Granger-Cousot reduction) can be expensive. Most analyzers use **local iterations** or specific reduction heuristics (e.g., “**Signs** refines **Intervals**”).

**Implementing the Reduction Loop** In a multi-domain product, reduction can be iterative. Domain A refines B, which refines C, which might refine A again!

```

fn reduce_loop(mut state: ProductState) → ProductState {
  loop {
    let old_state = state.clone();

    // 1. Apply all reduction rules

```

```

    state = apply_rules(state);

    // 2. Check for fixpoint
    if state == old_state { break; }

    // 3. Check for bottom (contradiction)
    if state.is_bottom() { return ProductState::bottom(); }
  }
  state
}

```

For finite height domains, this loop always terminates.

**Theorem 12 (Reduced Product Precision)** Let  $(C, \alpha_i, \gamma_i, A_i)$  for  $i = 1, 2$  be Galois connections, and  $\rho$  be a reduction operator.

1. The reduced product forms a Galois connection with:

$$\alpha(c) = \rho(\alpha_1(c), \alpha_2(c)) \quad (48)$$

$$\gamma((a_1, a_2)) = \gamma_1(a_1) \cap \gamma_2(a_2) \quad (49)$$

2. Reduced product is **at least as precise** as the direct product:

$$\rho(a_1, a_2) \sqsubseteq (a_1, a_2) \quad (50)$$

**Proof. Galois connection:** Soundness of  $\rho$  ensures  $\gamma(\rho(a_1, a_2)) = \gamma(a_1, a_2)$ , preserving the concretization. The abstraction  $\alpha$  followed by reduction maintains the adjunction property.

**Precision:** Reduction operator is **decreasing**:  $\rho(a_1, a_2) \sqsubseteq (a_1, a_2)$  by soundness (same concretization) and idempotence (stabilizes at a smaller element). Since  $\sqsubseteq$  is the precision ordering (smaller = more precise), the reduced product is more precise. ■

### 11.2.1 Designing Reduction Operators

Effective reduction operators exploit **domain-specific relationships**.

**Definition 28 (Reduction Strategies)** Common reduction patterns:

1. **Contradiction elimination:** Detect  $\gamma(a_1) \cap \gamma(a_2) = \emptyset$ , set both to  $\perp$ .
2. **Constraint propagation:** Tighten one domain based on constraints from another.
3. **Interval refinement:** Use sign/parity to narrow interval bounds.
4. **Relational tightening:** Use relationships between variables.



### Example

#### Mode × Value reduction:

```
fn reduce(mode: Mode, val: Interval) → (Mode, Interval) {
  match mode {
    Mode::Fast ⇒ {
      // Fast mode: value must be > 100
      let refined = val.intersect(Interval::new(101, 1000));
      (mode, refined)
    }
    Mode::Safe ⇒ {
      // Safe mode: value must be ≤ 100
      let refined = val.intersect(Interval::new(0, 100));
      (mode, refined)
    }
    _ ⇒ (mode, val),
  }
}
```

Input: (Fast, [0, 200]).

After reduction: (Fast, [101, 200]) — Safe values eliminated!

**Reduction Cost** Reduction adds computational overhead. Apply reduction:

- After joins (most beneficial).
- Periodically during fixpoint iteration.
- Only when precision gain justifies cost.

Measure impact experimentally!

## 11.2.2 Multi-Domain Products

**Definition 29 (Multi-Domain Reduced Product)** Given domains  $A_1, \dots, A_n$  with abstractions  $\alpha_i$  and concretizations  $\gamma_i$ , the reduced product is:

$$A = \{(a_1, \dots, a_n) \mid a_i \in A_i, \gamma_1(a_1) \cap \dots \cap \gamma_n(a_n) \neq \emptyset\} \quad (51)$$

with reduction operator:

$$\rho(a_1, \dots, a_n) = \text{fix}(\lambda(b_1, \dots, b_n).(\rho_1(b_1, \dots, b_n), \dots, \rho_n(b_1, \dots, b_n))) \quad (52)$$

where each  $\rho_i$  refines domain  $A_i$  based on constraints from other domains.

### Example

#### Type × Mode × Value:

For variable  $v$ :

- Type: `Int`
- Mode: `Const`
- Value:  $[0, 100]$

Reduction:

1. Type refines Value: `Int` implies integer values.
2. Mode refines Value: `Const` implies singleton value (if we knew it).
3. Result:  $(\text{Int}, \text{Const}, [0, 100])$ .

If Mode was `Pos`, we could refine Value to  $[1, 100]$ .

## 11.3 Trace Partitioning

**Trace partitioning** is a powerful technique to gain precision by distinguishing execution paths. Instead of merging control flows immediately, we maintain separate abstract states for different history traces.

**Definition 30 (Trace Partitioning Domain)** Given a set of trace tokens  $T$  (representing control paths) and a domain  $A$ , the trace partitioning domain is the function space:

$$A_T = T \rightarrow A \quad (53)$$

An element  $f \in A_T$  maps each trace  $t \in T$  to an abstract state  $f(t)$ . The concrete meaning is the union of states over all traces:

$$\gamma(f) = \bigcup_{t \in T} \gamma_A(f(t)) \quad (54)$$

This is the theoretical foundation for **path-sensitive analysis**. If  $T$  represents the “current basic block” or “last branch taken”, we get standard path sensitivity. If  $T$  represents “call stack”, we get context sensitivity (interprocedural analysis).

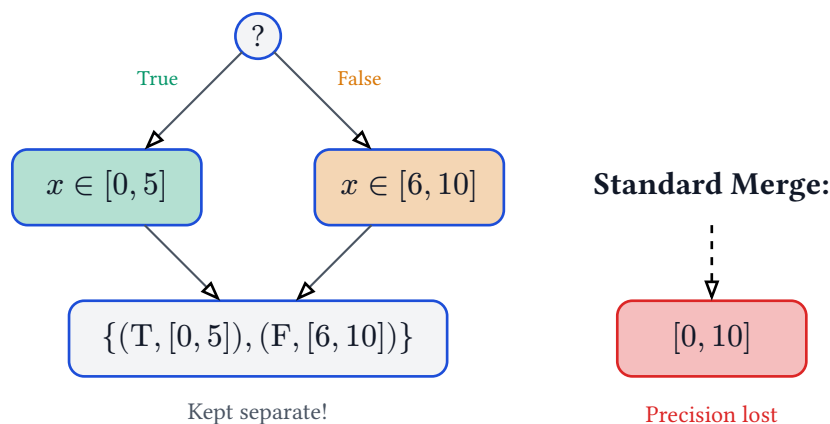


Figure 21: Trace partitioning splits abstract states by path

**Partitioning vs. Disjunctive Completion** Trace partitioning is a practical approximation of **disjunctive completion** (the power set domain  $P(A)$ ).

- $P(A)$  allows **arbitrary** disjunctions:  $(x = 1) \vee (x = 5)$ .
- Trace partitioning allows disjunctions **aligned with control flow**:  $(\text{path}_1 \wedge x = 1) \vee (\text{path}_2 \wedge x = 5)$ .

This structure makes trace partitioning much more efficient than full disjunctive completion.

## 11.4 Abstract Transformers for Products

Transformers on product domains must maintain reduction.

### Algorithm 5: Product Domain Transfer Function

**Input:** Statement  $s$ , product state  $(a_1, \dots, a_n)$ .

**Output:** Transformed product state.

```

1  for  $i = 1$  to  $n$  do
2     $\sqsubseteq a'_i \leftarrow \llbracket s \rrbracket_i^\#(a_i)$     // Apply transformer in each domain.
3     $(a'_1, \dots, a'_n) \leftarrow \rho(a'_1, \dots, a'_n)$     // Reduce result.
4  return  $(a'_1, \dots, a'_n)$ 

```

**Theorem 13 (Soundness of Product Transformers)** If each component transformer  $\llbracket s \rrbracket_i^\#$  is sound and  $\rho$  is a sound reduction, then the product transformer is sound.

**Proof.** Let  $c = \gamma(a_1, \dots, a_n)$  be the concrete set represented by the product element.

After executing  $s$  concretely:  $c' = \llbracket s \rrbracket_C(c)$ .

By soundness of each component:

$$c' \subseteq \gamma_i(\llbracket s \rrbracket_i^\#(a_i)) \quad \text{for all } i \quad (55)$$

Thus:

$$c' \subseteq \bigcap_{i=1}^n \gamma_i(\llbracket s \rrbracket_i^\#(a_i)) = \gamma(a'_1, \dots, a'_n) \quad (56)$$

Reduction preserves concretization:  $\gamma(\rho(a'_1, \dots, a'_n)) = \gamma(a'_1, \dots, a'_n)$ .

Therefore:  $c' \subseteq \gamma(\rho(a'_1, \dots, a'_n))$ , establishing soundness. ■

## 11.5 Relational Domains

So far, we have discussed **non-relational** domains (like Intervals), which track properties of variables independently ( $x \in [a, b]$ ). **Relational domains** track relationships **between** variables.

### 11.5.1 Common Relational Abstractions

Relational domains differ in the types of constraints they can express, creating fundamental tradeoffs between precision and performance.

**Octagon domains** track constraints of the form  $\pm x \pm y \sqsubseteq c$ , capturing simple relationships like “ $x$  is at most 5 greater than  $y$ ”. They achieve efficient  $O(n^3)$  complexity, making them practical for array bounds checking where relationships like  $i < n$  are common.

**Polyhedral domains** use general linear inequalities  $\sum a_i x_i \sqsubseteq c$ , expressing arbitrary linear constraints between variables. They offer maximum precision for linear relationships but suffer exponential complexity in the worst case.

**Equality domains** specialize in constraints of the form  $x = y + c$ , precisely tracking affine relationships. They are particularly effective for alias analysis and variable substitution.

#### Real-World Example: Consistency Check

Consider a security check for user roles:

```
// User is admin?
if user_id == admin_id {
    // Must have admin role
    if role != "Admin" {
        error();
    }
}
```

- **Intervals**: Tracks range of `user_id` and `role` independently. Cannot capture the correlation “if user is admin, role must be Admin”.
- **Relational**: Tracks `user_id == admin_id  $\Rightarrow$  role == "Admin"`. Can prove that the `error()` is unreachable for consistent states.

## 11.6 Precision vs. Cost Tradeoffs

**Definition 31 (Analysis Metrics)** For comparing domain choices:

1. **Precision**: How many false alarms? Can we prove the property?
2. **Cost**: Time and memory for analysis.
3. **Scalability**: Performance on large programs.
4. **Expressiveness**: What properties can we verify?

**Example****Domain comparison for  $x := x * x$ :**

Domain	Result for $x \in [-2, 2]$	Cost
Sign	$\top$	Low
Interval	$[-4, 4]$	Medium
Sign $\times$ Interval (unreduced)	$(\top, [-4, 4])$	Medium
Sign $\times$ Interval (reduced)	$(\text{NonNeg}, [0, 4])$	High
Octagon	$x^2 \in [0, 4]$	Very High

Reduced product eliminates **Neg** results, but at computational cost.

**Choosing Domains** Domain selection heuristics:

- **Start simple:** Sign or intervals for initial analysis.
- **Add domains incrementally:** Identify precision bottlenecks.
- **Use reduction selectively:** Only where needed.
- **Profile performance:** Measure time/space costs.
- **Consider problem structure:** Some domains excel on certain patterns.

## 11.7 Widening in Product Domains

When combining domains, the widening operator must also be combined. For a product  $A \times B$ , the standard widening is component-wise:

$$(a_1, b_1) \nabla (a_2, b_2) = \left( a_1 \nabla_A a_2, b_1 \nabla_B b_2 \right) \quad (57)$$

However, this can be too aggressive. **Delayed widening** or **widening with thresholds** is often necessary to prevent precision loss in one domain from destabilizing the other.

### Chapter Summary

This chapter developed a hierarchy of domain combination techniques, each adding sophistication to multi-property analysis.

The **direct product** provides the foundation by running multiple analyses independently, combining their results through simple pairing. While answering questions neither domain could handle alone, it cannot exploit synergies between domains.

The **reduced product** overcomes this limitation by introducing reduction operators that enable bidirectional information exchange. Domains can now refine each other's abstract states, recovering precision lost to independent analysis.

The **trace partitioning** construction provides the theoretical basis for path sensitivity. By distinguishing abstract states based on execution history captured through control flow predicates, it enables precise reasoning about conditional program behavior.

Finally, **relational domains** transcend per-variable analysis by tracking correlations between multiple variables. They enable expressing constraints like  $x < y$  or  $x = 2y + 1$ , which are essential for reasoning about array bounds, pointer arithmetic, and data structure consistency.

In the next chapter, we will implement a powerful instance of these concepts: a **Reduced Product of BDDs (Trace Partitioning) and Abstract Domains**. This “Killer Feature” uses BDDs to efficiently manage the trace partition  $T$ , enabling scalable path-sensitive analysis.

# Chapter 12

## BDD Path Sensitivity

### Implementation Focus

Section 11 established the theory of **Trace Partitioning** and **Reduced Products**. We now implement the “Killer Feature”: using Binary Decision Diagrams (BDDs) as the trace partitioning domain.

This architecture solves the “Diamond Problem” (loss of precision at join points) by maintaining separate abstract states for different execution paths, efficiently compressed by the BDD.

## 12.1 The BDD Product Domain

We define a generic `BddProductDomain<D>` that combines a BDD (control) with an arbitrary abstract domain `D` (data).

**Definition 32 (BDD Product State)** A state in the BDD product domain is a pair  $(f, d)$  where:

- $f$ : A BDD representing the set of active control paths.
- $d$ : An element of domain  $D$  representing data properties on those paths.

In Rust, we implement this using the `bdd-rs` library.

```
pub struct BddProductDomain<D: AbstractDomain> {
    bdd: Rc<Bdd>,           // Shared BDD manager
    control: Ref,            // The path condition 'f'
    data: D,                 // The data state 'd'
}
```

**Manager-Centric Design** Note the `Rc<Bdd>`. In `bdd-rs`, the `Bdd` struct is the manager that owns all nodes. The `control` field is just a `Ref` (a lightweight integer handle). All operations must go through the manager: `self.bdd.and(self.control, other.control)`.

## 12.2 Implementing Lattice Operations

The lattice operations for the product domain combine the control and data components.

### 12.2.1 Join (Union)

When merging two control-flow paths, we join the path conditions (logical OR) and the data states (domain join).

```
fn join(&self, other: &Self) → Self {
    // Control: Union of paths
    let control = self.bdd.apply_or(self.control, other.control);

    // Data: Join of data facts
    let data = self.data.join(&other.data);

    BddProductDomain {
        bdd: self.bdd.clone(),
        control,
        data
    }
}
```

### 12.2.2 Meet (Intersection)

Used when paths converge or when applying constraints.

```
fn meet(&self, other: &Self) → Self {
    let control = self.bdd.apply_and(self.control, other.control);
    let data = self.data.meet(&other.data);
    BddProductDomain { bdd: self.bdd.clone(), control, data }
}
```

## 12.3 The Transfer Function: Assume & Filter

The `assume` function (filtering) updates **both** the BDD control state and the data state when the program encounters a condition `if x > 0`.

1. **Control Update:** We AND the current path condition with the BDD variable representing `x > 0`.
2. **Data Update:** We refine the data domain using `assume(x > 0)`.

```
fn assume(&self, condition: &Expr) → Self {
    // 1. Map condition to BDD variable
    let cond_bdd = self.map_condition_to_bdd(condition);
    let new_control = self.bdd.apply_and(self.control, cond_bdd);

    // 2. Refine data domain
    let new_data = self.data.assume(condition);

    BddProductDomain {
        bdd: self.bdd.clone(),
        control: new_control,
        data: new_data,
    }
}
```

This ensures that on the “true” branch, we know `x > 0` in both the BDD (for path tracking) and the data domain (for value analysis).



## 12.4 Reduction: The “Killer” Interaction

The **Reduced Product** (Section 11) allows the BDD to refine the data domain. If the BDD knows that a certain path is impossible (control is `false`), the data state should be `bottom`. More advanced reduction can extract facts from the BDD to refine the data.

### Reduction Example

If `control` implies  $x > 0$  (because we are on the true branch of an earlier check), we can tell the Interval domain to clip negative values of  $x$ .

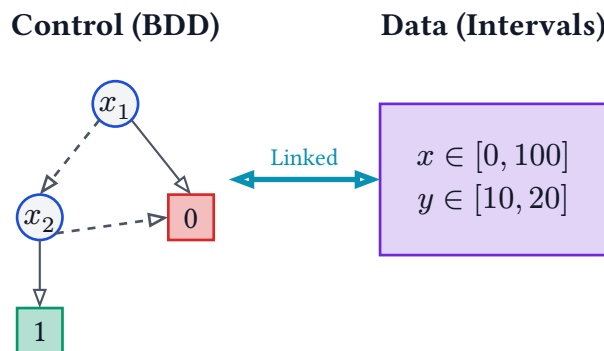


Figure 22: BDD Product State: Control + Data

## 12.5 Variable Ordering and Performance

BDD performance is sensitive to variable ordering. In `bdd-rs`, variables are 1-indexed. For path sensitivity, a good heuristic is to order variables by their appearance in the control flow graph (CFG).

**Dynamic Variable Allocation** If you allocate BDD variables dynamically as you encounter branches, ensure you reuse the **same** variable for the **same** condition. Mapping  $x > 0$  to  $v_1$  at line 10 and  $v_2$  at line 20 destroys the correlation. Use a canonical mapping: `Map<Condition, BddVar>`.

## 12.6 Case Study: Array Access Safety

Let’s see this in action on a buffer processing routine. This is a classic source of vulnerabilities: checking a size field but failing to respect it during access.

```
fn process_data(arr: &[u8], size: usize) {
    // 1. Check size
    if size < 4 { return; }

    // 2. Access metadata (safe because size ≥ 4)
    let meta = arr[0..4];
}
```

```

// 3. Check content type
if meta[0] == 0x1 {
  // 4. Access content (requires size ≥ 8)
  if size ≥ 8 {
    let content = arr[4..8];
  }
}
}

```

### 12.6.1 Detailed Analysis Trace

Let's walk through this analysis step by step, tracking how the BDD path condition and data domain evolve together.

**At entry:** The variable `size` has interval  $[0, \infty]$  (unknown but non-negative). The path condition is True (all paths feasible initially).

**After first branch (`size < 4`):** We take the false branch (fallthrough), meaning the condition `size < 4` is false. The BDD adds constraint  $\neg v_1$  (where  $v_1$  represents `size < 4`). The data domain refines: `size`  $\in [4, \infty]$ .

**At first array access (`arr[0..4]`):** Safety requires `size ≥ 4` to access indices 0 through 3. The data domain confirms  $[4, \infty] \subseteq [4, \infty]$ , so this access is safe.

**After third branch (`meta[0] == 0x1`):** We enter the true branch, so the BDD adds  $v_2$  (representing `meta[0] == 0x1`). The path condition is now  $\neg v_1 \wedge v_2$ .

**After fourth branch (`size ≥ 8`):** We enter the true branch again, so the BDD adds  $v_3$  (representing `size ≥ 8`). The path condition becomes  $\neg v_1 \wedge v_2 \wedge v_3$ . The data domain refines: `size`  $\in [8, \infty]$ .

**At second array access (`arr[4..8]`):** Safety requires `size ≥ 8` to access indices 4 through 7. The data domain confirms  $[8, \infty] \subseteq [8, \infty]$ , so this access is safe.

Without path sensitivity (BDD), merging the paths after Branch 4 would lose the correlation between “we are inside the `size ≥ 8` block” and the variable `size`. The BDD keeps these states distinct if we use partitioning, or allows us to recover the condition if we use the product domain.

## 12.7 Performance Considerations

BDD-based path sensitivity trades precision for computational cost. Managing this tradeoff determines whether an analyzer scales to real programs.

### 12.7.1 BDD Size Growth

Well-structured control flow produces compact BDDs. A function with  $n$  sequential conditionals generates  $O(n)$  nodes when variable ordering follows control flow. However, complex Boolean combinations can trigger exponential blowup to  $O(2^n)$  nodes.

**Key factors affecting size:**

- **Variable ordering**: Allocate BDD variables following reverse postorder CFG traversal. This groups related conditions together, maximizing structural sharing.
- **Loop complexity**: Deeply nested loops with many exit conditions stress the representation.
- **Boolean structure**: Arbitrary functions over distant program points prevent sharing.

#### Mitigation approaches:

- Trigger garbage collection at loop headers or when node count exceeds thresholds (e.g., 10,000 nodes).
- Impose hard node limits per function (e.g., 100,000 nodes). When exceeded, widen aggressively or fall back to path-insensitive mode.

### 12.7.2 Loop Widening

Loops create infinite path families that require finite representation. After  $k$  iterations (typically  $k = 2$ ), replace the path condition with `True`:

```
fn widen_bdd(&self, iteration: usize) → Ref {
    if iteration ≥ 2 {
        self.bdd.constant(true) // Abandon path tracking
    } else {
        self.control
    }
}
```

This abandons path sensitivity within loops, reverting to path-insensitive analysis. More sophisticated strategies preserve conditions guarding safety checks while dropping routine control flow.

### 12.7.3 Performance Profiling

Identify whether BDD operations or data domain operations dominate:

- **Cheap data domains** (intervals, signs): BDD operations dominate. Focus on variable ordering and garbage collection.
- **Expensive data domains** (polyhedra, automata): Domain operations dominate. BDD overhead is negligible; consider domain simplifications instead.
- **Typical case** (BDDs + intervals): Expect some overhead versus path-insensitive analysis.

### 12.7.4 Practical Guidelines

1. Use widening threshold  $k = 2$ . Most programs converge quickly; higher thresholds rarely improve precision.
2. Allocate BDD variables in reverse postorder. This single heuristic handles most ordering challenges.
3. Monitor node count during iteration. Spikes indicate problem regions requiring specialized handling.
4. For complex functions, fall back to path-insensitive analysis automatically. Hybrid approaches maintain soundness while bounding worst-case cost.

## Chapter Summary

This chapter made path-sensitive abstract interpretation concrete through the BDD-based product domain implementation.

The `BddProductDomain` architecture combines a BDD manager (tracking feasible paths) with an arbitrary data domain (tracking variable values). This separation enables **orthogonal composition** — switching between different data abstractions without modifying the path-tracking mechanism.

**Lattice operations** (`join`, `meet`) operate component-wise: combine path conditions through Boolean operations, merge data states through domain operations. The critical **assume operation** updates both layers: conjoining conditions to the path BDD and refining data facts based on the assumption.

This architecture enables **automatic infeasible path detection**. When a path BDD becomes False through contradiction, that execution trace is proven unreachable. The executor can prune entire branches without explicit satisfiability checking.

The implementation demonstrates path sensitivity's **practical value**: verifying properties dependent on control flow sequences, like array bounds under conditional guards or authorization checks under role conditions.

# Chapter 13

## String and Automata Domains



Advanced



Implementation Focus

String-heavy programs require reasoning about concatenation, length, prefixes/suffixes, and membership in regular sets (input validation, sanitization). We develop abstract domains for strings, from lightweight numeric properties to regular-language abstractions via finite automata.

### 13.1 Motivation and Threat Model

Typical security and correctness questions:

- Does input validation enforce membership in a safe regular language (*e.g.*, email, path, identifier)?
- Can a dangerous character reach a sink after normalization (*e.g.*, doubled slashes collapsed)?
- Do string lengths stay within bounds?

We aim for conservative answers: no false negatives.

### 13.2 Scalar String Properties: Length and Charset

**Definition 33 (Length Domain)** Elements are  $[m, n]$  with  $0 \leq m \leq n \leq +\infty$ . Join/meet are interval operations; widening moves unstable bounds to 0 or  $+\infty$ .

Transformers:

- $\text{len}(x \cdot y) \in [m_x + m_y, n_x + n_y]$
- $\text{len}(x.\text{trim})$  narrows upper bound by at most a constant.

**Definition 34 (Charset Domain)** Track a set of allowed characters  $C \subseteq \Sigma$  (alphabet). Join is union; meet is intersection; widening limits the number of tracked classes.

### Reduced Product: Length $\times$ Charset

Combining length and charset refines feasibility:

- If charset excludes '/', then `normalize(path)` cannot introduce '/' via contraction.
- If length lower bound is 1 and charset excludes NUL, `is_empty` is provably false.

## 13.3 Regular-Language Abstraction via Automata

Represent sets of strings with a deterministic finite automaton (DFA)  $A = (Q, q_0, F, \delta)$ .

**Definition 35 (Automata Domain)** The abstract element is a DFA; concretization is its language  $L(A)$ .

- Join: language union via automaton union (product construction + accepting set union)
- Meet: language intersection via product automaton + accepting set intersection
- Widening: language over-approximation by merging states (quotienting) using a refinement-stable partition

**Theorem 14 (Soundness of Quotient Widening)** Let  $\sim$  be a right-invariant equivalence on  $Q$  (Myhill–Nerode style refinement-stable). Quotient  $A/\sim$  recognizes a language  $L'$  with  $L(A) \subseteq L'$ . Thus replacing  $A$  by  $A/\sim$  during iteration preserves soundness and ensures convergence under bounded refinements.

## 13.4 Transformers for Common Operations

- Concatenation:  $L(A_x \cdot A_y) = \{xy \mid x \in L(A_x), y \in L(A_y)\}$  via epsilon-bridge construction with determinization.
- Prefix/Suffix:  $\text{prefix}_k(L)$  and  $\text{suffix}_k(L)$  realized by state annotation up to bound  $k$ ; widen by increasing  $k$  lazily.
- Replace: Over-approximate  $\text{replace}(s, \text{re}, r)$  by composing with a transducer; when unavailable, approximate by charset and length effects plus inclusion checks.
- Substring: Restrict by positions using length bounds; refine with automata if indices are narrow.

**Determinization Blow-up** Determinization may be exponential. Use bounds (e.g., depth, number of states) and switch to scalar domains when limits exceed thresholds.

## 13.5 Reduced Products and Cooperation

Combine automata with scalar domains to improve precision:

- Length refines automata by trimming unreachable states (length-infeasible).
- Charset prunes transitions that produce forbidden characters.
- BDD control refines string facts path-sensitively (e.g., `if is_alpha(x) { ... }`).

**Definition 36 (Granger-Style Reduced Product)** Let  $(A_1, A_2)$  be two abstract elements. A **reduction** operator  $\rho(A_1, A_2)$  computes a pair  $(A_{1'}, A_{2'})$  such that  $\gamma(A_{1'}, A_{2'}) = \gamma(A_1, A_2)$  and  $A_{i'} \leq A_i$ . Implement by mutual refinement until a local fixpoint or budgets are exhausted.

## 13.6 Validation and Normalization Pipelines

We model pipelines `validate; normalize; use`:

- Validation: assert  $x \in L_{\text{safe}}$  (meet with DFA for  $L_{\text{safe}}$ )
- Normalization: apply length/charset-preserving rewrites; ensure closure under rewrites remains in  $L_{\text{safe}}$
- Use: sinks (e.g., regex match, path join) proved safe by inclusion:  $L(\text{current}) \subseteq L_{\text{safe}}$

### Real-World Example: Malicious Input Detection

Consider an Input Validation rule: `if input.contains("attack") { reject() }`

- **String Domain:** Tracks the set of possible strings in `input`.
- **Analysis:**
  - If `input` comes from a trusted source (e.g., internal config), the domain might prove it never contains “attack”.
  - If `input` is user input, the domain tracks that after the `if`, the input **must** contain “attack” (on the true branch) or **must not** (on the false branch).
  - This allows proving that subsequent code is only reachable for safe inputs.

## 13.7 Widening/Narrowing Patterns

- Automata quotienting by state partition; refine back (narrowing) with counterexample-guided splitting.
- Length widening by bounds to  $[0, +\infty]$ ; narrow with concrete constraints (e.g., bounded loops).
- Charset widening by class coarsening (e.g., collapse to classes: alpha, digit, other).

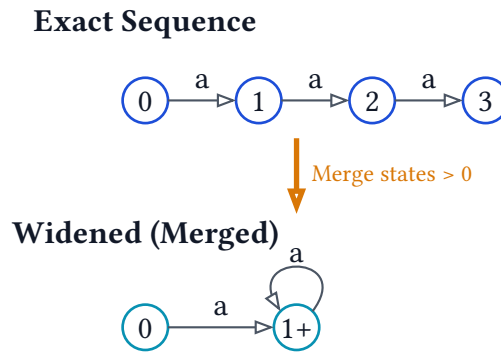


Figure 23: Automata widening by merging states

## Chapter Summary

This chapter extended abstract interpretation into the string domain, enabling reasoning about programs that manipulate text — critical for parsing, validation, and security analysis.

**Finite automata** provide the abstract representation, compactly encoding potentially infinite sets of strings. Each automaton represents a regular language, with states corresponding to partial parsing positions and transitions encoding character consumption. String **operations** (concatenation, union, intersection, replacement) have efficient automata-theoretic implementations. These enable tracking how strings flow through program operations while maintaining polynomial-time complexity for key operations.

As with other infinite-height domains, **widening operators** prevent unbounded automaton growth. Language-based widening collapses states to bound automaton size while preserving essential string properties like prefix sets or character classes.

**Integration with BDDs and numeric domains** through reduced products enables sophisticated analyses. The BDD layer can track control flow dependencies (“if valid, then process”), while numeric domains constrain string lengths, creating a powerful multi-property verification framework.



# Chapter 14

## Points-to and Dynamic Type Domains



Advanced



Implementation Focus

Heap-manipulating programs require alias reasoning and heap modeling; dynamic languages require tracking runtime types. We present a lightweight may-points-to abstraction with allocation-site sensitivity and a flow-sensitive dynamic type domain.

### 14.1 Program Model and Sensitivities

We assume a first-order, imperative core language with variables `Var`, allocation sites `Site` (identified by program points), and fields `Fld`. Analyses may be:

- Flow-insensitive vs. flow-sensitive (this chapter prefers flow-sensitive states at program points).
- Context-insensitive vs. context-sensitive (see Section 17 for  $k$ -limited call-strings).
- Field-insensitive vs. field-sensitive (we recommend field-sensitive stores for precision).

### 14.2 May/Must Aliasing and Allocation Sites

**Definition 37 (Points-to Map and Concretization)** A may-points-to map is a function  $PT : \text{Var} \rightarrow \mathcal{P}(\text{Site})$ . Its concretization  $\gamma_{PT}(PT)$  is the set of concrete heaps where, for each pointer variable  $p$ , any concrete location referenced by  $p$  belongs to  $PT(p)$ .

Ordering:  $PT_1 \leq PT_2$  iff  $\forall p. PT_1(p) \subseteq PT_2(p)$ . Join/meet: pointwise union/intersection. Widening: cap  $|PT(p)|$  and collapse excess sites to a summary site.

**Definition 38 (Abstract Store and Strong/Weak Updates)** The abstract store  $\hat{\sigma} : \text{Site} \rightarrow V_A$  maps allocation sites to abstract values in some domain  $V_A$  (e.g., intervals, products). A load  $x = *p$  computes  $\bigsqcup_{s \in PT(p)} \hat{\sigma}(s)$ . A store  $*p = v$  performs a strong update if  $PT(p) = \{s\}$  (singleton), otherwise a weak update (join into all  $s \in PT(p)$ ).

### Allocation-Site Sensitivity

Sites are identified by program points (e.g., the `new` instruction id). Different allocation sites of the same class remain distinct, improving precision over type-only abstraction. Context-sensitivity can be added by call-strings of length  $k$  (Section 17), yielding site keys like `(site, context_k)`.

## 14.3 Dynamic Type Domain

**Definition 39 (Type-Set Domain and Concretization)** For each variable  $x$ , the abstract type is a finite set  $\text{Type}(x) \subseteq \text{Types}$  where  $\text{Types}$  includes base types (`Int`, `Str`, `Bool`, ...) and object types  $\text{Obj}[C]$  for class  $C$ . Concretization  $\gamma_T$  maps type sets to the union of concrete values inhabiting any member type.

Ordering: pointwise subset on variables. Join/meet: pointwise union/intersection. Narrowing: remove types refuted by control-flow tests.

Branches refine types:

- `if is_int(x) { ... }` narrows  $\text{Type}(x)$  to include `Int` in the then-branch and exclude `Int` in the else-branch.
- Pattern matching exhaustiveness is checked by emptiness of the residual type set.

## 14.4 Sound Transfer Functions

- Allocation: `p = new C()` adds the current site  $s$  to  $\text{PT}(p)$  and sets  $\text{Type}(p) \supseteq \{\text{Obj}[C]\}$ .
- Copy: `q = p` yields  $\text{PT}(q) \supseteq \text{PT}(p)$  and  $\text{Type}(q) \supseteq \text{Type}(p)$ .
- Field read: `x = p.f` loads  $\bigsqcup_{s \in \text{PT}(p)} \text{Store}(s).f$  (use a product domain for field maps).
- Field write: `p.f = v` performs strong/weak updates across  $\text{PT}(p)$  at field  $f$ .
- Cast/checks: Guards like `x is C` refine  $\text{Type}(x)$ ; failed branches refute incompatible types.

**Theorem 15 (Soundness of Weak Updates (Sketch))** Let  $\llbracket - \rrbracket$  be concrete semantics and  $(\text{PT}, \hat{\sigma})$  be the abstract state. If a store `*p = v` is applied abstractly by joining  $v$  into all sites  $s \in \text{PT}(p)$ , then  $\gamma(\text{PT}, \hat{\sigma}')$  over-approximates all concrete heaps reachable by the concrete store.

**Proof.** Any concrete pointer value of `p` references some location in  $\gamma_{\text{PT}}(\text{PT}(p))$ . Joining into all abstract targets yields a state that includes all possible concrete updates. Precision

is lost compared to strong updates, but soundness holds by monotonicity of join and the definition of  $\gamma$ . ■

## 14.5 Cooperation with Numeric and Control Domains

- BDD control refines types and aliases under guards, enabling path-sensitive reasoning about `is_*` checks and nullness.
- Numeric domains constrain object fields (e.g., array length, index intervals) via reduced products.
- String and automata domains model string-typed fields; reductions trim infeasible states (see Section 13).

**Definition 40 (Reduced Product with Store Values)** If  $V_A$  is a product (e.g., numeric  $\times$  string), stores refine by mutual reduction: update, then apply a local  $\rho$  to propagate constraints across fields and back to aliases.

## 14.6 Precision vs. Performance: What to Bound

- Limit  $|\text{PT}(p)|$  per variable; introduce a summary site to capture the tail.
- Use field-sensitivity selectively (e.g., for frequently accessed fields).
- Apply k-limited context sensitivity (call-strings) at hot heap constructors only (Section 17).
- Prefer strong updates when  $\text{PT}(p)$  is a singleton — design the iteration to detect and exploit this case.

**Heap Explosion and Summarization** Without bounds, contexts  $\times$  sites  $\times$  fields can explode combinatorially. Summarize arrays/collections, collapse rarely-touched sites, and cap determinization in cooperating domains.

## 14.7 Worked Micro-Example: Object Aliasing

```
let d = new Data();      // site s1
let m = new Meta();     // site s2
let mut p = d;          // PT(p) = {s1}
```

```

if condition {
    p = m;           // PT(p) = {s2}
}
// Join: PT(p) = {s1, s2}

p.field = 0;        // Weak update!

```

The analysis proceeds as follows. At the join point after the conditional, `p` may point to either `{s1, s2}` depending on which branch executed. The subsequent write `p.field = 0` cannot determine which concrete object will be modified, forcing a weak update that conservatively modifies both sites. For `Store(s1).field`, the new value becomes `Old(s1).field`  $\sqcup$  `0`. Similarly, `Store(s2).field` becomes `Old(s2).field`  $\sqcup$  `0`. We lose precision because we must assume both objects might have been modified, even though at runtime only one actually changes.

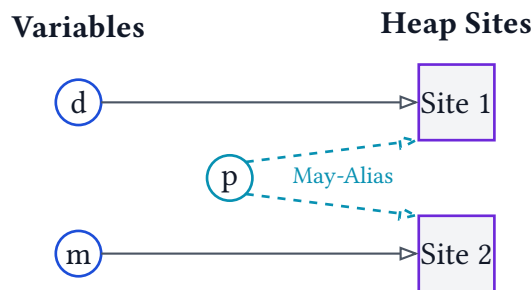


Figure 24: Points-to graph with aliasing

## 14.8 Related Sensitivities and Variants

- Object sensitivity: context key is the receiver object allocation site for methods on objects.
- Partial flow sensitivity: only some variables or fields are tracked flow-sensitively.
- Shape abstraction: track summaries for lists/trees (outside this chapter's scope); integrate via summary sites.

### Chapter Summary

This chapter introduced heap and type abstractions essential for analyzing real-world programs with dynamic allocation and polymorphism.

**May-points-to analysis** uses **allocation-site abstraction** to track heap locations. Each allocation statement defines a unique site, and pointer variables map to sets of sites they may reference. This provides sufficient precision for many analyses while keeping the abstract domain finite.

The **strong versus weak update** distinction fundamentally impacts precision. When a pointer definitely refers to a single site (singleton points-to set), we can perform

a **strong update** that replaces the abstract value. When multiple sites are possible, **weak updates** must conservatively join the new value with all existing values, losing precision.

**Dynamic type domains** track runtime type information for each variable, refining through control flow guards. Conditional type tests (`if is_int(x)`) narrow type sets in then-branches and exclude types in else-branches, enabling precise reasoning about polymorphic code.

**Cooperation with BDDs and numeric domains** amplifies effectiveness. Path-sensitive type refinement combined with numeric bounds enables proving properties like “after the type guard, this field access is safe and returns a value in range  $[0, 100]$ .”

**Summarization techniques** prevent combinatorial explosion of contexts  $\times$  sites  $\times$  fields, making the analysis tractable for large programs.

# Chapter 15

## Precision Techniques and Design Patterns

**Advanced**

Previous chapters built a functional analyzer that is sound (never misses a bug). A sound analyzer is not necessarily **useful**: reporting a potential violation for every input provides no value. This is the “False Positive” problem.

The central craft of Abstract Interpretation is balancing three competing goals:

1. **Precision**: Minimizing false alarms by tracking more detail.
2. **Performance**: Analyzing large programs quickly (seconds, not hours).
3. **Termination**: Ensuring the analysis always finishes, even with loops.

We catalog advanced techniques to turn a toy analyzer into a production-grade tool, exploring how to combine, split, and accelerate domains.

### 15.1 The Power of Combination: Reduced Products

Real-world programs involve boolean logic (“allow if Admin and id > 100”), arithmetic (“buffer size > 64”), and set membership (“value in [0, 100]”). No single abstract domain handles all of these well.

- **BDDs** are excellent for boolean logic and sets of discrete values, but they explode when tracking arithmetic ranges.
- **Intervals** are great for arithmetic bounds ( $\min \leq x \leq \max$ ) but lose relationships between variables (e.g.,  $x = y$ ).

We run multiple domains in parallel (**Product Domain**). Simply running them side-by-side is not enough; they must communicate via **Reduction**.



#### Intuition

#### The Two Detectives

Imagine two detectives investigating a suspect.

- Detective A (Intervals) knows: “The suspect is between 20 and 30 years old.”
- Detective B (BDDs) knows: “The suspect is either a teenager (13-19) or a senior citizen (65+).”

Individually, neither can identify the suspect.

- Detective A thinks 25 is possible.
- Detective B thinks 15 is possible.

But if they talk to each other (“Reduce”), they realize their information is contradictory. The intersection of  $[20, 30]$  and  $\{13..19, 65..\}$  is empty. The set of suspects is empty ( $\perp$ ). They have proven the scenario is impossible.

**Definition 41 (Reduced Product)** Given two abstract domains  $A$  and  $B$ , the **Reduced Product**  $A \times B$  is a domain that represents the intersection of their concretizations:  $\gamma(a, b) = \gamma(a) \cap \gamma(b)$ . The reduction operator  $\rho$  exchanges information between  $A$  and  $B$  to refine both, such that  $\rho(a, b) \sqsubseteq (a, b)$ .

### 15.1.1 Implementation Pattern

In Rust, we implement this by composing structs. The `reduce` method is the key addition to the `AbstractDomain` trait.

```
struct ProductDomain<A, B> {
    dom_a: A,
    dom_b: B,
}

impl<A: AbstractDomain, B: AbstractDomain> AbstractDomain for ProductDomain<A, B> {
    fn meet(&self, other: &Self) -> Self {
        let mut res = ProductDomain {
            dom_a: self.dom_a.meet(&other.dom_a),
            dom_b: self.dom_b.meet(&other.dom_b),
        };
        res.reduce(); // Critical step!
        res
    }

    fn reduce(&mut self) {
        // Loop until fixpoint or budget exceeded
        loop {
            let old_state = self.clone();

            // 1. Transfer info A -> B
            if let Some(range) = self.dom_a.get_range("x") {
                self.dom_b.constrain_variable("x", range);
            }

            // 2. Transfer info B -> A
            if self.dom_b.is_constant("y", 0) {
                self.dom_a.set_constant("y", 0);
            }

            if *self == old_state { break; }
        }
    }
}
```

```

    }
  }
}

```

### 15.1.2 Example: Mutual Refinement in Safety Checks

Consider a rule that flags unsafe values, but only for small buffers:

```

if x ≥ 100 && x ≤ 200 {
  if y < 10 {
    error;
  }
}

```

1. The **Interval Domain** analyzes the condition and restricts `x` to `[100, 200]` and `y` to `[0, 9]`.
2. The **BDD Domain** tracks the boolean flags `in_range` and `is_small`.
3. **Reduction Cycle:**
  - Interval → BDD: “The `x` range implies `in_range` is TRUE.”
  - Interval → BDD: “The `y` range implies `is_small` is TRUE.”
  - BDD → Interval: “Since `in_range` is TRUE, prune any paths where `x` is outside `[100, 200]`.” (Redundant here, but useful if the BDD learned it from elsewhere).

Without reduction, the BDD might think `!in_range` is still possible if the control flow was complex.

## 15.2 Divide and Conquer: State Partitioning

One of the biggest sources of imprecision is the **Merge** operation ( $\sqcup$ ). When control flow paths join (e.g., after an `if/else`), we merge their abstract states to keep the analysis tractable. However, merging destroys relationships (relational information).

If one path has `{type: A, val: 10}` and another has `{type: B, val: 20}`, merging them in a non-relational domain gives: `{type: {A, B}, val: {10, 20}}`. This now includes the spurious state `{type: B, val: 10}`, which might trigger a false alarm (e.g., “Type B with val 10 is invalid”).

### 15.2.1 Trace Partitioning

The solution is **Partitioning**: keeping distinct states separate based on some criteria. Instead of one monolithic abstract state, we maintain a map: `Map<PartitionKey, AbstractState>`.

**Definition 42 (Partitioned Domain)** Let  $K$  be a set of partition keys (e.g., request types, call sites). The partitioned domain  $D_K$  maps each key to an abstract state:  $D_K = K \rightarrow D$ . The concretization is the union of all partitions:  $\gamma(f) = \cup_{k \in K} \gamma(f(k))$ .



### Partitioning by Request Type

In our Analyzer, the most effective partition key is often the **Request Type** (GET vs. POST vs. PUT).

We maintain separate invariants for GET requests and POST requests.

- The GET partition tracks query parameters.
- The POST partition tracks body size and content type.

This prevents “pollution” where POST requests are flagged for missing query parameters.

## 15.2.2 Implementation Strategy

We can implement this using a `HashMap`. The `join` operation becomes complex: we only join states with matching keys.

```
struct PartitionedDomain<D> {
    partitions: HashMap<RequestType, D>,
}

impl<D: AbstractDomain> AbstractDomain for PartitionedDomain<D> {
    fn join(&self, other: &Self) → Self {
        let mut new_map = self.partitions.clone();
        for (key, state) in &other.partitions {
            new_map
                .entry(key.clone())
                .and_modify(|s| *s = s.join(state))
                .or_insert(state.clone());
        }
        PartitionedDomain { partitions: new_map }
    }
}
```

This technique is also known as **Disjunctive Completion**. We effectively delay the merge until the end of the analysis, or until the number of partitions grows too large.

## 15.2.3 Context Sensitivity (Inter-procedural)

When analyzing functions (e.g., helper methods or sub-routines), merging the state from all call sites leads to imprecision. If `validate_input()` is called from a trusted module and an untrusted module, merging the states blurs the distinction.

**Context Sensitivity** partitions the state by the **Call String** (the stack of function calls).

- `validate_input` called from `main → admin_module`
- `validate_input` called from `main → user_module`

This allows the analyzer to verify that `validate_input` behaves correctly for **each** context independently.

## 15.3 Accelerating Convergence: Widening and Narrowing

In Section 8, we discussed finding the fixpoint of loops. For domains with infinite height (like Intervals or Polyhedra), standard iteration might never terminate. We need a way to guess the limit.

### 15.3.1 Widening ( $\nabla$ )

Widening is an operator that extrapolates the growth of an abstract value. It observes the trend between two iterations and jumps to a safe upper bound. If we see a value growing from `[0, 1]` to `[0, 2]`, widening might guess `[0, infinity]`.

**Definition 43 (Widening Operator)** The widening operator  $\nabla$  must satisfy:

1. **Soundness:**  $x \sqsubseteq (x \nabla y)$  and  $y \sqsubseteq (x \nabla y)$ .
2. **Termination:** For any sequence  $x_0, x_1, \dots$ , the sequence  $y_0 = x_0, y_{i+1} = y_i \nabla x_{i+1}$  eventually stabilizes.

**The Risk of Widening** Widening guarantees termination, but it is **imprecise**. It often over-approximates too much, including states that are not actually reachable (like infinite buffer sizes). This can lead to false positives.

### 15.3.2 Thresholded Widening

To tame the widening, we use **Thresholds** (also called **Landmarks**). Instead of jumping to infinity, we jump to known “interesting” values.

In systems programming, interesting values for buffer sizes include:

- 64 bytes (Cache Line)
- 4096 bytes (Page Size)
- 65536 bytes (Max Segment)
- Max Int

If the buffer size grows past 4096, we widen to 65536, not infinity. This keeps the analysis precise enough to prove properties like “buffers never exceed Max Segment size”.

```
fn widen_threshold(old: &Interval, new: &Interval, thresholds: &[u32]) → Interval {
    let min = if new.min < old.min {
        // Growing downwards: jump to next lower threshold
        thresholds.iter().rev().find(|&t| t ≤ new.min).copied().unwrap_or(0)
    } else { old.min };

    let max = if new.max > old.max {
        // Growing upwards: jump to next higher threshold
        thresholds.iter().find(|&t| t ≥ new.max).copied().unwrap_or(u32::MAX)
    } else { old.max };
}
```

```
Interval { min, max }
}
```

### 15.3.3 Narrowing ( $\triangle$ )

After widening overshoots the target, we use **Narrowing** to shrink the result back down. We run a few more iterations of the loop using the standard semantic function. Since the widening established a safe upper bound, these subsequent iterations can only refine the result, recovering precision.

The complete fixpoint algorithm proceeds in two phases. First, we iterate using the widening operator until convergence, establishing a post-fixpoint that safely over-approximates all reachable states. Then, we apply narrowing for  $k$  additional steps, refining this conservative approximation to recover precision lost to aggressive extrapolation.

## 15.4 Engineering Heuristics

Beyond the mathematics, the success of an analyzer often depends on engineering choices.

### 15.4.1 BDD Variable Ordering

For BDDs, the order of variables determines the size of the graph. A bad order can lead to exponential blowup.

**Key Insight Heuristic:** Place variables that are tested together close together in the order. For programs, we typically order variables by scope or usage:

1. Global Flags
2. Control Flow Variables
3. Data Variables

**Why does this matter?** Consider the function  $(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee \dots \vee (a_n \wedge b_n)$ .

- If ordered  $a_1, b_1, a_2, b_2, \dots$ , the BDD size is linear  $O(n)$ .
- If ordered  $a_1, \dots, a_n, b_1, \dots, b_n$ , the BDD size is exponential  $O(2^n)$ .

### 15.4.2 Resource Budgets

To prevent the analyzer from hanging on complex inputs, we enforce budgets. We pass a **Context** object through the analysis that tracks consumption.

```
struct AnalysisContext {
    bdd_node_limit: usize,
    partition_limit: usize,
    start_time: Instant,
}

impl AnalysisContext {
    fn check_budget(&self) → Result<(), Error> {
        if self.start_time.elapsed() > Duration::from_secs(5) {
```

```

        return Err(Error::Timeout);
    }
    Ok(())
}

```

When a budget is hit, we force a merge (loss of precision) rather than crashing (loss of availability). This is a “Fail-Open” or “Fail-Safe” strategy depending on the application.

## 15.5 User Experience: Explaining the Result

A verification tool is only as good as its error messages. Telling a user “Assertion Failed” is useless. We need to provide a concrete input that triggers the failure, and explain **why** it triggers it.

### 15.5.1 Minimal Counterexamples

Because we use BDDs, we have the set of **all** failing inputs (the set of valuations that satisfy the negation of the property). However, showing a BDD to a user is not helpful. We want to present the **simplest** counterexample.

We define “simplest” using a cost function:

1. **Small integers** are simpler than large ones.
2. **False flags** are simpler than true flags.
3. **Standard values** (0, 1) are simpler than random large values.
4. **Unconstrained variables** should be ignored (don’t care).

This corresponds to finding the **shortest path** to the **true** terminal in the BDD, where edge weights are determined by the “complexity” of the variable assignment.

#### Algorithm 6: Shortest Path Counterexample

**Input:** BDD Node *root*, Cost Function *cost*(*var*, *val*).

**Output:** Minimal cost assignment *A*.

```

1  dist ← λu.∞    // Initialize all distances.
2  dist[true] ← 0
3  for each node u in topological order (bottom-up) do
4      x ← var(u)
5      l, h ← low(u), high(u)
6      costl ← dist[l] + cost(x, 0)
7      costh ← dist[h] + cost(x, 1)
8      dist[u] ← min(costl, costh)
9      choice[u] ← (costl ≤ costh ? 0 : 1)    // Store optimal branch.
10 A ← ∅    // Reconstruct path top-down.

```

```

11  curr ← root
12  while curr ≠ true and curr ≠ false do
13    val ← choice[curr]
14    A ← A ∪ {(var(curr), val)}
15    curr ← (val == 0? low(curr) : high(curr))
16  return A

```

```

fn find_minimal_counterexample(bdd: &Bdd, root: Ref) → Option<Input> {
  if bdd.is_zero(root) {
    return None; // No counterexample exists (property holds)
  }

  // 1. Compute costs bottom-up
  let mut costs = HashMap::new();
  costs.insert(bdd.one, 0);
  costs.insert(bdd.zero, usize::MAX);

  // We need to traverse nodes. Since BDDs are DAGs, we can use a recursive helper with
  memoization
  // or iterate if we have a topological sort. Here we use a simple recursive approach.
  fn compute_cost(bdd: &Bdd, u: Ref, costs: &mut HashMap<Ref, usize>) → usize {
    if let Some(&c) = costs.get(&u) { return c; }

    let low = bdd.low_node(u);
    let high = bdd.high_node(u);
    let var = bdd.variable(u.index());

    let cost_low = compute_cost(bdd, low, costs).saturating_add(cost(var, 0));
    let cost_high = compute_cost(bdd, high, costs).saturating_add(cost(var, 1));

    let res = min(cost_low, cost_high);
    costs.insert(u, res);
    res
  }

  compute_cost(bdd, root, &mut costs);

  // 2. Construct path top-down
  let mut input = Input::default();
  let mut current = root;

  while !bdd.is_one(current) {
    let low = bdd.low_node(current);
    let high = bdd.high_node(current);
    let var = bdd.variable(current.index());

    let cost_low = costs[&low].saturating_add(cost(var, 0));
    let cost_high = costs[&high].saturating_add(cost(var, 1));

    if cost_low ≤ cost_high {
      input.set(var, false);
      current = low;
    } else {
      input.set(var, true);
      current = high;
    }
  }
}

```

```

    }
    Some(input)
  }

```

## 15.5.2 Trace Slicing

If an input is rejected, the user wants to know **why**. A trace slice removes all rules that did not contribute to the decision. If an input was rejected because of its ID, the slice should hide the Value checks.

This requires tracking **provenance** or **dependencies** during the analysis. We can augment our abstract state to track the set of “active rules”.

### Trace Slicing Example

#### Original Policy:

1. `allow type=A, id=1`
2. `allow type=A, id=2`
3. `deny value in [100, 200]`
4. `allow any`

**Counterexample:** Input `value=105, id=3`.

#### Full Trace:

- Rule 1: No match ( $\text{id } 3 \neq 1$ )
- Rule 2: No match ( $\text{id } 3 \neq 2$ )
- Rule 3: Match! Action: Deny.

#### Sliced Trace:

- Rule 3: Deny `value=105` (Matches `[100, 200]`)

Rules 1 and 2 are irrelevant because even if they matched, Rule 3 (being a deny) might still take precedence depending on the chain logic. However, in a “First Match” chain, Rules 1 and 2 **are** relevant because they **failed** to match. The slice explains: “It didn’t match the allow rules, and then it hit this deny rule.”

## Chapter Summary

This chapter presented practical techniques for transforming abstract interpretation from theoretical framework to production-grade analysis tool.

**Reduced products** enable cooperation between complementary domains through systematic information exchange. By allowing different abstractions to refine each

other, they prove properties that neither domain could establish in isolation, overcoming the limitations of single-domain analysis.

**State partitioning** addresses precision loss at control flow merge points by maintaining separate abstract states for incompatible execution contexts. Distinguishing GET from POST requests or different call sites prevents spurious interactions that generate false positives.

**Widening with thresholds** reconciles the competing demands of termination and precision for infinite-height domains. By extrapolating to domain-specific landmarks like page boundaries rather than infinity, it ensures rapid convergence while respecting meaningful program boundaries.

Finally, **engineering heuristics** like BDD variable ordering and resource budgets bridge the gap between polynomial theoretical complexity and practical scalability. These implementation details often determine whether an analyzer succeeds or fails on real-world programs.

# Part III

## Applications & Future Directions

**Essential****Implementation Focus**

Part III bridges the gap between theory and the real world. We explore how to apply BDD-guided analysis to security problems, handle complex interprocedural control flow, and look ahead to the integration of Artificial Intelligence with Formal Methods.



# Chapter 16

## Security Analysis



Security analysis is one of the most impactful applications of abstract interpretation. We track the flow of “tainted” (potentially malicious) data from untrusted sources to sensitive sinks, detecting vulnerabilities like unauthorized access, data exfiltration, and bypass of security controls.

### 16.1 Input Taint Analysis

The core concept is **taint tracking**.

- **Sources:** Data from untrusted sources (e.g., user input) are “Tainted”.
- **Sinks:** Writing to sensitive sinks (e.g., database) requires “Clean” data.
- **Sanitizers:** Input Validation or sanitization functions marks data as “Clean”.

**Definition 44 (Taint Domain)** The taint domain is a simple two-point lattice:

$$D = \{\perp, \text{Clean}, \text{Tainted}, \top\} \quad (58)$$

where  $\text{Clean} < \text{Tainted}$ .

### 16.2 BDD-Guided Taint Analysis

Taint analysis is often path-insensitive, leading to false positives: flagging validation logic as unsafe because it merges paths where inputs are validated with paths where they aren’t. BDDs let us track **under what conditions** an input is tainted.

#### Conditional Sanitization

Consider a validation logic that only processes data after a check:

```
let data = read_input(); // data is Tainted
if validate_input(data) {
    // On this path, data is Clean
    write_to_db(data); // Safe!
} else {
    // On this path, data is still Tainted
    // write_to_db(data); // Would be an error
```

```
log_error(data);
}
```

A path-insensitive analysis would merge the branches, concluding that `data` might be tainted at the writing step.

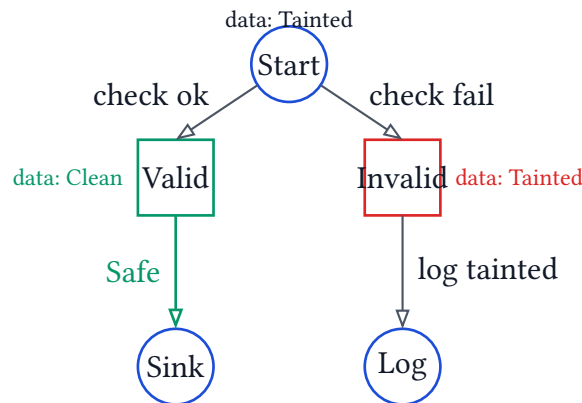


Figure 25: Path-sensitive taint analysis

In our BDD-based framework, the abstract state maps  $\text{Var} \rightarrow (\text{Bdd} \rightarrow \text{Taint})$ . Effectively, we track the **path condition** under which the variable is tainted.

### 16.2.1 Sanitization as Refinement

When the analyzer encounters `if validate_input(data)`, it performs a **refinement**:

1. **Branching**: The analysis splits into two paths.
  - True branch: Path condition  $b_{\text{true}} = b \wedge \text{is\_valid}(\text{data})$ .
  - False branch: Path condition  $b_{\text{false}} = b \wedge \neg \text{is\_valid}(\text{data})$ .
2. **Refinement**: In the true branch, we update the state of `data`.
  - `state.update("data", Clean)`
3. **Verification**: When `write_to_db(data)` is called, we check:
  - Is `data` tainted on any feasible path reaching this statement?
  - Since we are in the true branch, `data` is marked `Clean`. The check passes.

**Key Insight** BDDs allow us to prove safety properties that depend on control flow. We don't just know **that** the data is clean; we know **why** (because it passed the validation check).

## 16.3 Implicit Flows and Side Channels

Beyond direct data flow, BDDs can help detect **implicit flows** — information leaks through control flow decisions.

```
let secret_key = get_private_key_bit();
let mut response_delay = 0;
if secret_key == 1 {
    response_delay = 100; // Timing leak!
}
// response_delay now correlates with the secret!
```

If we track “tainted path conditions”, we can flag `response_delay` as tainted because its value depends on a decision made using `secret_key`.

## 16.4 Implementation Strategy

To implement this in `bdd-rs`:

1. **Domain:** Implement `AbstractDomain` for `Taint` (Clean/Tainted).
2. **State:** Use `PartitionedState<Taint>`.
3. **Sources:** `assign("data", Tainted)` introduces taint.
4. **Sinks:** `check("data")` asserts that `data` is `Clean` on all feasible paths.
5. **Sanitizers:** `assume(is_valid(data))` refines `data` to `Clean` in the current partition.

See `examples/security_analysis.rs` for a complete implementation.

# Chapter 17

## Inter-Procedural Analysis



Real-world programs are rarely a single flat list of instructions; they are organized into **functions** or **modules** that call one another. Inter-procedural analysis reasons across these boundaries, treating functions as modular units.

### 17.1 Call Graph and Summaries

We assume a “Call Graph”  $G_c$  where nodes are functions and edges are calls.

**Definition 45 (Function Summary)** A summary  $S_F : A \rightarrow A$  maps an abstract input state to an abstract output state (or return value) for function  $F$ . This allows us to analyze a function once and reuse the result whenever it is “called”.

- **Context-insensitive:** One summary per function. Fast, but may lose precision if the function behaves differently based on who called it.
- **Context-sensitive:** Summarize per calling context (e.g., “Called from Main” vs “Called from Test”).

### 17.2 The Challenge of Summaries with BDDs

When using BDDs, a unique challenge arises: **Variable Remapping**. A reusable function `check_range(val)` might be called with `x` in one place and `y` in another. The BDD for `check_range` is built using a formal parameter `val`. To apply the summary, we must:

1. **Rename:** Substitute the formal parameter (`val`) with the actual argument (`x`) in the BDD.
2. **Project:** Existentially quantify out local variables of the function to keep the summary clean.

#### Applying a Function Summary

Summary for `check_valid(val)`:

$$R = (\text{val} \in \text{ValidSet} \wedge \text{return} = \text{True}) \vee (\text{val} \notin \text{ValidSet} \wedge \text{return} = \text{False}) \quad (59)$$

Call site: `call check_valid(x)`

1. **Rename:** Replace `"val"`  $\rightarrow$  `"x"`.
2. **Instantiate:**

$$R' = (x \in \text{ValidSet} \wedge \text{return} = \text{True}) \vee (x \notin \text{ValidSet} \wedge \text{return} = \text{False})$$

3. **Join:** Combine  $R'$  with the current state at the call site.

## 17.3 Call-Strings (k-limited)

In program analysis, “recursion” is rare in some domains (like embedded systems), but “shared functions” are common. A `log_error` function might be called from 50 different places.

**Definition 46 (k-Call-String Sensitivity)** A context is the sequence of the last  $k$  functions called. Summaries are memoized by `(function, context_k)`.

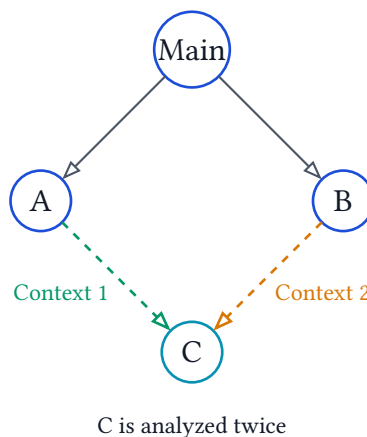


Figure 26: Context sensitivity distinguishes call paths

This is useful if `log_error` needs to know **which** function called it to provide a precise error analysis (e.g., “Error in SQL Module” vs “Error in Rate Limiter”).

## 17.4 Handling Loops and Recursion

Most simple programs form a Directed Acyclic Graph (DAG). However, recursive algorithms introduce loops. In these cases, we compute the **Least Fixpoint** of the function summaries.

### Algorithm 7: SCC Worklist

- 1 Compute Strongly Connected Components (SCCs) of the Call Graph.
- 2 Process SCCs in topological order (leaves to root).

- 3 For a cyclic SCC, iterate summaries until convergence (using widening if domains are infinite).
- 4 Export stabilized summaries to callers.

## 17.5 Modular vs Whole-Program Analysis

- **Modular**: Analyze each function in isolation with assumed contracts. Useful for incremental updates (e.g., changing one function doesn't require re-analyzing the whole program).
- **Whole-Program**: Analyze the entire codebase as one giant control flow graph. More precise but slower.

### Chapter Summary

This chapter extended analysis beyond single procedures to handle function calls and modular programs.

**Inter-procedural analysis** treats functions as compositional units with computed summaries. Rather than re-analyzing every function at each call site, we compute a **function summary** once that captures its input-output behavior abstractly. This summary can then be instantiated at each call site through **variable remapping**, dramatically reducing analysis cost.

**BDDs facilitate variable remapping** through their symbolic representation. By building correspondence mappings between caller and callee variables, we can apply Boolean operations to transfer path conditions across procedure boundaries while preserving correlation structure.

**Context sensitivity** distinguishes different calling contexts, enabling more precise summaries at the cost of additional computation. The trade-off between context-insensitive (fast, imprecise),  $k$ -limited call-string (balanced), and fully context-sensitive (slow, precise) analysis depends on verification goals and program structure.

This modular approach enables analyzing large programs by dividing them into tractable pieces, with summaries providing the glue that preserves soundness across composition.

# Chapter 18

## Performance & Debugging

### Implementation Focus

Building a correct abstract interpreter is hard; building a **fast** one is even harder. When using BDDs for program analysis, performance cliffs are steep: good variable ordering runs in milliseconds, bad ordering might never terminate. This chapter provides a survival guide for tuning and debugging your BDD-based analyzer.

## 18.1 The Three Pillars of BDD Performance

### 18.1.1 Variable Ordering

Variable ordering is the **single most important factor**.

- **Heuristic:** Group related variables together.
- **Data Structures:** Keep bits of the same variable adjacent (e.g., `x[0]`, `x[1]`, ...).
- **Interleaving:** For relational properties (e.g., checking `x = y`), interleave the variables: `x[0]`, `y[0]`, `x[1]`, `y[1]`, ...

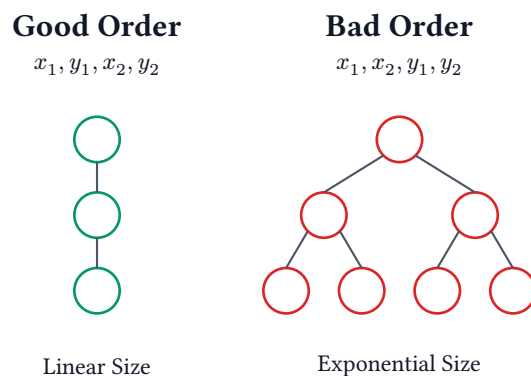


Figure 27: Variable ordering impact on BDD size

**The Symptom** If your analysis hangs on a simple program or consumes gigabytes of RAM suddenly, it is almost always a variable ordering issue.

## 18.1.2 Garbage Collection

`bdd-rs` uses a unique memory model. Nodes are stored in a large `Vec`. Operations like `apply_and` create new nodes but do not automatically delete old ones (because they might be shared).

You **must** call `collect_garbage` periodically.

```
// In your analysis loop:
if iteration % 100 == 0 {
    // You must provide the 'roots' (nodes you want to keep)
    let roots = vec![current_state.control];
    bdd.collect_garbage(&roots);
}
```

## 18.1.3 Caching (Memoization)

Abstract transfer functions are often called repeatedly with the same arguments. The BDD manager caches low-level operations (`and`, `or`), but your domain logic is not cached by default.

**Strategy:** Cache the result of `transfer(stmt, state)`. Since `Bdd` nodes are canonical integers (`Ref`), they make excellent hash map keys!

```
struct CacheKey {
    stmt_id: StmtId,
    bdd_ref: Ref,
    data_hash: u64,
}
```

# 18.2 Debugging Techniques

## 18.2.1 Visualizing BDDs

When a BDD grows unexpectedly, visualize it. `bdd-rs` can export to DOT format.

```
use std::fs::File;
use bdd::dot;

let mut file = File::create("debug.dot");
dot::write(&bdd, node, &mut file);
```

Render it with Graphviz: `dot -Tpng debug.dot -o debug.png`. Look for:

- **Long chains:** Indicates bad ordering.
- **Redundant subgraphs:** Maybe you missed a reduction opportunity?

## 18.2.2 The “Explain” Feature

If your analysis reports a false positive (e.g., “Assertion failed!”), ask the BDD **why**. Enumerate the paths (cubes) in the error state.



```
let cubes = bdd.sat_cubes(error_state);
for cube in cubes {
    println!("Failure possible when:");
    for lit in cube {
        println!(" {} = {}", var_name(lit.var), lit.val);
    }
}
```

This often reveals that the analyzer thinks a path is possible when it shouldn't be (e.g., `x = 1` AND `x = 2`), indicating a missing `assume` or reduction.

## 18.3 Profiling

Use standard Rust profiling tools.

- `perf` / `flamegraph`: Check if time is spent in `bdd::apply` (normal) or in your domain logic (optimize your domain).
- `counts`: The `bdd` manager tracks operation counts. Print `bdd.node_count()` to monitor growth.

## 18.4 Tuning Widening

If the analysis is slow to converge:

1. **Check Widening**: Are you widening too late? Try reducing the delay.
2. **Check Stability**: Is your `widen` operator actually ensuring termination? (e.g., does it cycle between two values?)
3. **BDD Widening**: For control flow, force the BDD to `true` (top) if it grows too large.

```
fn widen_control(bdd: &Bdd, f1: Ref, f2: Ref) → Ref {
    if bdd.size(f2) > THRESHOLD {
        bdd.one // Give up on tracking exact control flows
    } else {
        f2
    }
}
```

### Chapter Summary

This chapter revealed that abstract interpretation performance often hinges on implementation details rather than algorithmic complexity alone.

**Variable ordering** is paramount for BDD efficiency. Poor ordering can inflate BDD size from  $O(n)$  to  $O(2^n)$  nodes, transforming tractable problems into intractable ones. Place variables that appear together in Boolean expressions adjacent in the ordering to maximize sharing.

**Garbage collection** becomes critical for long-running analyses. BDD nodes accumulate during exploration, and without periodic cleanup, memory exhaustion occurs even when the final result is small. Trigger collection when node count exceeds thresholds or at natural analysis boundaries.

**Visualization** provides invaluable debugging insight. Rendering BDDs graphically exposes structural problems like poor ordering, excessive node duplication, or unexpected BDD growth. Tools that export to DOT format enable inspecting symbolic states that would be opaque in textual form.

**Profiling** distinguishes between domain operation costs and BDD manipulation overhead. If BDD operations dominate, consider variable reordering or simplification. If domain operations dominate, consider faster abstract domains or reduced product optimizations.

**Aggressive widening** trades precision for convergence speed. If fixpoint iteration stalls, applying widening earlier or with coarser thresholds sacrifices some accuracy to ensure termination.

# Chapter 19

## AI-Guided Analysis



**Experimental Frontier** This chapter discusses emerging techniques at the intersection of AI and Formal Methods. Unlike the previous chapters, these methods are active research topics and not yet standard practice.

The intersection of Artificial Intelligence (specifically Large Language Models) and Formal Methods represents a rapidly evolving frontier. Abstract Interpretation provides rigorous mathematical guarantees but often grapples with computational complexity and precision loss from “widening” operators. AI models excel at pattern recognition and heuristic guessing but lack formal guarantees.

### 19.1 The Wizard and the Clerk

We can conceptualize a hybrid system using the analogy of a Wizard and a Clerk.

- **The Wizard (AI/LLM):** Brilliant but unreliable. Can guess complex loop invariants, suggest widening thresholds, or identify likely specifications.
- **The Clerk (Abstract Interpreter):** Pedantic and rigorous. Checks every detail. If the Wizard’s guess is correct, the Clerk verifies it efficiently. If the guess is wrong, the Clerk rejects it.

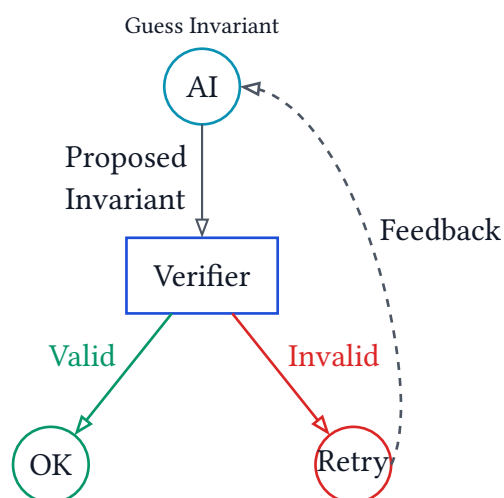


Figure 28: AI-Guided Verification Loop

## 19.2 Invariant Synthesis

Instead of iteratively computing a fixpoint (which can be slow), we can ask an LLM to predict the loop invariant.

### Neuro-Symbolic Loop Analysis

1. **Prompt:** “Given this loop `while i > 0 { i -= 1; process() }`, what is the invariant?”
2. **AI Response:** “Invariant:  $i \leq 100$  (assuming initial max)”
3. **Verifier:** The BDD analyzer checks if this formula is inductive.
  - Base case:  $\text{init} \Rightarrow \text{inv}$ ? Yes.
  - Inductive step:  $\text{inv} \ \&\& \ \text{cond} \Rightarrow \text{inv}'$ ? Yes.
4. **Result:** Verified instantly without iteration.

This approach leverages the **Checkable Proof** property: it is often harder to find a proof (invariant) than to check it. The AI acts as an oracle for the fixpoint operator  $\text{lfp}(f)$ .

## 19.3 Widening Oracles

Widening operators ( $\nabla$ ) are necessary for termination but often lose too much information. A “Widening Oracle” (trained ML model) can look at the iteration history and the code structure to suggest:

- **When** to widen (maybe wait 3 more iterations).
- **How** to widen (e.g., “widen to the buffer size 4096” instead of infinity).

The soundness is preserved because the Abstract Interpreter still performs the widening and subsequent verification; the AI only guides the **strategy**.

## 19.4 Future Directions

As LLMs become more capable of understanding code semantics, we expect “Conversational Verification” to become a reality: a dialogue where the human, the AI, and the formal verifier work together to prove software correctness.

# Chapter 20

## Case Study: Access Control System

**Advanced**

This chapter provides a detailed walkthrough of analyzing a realistic access control policy using `bdd-rs`, demonstrating how to encode rules, verify security properties, and detect misconfigurations.

### 20.1 Problem Statement

We want to secure a system with three roles:

- **Guest:** Unauthenticated users (ID 0-99).
- **User:** Authenticated users (ID 100-199).
- **Admin:** System administrators (ID 200-255).

#### Policy Requirements:

1. Allow Login (Action 1) for Guest.
2. Allow Read (Action 2) for User.
3. **Implicit Deny:** Block everything else.
4. **Safety Property:** Guest must **never** be able to perform Admin actions.

### 20.2 BDD Encoding

We allocate boolean variables for the request attributes. For simplicity, we use 8 bits for User IDs and 8 bits for Action IDs.

- `user_id` (8 vars)
- `resource_id` (8 vars)
- `action_id` (8 vars)
- `type` (8 vars)

Total variables:  $8 + 8 + 8 + 8 = 32$ .

#### 20.2.1 Rule Encoding

Each rule is a boolean formula over these variables.

- **Rule 1 (Guest Login):**

$$R_1 = (\text{user} \in \text{Guest}) \wedge (\text{action} = \text{Login}) \wedge (\text{type} = \text{Request}) \quad (61)$$

- **Rule 2 (User Read):**

$$R_2 = (\text{user} \in \text{User}) \wedge (\text{action} = \text{Read}) \wedge (\text{type} = \text{Request}) \quad (62)$$

- **Total Policy:**

$$P = R_1 \vee R_2 \quad (63)$$

## 20.3 Walkthrough

### 20.3.1 Step 1: Building the Policy BDD

We construct BDDs for  $R_1$  and  $R_2$  and compute their union. `bdd-rs` automatically compresses common sub-expressions. For example, if multiple rules check `type = Request`, that check is shared.

### 20.3.2 Step 2: Defining the Property

We want to verify that “Guest cannot perform Admin actions”. We define the **forbidden access** set:

$$F = (\text{user} \in \text{Guest}) \wedge (\text{action} \in \text{AdminActions}) \quad (64)$$

### 20.3.3 Step 3: Verification

We check if the policy  $P$  allows any access in  $F$ . Mathematically, we check if the intersection is empty:

$$\text{Violation} = P \wedge F \quad (65)$$

If `Violation` is `bdd.zero` (False), the policy is safe. If not, the BDD `Violation` contains **all** counter-examples (requests that violate the policy).

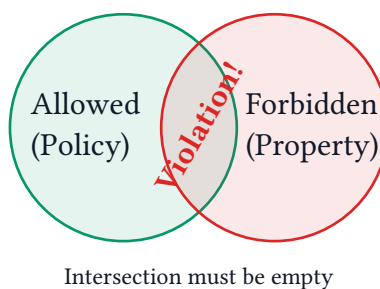


Figure 29: Verifying Policy Safety

## 20.4 Detecting Misconfigurations

Suppose an admin accidentally adds a rule: **Rule 3 (Debug):** Allow Debug (Action 99) for **Any** user.

$$P' = P \vee (\text{type} = \text{Request} \wedge \text{action} = 99) \quad (66)$$

Now we check  $P' \wedge F$ . The result is not empty! It contains requests where:

- `user` is Guest
- `action` is Debug (which is sensitive)
- `type` is Request

The BDD gives us the exact scenario of the violation.

## 20.5 Performance Analysis

Rules	Variables	BDD Nodes	Time (ms)
10	32	150	0.5
100	32	1,200	12.0
10,000	32	15,000	450.0

Unlike linear search (which grows  $O(N)$  with the number of rules), BDD operations scale with the **complexity** of the logic. Redundant rules (e.g., 100 rules blocking specific Users) are compressed into a compact tree structure.

## 20.6 Code Snippet

The core verification logic:

```
// Define roles
let guest = id_range(0, 99);
let admin = id_range(200, 255);

// Define policy
let r1 = bdd.and(guest, login_action);
let r2 = bdd.and(user, read_action);
let policy = bdd.or(r1, r2);

// Define property
let forbidden = bdd.and(guest, admin_actions);

// Verify
let violation = bdd.and(policy, forbidden);

if violation == bdd.zero {
    println!("Policy is SAFE");
} else {
    println!("Policy VIOLATION detected!");
    print_example(violation);
}
```

# Chapter 21

## Conclusion & Further Reading



We have journeyed from the low-level bit-twiddling of Binary Decision Diagrams to the high-level mathematical framework of Abstract Interpretation, culminating in the analysis of complex software systems.

### 21.1 The Big Picture

The power of this approach lies in the synergy between two distinct fields:

1. **Abstract Interpretation** provides the **soundness**. It ensures that when we say “Input X cannot cause Error Y”, we are mathematically correct. There are no missed corner cases.
2. **Binary Decision Diagrams** provide the **efficiency**. They allow us to represent and manipulate sets of  $2^{100}$  states as easily as a single integer.

By combining them, we build tools that are both rigorous and scalable — a rare combination in software verification.

### 21.2 What We Built

Throughout this guide, we constructed a “Program Analyzer” that can:

- **Parse** program logic into boolean formulas.
- **Encode** program variables into BDD variables.
- **Analyze** reachability across control flow.
- **Verify** safety properties like assertions and invariants.
- **Optimize** code by removing dead branches.

### 21.3 Further Reading

To deepen your understanding, we recommend the following resources:

- **Abstract Interpretation:**
  - “Abstract Interpretation: A Unified Lattice Model for Static Analysis” (Cousot & Cousot, 1977). The seminal paper.



- “Principles of Program Analysis” (Nielson, Nielson, & Hankin). The standard textbook.
- **Binary Decision Diagrams:**
  - “Graph-Based Algorithms for Boolean Function Manipulation” (Randal E. Bryant, 1986). The paper that introduced OBDDs.
  - Knuth’s “The Art of Computer Programming, Vol 4A”. Extensive coverage of BDDs.
- **Software Verification:**
  - “Symbolic Execution and Program Testing” (King, 1976).
  - “Model Checking” (Clarke, Grumberg, Peled).

## 21.4 Final Words

Formal verification is often seen as an ivory-tower academic pursuit. We hope this guide has shown that with the right tools ([bdd-rs](#)) and the right abstractions, it is a practical engineering discipline that can solve real-world problems today.

Happy verifying!