Software Verification

Gabriel Rovesti

April 20, 2025

Abstract

This comprehensive guide covers the fundamentals of software verification, structured around four main pillars: program semantics, abstract interpretation, dataflow analysis, and verification tools. The material progresses from theoretical foundations of program behavior modeling through practical applications in static analysis and verification. Formal definitions and theorems are complemented with examples and exercises to facilitate deeper understanding of these essential topics in computer science.

Contents

Chapter 1

Program Semantics

1.1 Introduction to Software Verification

Software verification encompasses formal techniques for proving or disproving the correctness of software systems with respect to specified properties. Unlike testing, which can only demonstrate the presence of bugs but not their absence, formal verification aims to mathematically prove that a system satisfies its requirements under all possible scenarios.

The field combines theoretical computer science with practical engineering concerns, addressing questions such as:

- Does the program terminate for all valid inputs?
- Does the program correctly implement its specification?
- Are there any inputs that could cause unexpected behavior?
- Can we guarantee the absence of certain classes of errors?

This course focuses on four interconnected areas:

1. **Program Semantics**: Mathematical models that precisely define the meaning and behavior of programs.
2. **Abstract Interpretation**: A framework for approximating program semantics to make analysis computationally feasible.
3. **Dataflow Analysis**: Techniques to determine information about program states at different program points.
4. **Verification Tools**: Practical implementations of these theories to verify real-world software.

1.2 Program Semantics

Program semantics provides formal, mathematical models of the meaning of programs. These models allow us to reason rigorously about program behavior, establish equivalence between programs, and verify correctness properties.

1.2.1 Approaches to Semantics

There are three main approaches to defining program semantics:

- **Operational Semantics**: Describes program execution as step-by-step state transitions.
- **Denotational Semantics**: Maps programs directly to mathematical functions representing their input-output behavior.
- **Axiomatic Semantics**: Focuses on logical assertions about program states before and after execution.

The choice of semantic approach depends on the properties we wish to analyze and the level of abstraction appropriate for our goals.

1.2.2 Syntax of the While Language

Throughout this course, we'll use a simple imperative language called While as our primary example. The While language has the following syntax:

$$S ::= x := a \mid \text{skip} \mid S_1; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S$$

Where:

- a represents arithmetic expressions
- b represents boolean expressions
- x represents variables

Syntactic Categories

The complete syntactic categories for While are:

- Numerals: $n \in \text{Num}$
- Variables: $x \in \text{Var}$
- Arithmetic expressions: $a \in \text{Aexp}$

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2$$

- Boolean expressions: $b \in \text{Bexp}$

$$b ::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2$$

- Statements: $S \in \text{Stm}$

$$S ::= x := a \mid \text{skip} \mid S_1; S_2 \mid \text{if } b \text{ then } S_1 \text{ else } S_2 \mid \text{while } b \text{ do } S$$

1.2.3 Semantic Categories

To define the meaning of programs, we need to establish mathematical structures that represent the computational concepts:

- Natural numbers: $\mathbb{N} = \{0, 1, 2, \ldots\}$
- Truth values: $\mathbb{T} = \{\texttt{tt}, \texttt{ff}\}$
- States: $\text{State} = \text{Var} \to \mathbb{N}$

A state $s \in \text{State}$ is a function that maps variables to their values. We use the notation:

- $s\ x$ to look up the value of variable x in state s
- $s[y \mapsto v]$ to denote state update, creating a new state that is like s except that variable y is mapped to value v

Formally, state update is defined as:

$$(s[y \mapsto v])\ x = \begin{cases} v & \text{if } x = y \\ s\ x & \text{if } x \neq y \end{cases}$$

1.2.4 Operational Semantics

Operational semantics focuses on modeling how programs execute step by step. There are two main approaches:

- Natural semantics (big-step semantics): Describes how the overall result of computation is obtained
- Structural operational semantics (small-step semantics): Describes individual steps of computation

Natural Semantics

Natural semantics describes the overall input-output relation of program execution. It is defined by a transition system $(, T, \rightarrow)$ where:

- $= \{(S, s) \mid S \in \text{While}, s \in \text{State}\} \cup \text{State}$
- $T = \text{State}$
- $\rightarrow \subseteq \{(S, s) \mid S \in \text{While}, s \in \text{State}\} \times \text{State}$

A typical transition has the form $(S, s) \rightarrow s'$ where S is the program, s is the initial state, and s' is the final state.

The transition relation is defined inductively with the following rules:

$$\frac{}{(x := a, s) \rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]}$$

$$\frac{}{(\text{skip}, s) \rightarrow s}$$

$$\frac{(S_1, s) \rightarrow s' \quad (S_2, s') \rightarrow s''}{(S_1; S_2, s) \rightarrow s''}$$

$$\frac{(S_1, s) \rightarrow s' \quad \mathcal{B}[\![b]\!]s = \texttt{tt}}{(\text{if } b \text{ then } S_1 \text{ else } S_2, s) \rightarrow s'}$$

$$\frac{(S_2, s) \rightarrow s' \quad \mathcal{B}[\![b]\!]s = \texttt{ff}}{(\text{if } b \text{ then } S_1 \text{ else } S_2, s) \rightarrow s'}$$

$$\frac{(S, s) \rightarrow s' \quad (\text{while } b \text{ do } S, s') \rightarrow s'' \quad \mathcal{B}[\![b]\!]s = \texttt{tt}}{(\text{while } b \text{ do } S, s) \rightarrow s''}$$

$$\frac{\mathcal{B}[\![b]\!]s = \texttt{ff}}{(\text{while } b \text{ do } S, s) \rightarrow s}$$

Structural Operational Semantics (SOS)

SOS describes how individual steps of computation take place. It is defined by a transition system $(, T, \Rightarrow)$ where:

- $= \{(S, s) \mid S \in \text{While}, s \in \text{State}\} \cup \text{State}$
- $T = \text{State}$
- $\Rightarrow \subseteq \{(S, s) \mid S \in \text{While}, s \in \text{State}\} \times$

The transition relation is defined with the following rules:

$$\frac{}{(x := a, s) \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]}$$

$$\frac{}{(\text{skip}, s) \Rightarrow s}$$

$$\frac{(S_1, s) \Rightarrow (S_1', s')}{(S_1; S_2, s) \Rightarrow (S_1'; S_2, s')}$$

$$\frac{(S_1, s) \Rightarrow s'}{(S_1; S_2, s) \Rightarrow (S_2, s')}$$

$$\frac{\mathcal{B}[\![b]\!]s = \texttt{tt}}{(\text{if } b \text{ then } S_1 \text{ else } S_2, s) \Rightarrow (S_1, s)}$$

$$\frac{\mathcal{B}[\![b]\!]s = \texttt{ff}}{(\text{if } b \text{ then } S_1 \text{ else } S_2, s) \Rightarrow (S_2, s)}$$

$$\frac{}{(\text{while } b \text{ do } S, s) \Rightarrow (\text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip}, s)}$$

Termination and Looping Programs

With operational semantics, we can formally define what it means for a program to terminate or loop:

- In natural semantics, a program S terminates on state s if there exists a derivation tree for $(S, s) \to s'$ for some s'. Otherwise, it loops or is stuck.
- In SOS, a program S terminates on state s if there exists a finite derivation sequence starting with (S, s) and ending in a state. It loops if there exists an infinite derivation sequence.

1.2.5 Denotational Semantics

Denotational semantics defines the meaning of programs directly as mathematical functions that map input states to output states.

Semantic Functions

The denotational semantics of While uses the following semantic functions:

- $\mathcal{A} : \text{Aexp} \to (\text{State} \to \mathbb{N})$ for arithmetic expressions

- $\mathcal{B} : \text{Bexp} \to (\text{State} \to \mathbb{T})$ for boolean expressions
- $\mathcal{S} : \text{Stm} \to (\text{State} \to \text{State})$ for statements

The semantic function for arithmetic expressions is defined compositionally:

$$\begin{aligned}
\mathcal{A}[\![n]\!]s &= \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![x]\!]s &= s\ x \\
\mathcal{A}[\![a_1 + a_2]\!]s &= \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 * a_2]\!]s &= \mathcal{A}[\![a_1]\!]s * \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 - a_2]\!]s &= \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s
\end{aligned}$$

Similarly, for boolean expressions:

$$\begin{aligned}
\mathcal{B}[\![\text{true}]\!]s &= \texttt{tt} \\
\mathcal{B}[\![\text{false}]\!]s &= \texttt{ff} \\
\mathcal{B}[\![a_1 = a_2]\!]s &= \begin{cases} \texttt{tt} & \text{if } \mathcal{A}[\![a_1]\!]s = \mathcal{A}[\![a_2]\!]s \\ \texttt{ff} & \text{if } \mathcal{A}[\![a_1]\!]s \neq \mathcal{A}[\![a_2]\!]s \end{cases} \\
\mathcal{B}[\![a_1 \leq a_2]\!]s &= \begin{cases} \texttt{tt} & \text{if } \mathcal{A}[\![a_1]\!]s \leq \mathcal{A}[\![a_2]\!]s \\ \texttt{ff} & \text{if } \mathcal{A}[\![a_1]\!]s > \mathcal{A}[\![a_2]\!]s \end{cases} \\
\mathcal{B}[\![\neg b]\!]s &= \begin{cases} \texttt{tt} & \text{if } \mathcal{B}[\![b]\!]s = \texttt{ff} \\ \texttt{ff} & \text{if } \mathcal{B}[\![b]\!]s = \texttt{tt} \end{cases} \\
\mathcal{B}[\![b_1 \wedge b_2]\!]s &= \begin{cases} \texttt{tt} & \text{if } \mathcal{B}[\![b_1]\!]s = \texttt{tt} \text{ and } \mathcal{B}[\![b_2]\!]s = \texttt{tt} \\ \texttt{ff} & \text{if } \mathcal{B}[\![b_1]\!]s = \texttt{ff} \text{ or } \mathcal{B}[\![b_2]\!]s = \texttt{ff} \end{cases}
\end{aligned}$$

Fixed Point Semantics for Loops

For iteration constructs like while loops, denotational semantics uses fixed point theory. For the while statement, we define:

$$\mathcal{S}[\![\text{while } b \text{ do } S]\!] = \text{fix}(F)$$

Where F is the functional:

$$F(f)(s) = \begin{cases} s & \text{if } \mathcal{B}[\![b]\!]s = \texttt{ff} \\ f(\mathcal{S}[\![S]\!]s) & \text{if } \mathcal{B}[\![b]\!]s = \texttt{tt} \end{cases}$$

And $\text{fix}(F)$ is the least fixed point of F.

1.2.6 Semantic Properties and Equivalence

Free Variables

The free variables of an expression are the variables that occur in it. Formally, for arithmetic expressions:

$$\begin{aligned} FV(n) &= \emptyset \\ FV(x) &= \{x\} \\ FV(a_1 + a_2) &= FV(a_1) \cup FV(a_2) \\ FV(a_1 * a_2) &= FV(a_1) \cup FV(a_2) \\ FV(a_1 - a_2) &= FV(a_1) \cup FV(a_2) \end{aligned}$$

A key property is that the value of an expression depends only on the values of its free variables:

Lemma 1. Let s and s' be two states satisfying that $s\ x = s'\ x$ for all $x \in FV(a)$. Then $\mathcal{A}[\![a]\!]s = \mathcal{A}[\![a]\!]s'$.

Substitutions

Substitution replaces occurrences of a variable with an expression. For arithmetic expressions:

$$\begin{aligned} n[y \mapsto a_0] &= n \\ x[y \mapsto a_0] &= \begin{cases} a_0 & \text{if } x = y \\ x & \text{if } x \neq y \end{cases} \\ (a_1 + a_2)[y \mapsto a_0] &= (a_1[y \mapsto a_0]) + (a_2[y \mapsto a_0]) \\ (a_1 * a_2)[y \mapsto a_0] &= (a_1[y \mapsto a_0]) * (a_2[y \mapsto a_0]) \\ (a_1 - a_2)[y \mapsto a_0] &= (a_1[y \mapsto a_0]) - (a_2[y \mapsto a_0]) \end{cases} \end{aligned}$$

An important property relates substitution to state update:

Lemma 2. $\mathcal{A}[\![a[y \mapsto a_0]]\!]s = \mathcal{A}[\![a]\!](s[y \mapsto \mathcal{A}[\![a_0]\!]s])$ for all states s.

Semantic Equivalence

Two statements S_1 and S_2 are semantically equivalent if they have the same input-output behavior. Formally:

- In natural semantics: S_1 and S_2 are equivalent if for all states s and s', $(S_1, s) \to s'$ if and only if $(S_2, s) \to s'$.

- In SOS: S_1 and S_2 are equivalent if for all states s, they either both terminate with the same final state or both loop.

Examples of equivalent statements:

- $S;\text{skip}$ and S
- while b do S and if b then $(S;\text{while } b \text{ do } S)$ else skip
- $S_1;(S_2;S_3)$ and $(S_1;S_2);S_3$

1.2.7 Extensions to the While Language

Several extensions to the While language can be considered to model additional programming features:

Abortion

Adding the abort statement allows modeling of program errors:

$$S ::= \ldots \mid \text{abort}$$

Non-determinism

Non-deterministic choice can be modeled with the or operator:

$$S ::= \ldots \mid S_1 \text{ or } S_2$$

With operational semantics rules:

$$\frac{(S_1, s) \to s'}{(S_1 \text{ or } S_2, s) \to s'}$$

$$\frac{(S_2, s) \to s'}{(S_1 \text{ or } S_2, s) \to s'}$$

$$\frac{}{(S_1 \text{ or } S_2, s) \Rightarrow (S_1, s)}$$

$$\frac{}{(S_1 \text{ or } S_2, s) \Rightarrow (S_2, s)}$$

Parallelism

Parallelism can be modeled with the par operator:

$$S ::= \ldots \mid S_1 \text{ par } S_2$$

With SOS rules:

$$\frac{(S_1, s) \Rightarrow (S_1', s')}{(S_1 \text{ par } S_2, s) \Rightarrow (S_1' \text{ par } S_2, s')}$$

$$\frac{(S_1, s) \Rightarrow s'}{(S_1 \text{ par } S_2, s) \Rightarrow (S_2, s')}$$

$$\frac{(S_2, s) \Rightarrow (S_2', s')}{(S_1 \text{ par } S_2, s) \Rightarrow (S_1 \text{ par } S_2', s')}$$

$$\frac{(S_2, s) \Rightarrow s'}{(S_1 \text{ par } S_2, s) \Rightarrow (S_1, s')}$$

1.3 Abstract Interpretation

Abstract interpretation provides a framework for approximating program semantics to make analysis tractable. It allows us to derive sound (though potentially incomplete) information about program behavior.

1.3.1 Introduction to Abstract Interpretation

Concrete program semantics is often too complex to compute exactly. Abstract interpretation approximates concrete semantics by mapping concrete values and operations to abstract domains that capture properties of interest.

The key insight is that we can work with abstract representations that preserve the properties we care about while discarding irrelevant details.

1.3.2 Galois Connections

The formal foundation of abstract interpretation is the Galois connection, which relates concrete and abstract domains.

Definition 1 (Galois Connection)**.** A Galois connection between two partially ordered sets $(C, \leq_C)$ and $(A, \leq_A)$ consists of two functions $\alpha : C \to A$ (abstraction) and $\gamma : A \to C$ (concretization) such that:

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

In this context:

- C is the concrete domain (e.g., sets of states)
- A is the abstract domain (e.g., interval constraints on variables)
- α abstracts concrete values
- γ gives concrete meaning to abstract values

1.3.3 Abstract Domains

Abstract domains represent properties of interest about program values. Common abstract domains include:

Sign Domain

The sign domain tracks whether values are positive, negative, or zero:

$$\text{Sign} = \{+, -, 0, \top, \bot\}$$

With ordering $\bot \leq \{+, -, 0\} \leq \top$, and abstraction function:

$$\alpha_{\text{Sign}}(n) = \begin{cases} + & \text{if } n > 0 \\ 0 & \text{if } n = 0 \\ - & \text{if } n < 0 \end{cases}$$

Interval Domain

The interval domain represents ranges of possible values:

$$\text{Interval} = \{[a, b] \mid a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}, a \leq b\} \cup \{\bot\}$$

With abstraction function:

$$\alpha_{\text{Interval}}(S) = \begin{cases} \bot & \text{if } S = \emptyset \\ [\min(S), \max(S)] & \text{otherwise} \end{cases}$$

Parity Domain

The parity domain tracks whether values are even or odd:

$$\text{Parity} = \{\text{even}, \text{odd}, \top, \bot\}$$

With ordering $\bot \leq \{\text{even}, \text{odd}\} \leq \top$, and abstraction function:

$$\alpha_{\text{Parity}}(n) = \begin{cases} \text{even} & \text{if } n \bmod 2 = 0 \\ \text{odd} & \text{if } n \bmod 2 = 1 \end{cases}$$

1.3.4 Abstract Semantics

Abstract semantics defines the meaning of program constructs in terms of abstract domains. It approximates concrete semantics while preserving soundness.

Abstract Operations

For the interval domain, abstract operations include:

$$\begin{aligned} [a,b]\hat{+}[c,d] &= [a+c, b+d] \\ [a,b]\hat{-}[c,d] &= [a-d, b-c] \\ [a,b]\hat{\times}[c,d] &= [\min(ac,ad,bc,bd), \max(ac,ad,bc,bd)] \end{aligned}$$

Abstract Transfer Functions

Abstract transfer functions model the effect of statements on abstract states:

$$\begin{aligned} \hat{\mathcal{S}}[\![x := a]\!](\hat{s}) &= \hat{s}[x \mapsto \hat{\mathcal{A}}[\![a]\!](\hat{s})] \\ \hat{\mathcal{S}}[\![\text{skip}]\!](\hat{s}) &= \hat{s} \\ \hat{\mathcal{S}}[\![S_1; S_2]\!](\hat{s}) &= \hat{\mathcal{S}}[\![S_2]\!](\hat{\mathcal{S}}[\![S_1]\!](\hat{s})) \\ \hat{\mathcal{S}}[\![\text{if } b \text{ then } S_1 \text{ else } S_2]\!](\hat{s}) &= \hat{\mathcal{S}}[\![S_1]\!](\hat{s} \sqcap \hat{\mathcal{B}}[\![b]\!]) \sqcup \hat{\mathcal{S}}[\![S_2]\!](\hat{s} \sqcap \hat{\mathcal{B}}[\![\neg b]\!]) \end{aligned}$$

For loops, we use a fixpoint computation with widening:

$$\hat{\mathcal{S}}[\![\text{while } b \text{ do } S]\!](\hat{s}) = \text{lfp}^{\nabla}_{\hat{s}}(\lambda X.(\hat{s} \sqcap \hat{\mathcal{B}}[\![\neg b]\!]) \sqcup (\hat{\mathcal{S}}[\![S]\!](X \sqcap \hat{\mathcal{B}}[\![b]\!])))$$

1.3.5 Widening and Narrowing

To ensure termination of fixpoint computations for abstract semantics, we use widening and narrowing operators.

Definition 2 (Widening Operator)**.** A widening operator $\nabla : A \times A \to A$ satisfies:

- $\forall x, y \in A : x \sqcup y \leq x \nabla y$ (over-approximation)

- For any ascending chain $x_0 \leq x_1 \leq x_2 \leq \ldots$, the sequence $y_0 = x_0, y_{i+1} = y_i \nabla x_{i+1}$ eventually stabilizes (termination)

For intervals, a common widening operator is:

$$[a,b]\nabla[c,d] = \begin{cases} [a,+\infty) & \text{if } d > b \\ (-\infty,b] & \text{if } c < a \\ [a,b] & \text{otherwise} \end{cases}$$

Definition 3 (Narrowing Operator)**.** A narrowing operator $\Delta : A \times A \to A$ helps refine results after widening:

- $\forall x, y \in A : y \leq x \implies y \leq x \Delta y \leq x$ (refinement)
- For any descending chain $x_0 \geq x_1 \geq x_2 \geq \ldots$, the sequence $y_0 = x_0, y_{i+1} = y_i \Delta x_{i+1}$ eventually stabilizes (termination)

1.3.6 Soundness and Completeness

A key property of abstract interpretation is soundness, which ensures that the abstract semantics correctly approximates concrete behavior.

Definition 4 (Soundness)**.** An abstract semantics $\hat{\mathcal{S}}$ is sound with respect to concrete semantics $\mathcal{S}$ if:

$$\forall S \in \text{Stm}, s \in \text{State}, \hat{s} \in \hat{\text{State}} : s \in \gamma(\hat{s}) \implies \mathcal{S}[\![S]\!](s) \in \gamma(\hat{\mathcal{S}}[\![S]\!](\hat{s}))$$

Abstract interpretation may sacrifice completeness (precision) for decidability and efficiency. The abstract results may be sound but conservative, reporting potential errors that cannot actually occur.

1.4 Dataflow Analysis

Dataflow analysis is a technique for computing information about the possible set of values at different program points. It's widely used in compilers for optimization and in static analyzers for detecting potential errors.

1.4.1 Control Flow Graphs

Dataflow analysis operates on control flow graphs (CFGs), which represent the flow of control in a program.

Definition 5 (Control Flow Graph)**.** A control flow graph $G = (N, E, n_0, n_f)$ consists of:

- A set of nodes N representing program points
- A set of edges $E \subseteq N \times N$ representing possible control flows
- An entry node $n_0 \in N$
- An exit node $n_f \in N$

1.4.2 Dataflow Equations

Dataflow analysis defines a set of equations that relate information at different program points. For each node n in the CFG:

$$\begin{aligned} \text{IN}[n] &= \sqcap_{p \in \text{pred}(n)} \text{OUT}[p] \\ \text{OUT}[n] &= f_n(\text{IN}[n]) \end{aligned}$$

Where:

- $\text{IN}[n]$ is the information at the entry to node n
- $\text{OUT}[n]$ is the information at the exit from node n
- $\text{pred}(n)$ is the set of predecessors of node n
- f_n is the transfer function for node n
- $\sqcap$ represents the meet operator (often set union or intersection)

1.4.3 Forward and Backward Analysis

Dataflow analyses can be categorized as forward or backward:

- **Forward Analysis**: Information flows in the direction of program execution. Used for reaching definitions, available expressions, etc.

$$\begin{aligned} \text{IN}[n] &= \sqcap_{p \in \text{pred}(n)} \text{OUT}[p] \\ \text{OUT}[n] &= f_n(\text{IN}[n]) \end{aligned}$$

- **Backward Analysis**: Information flows opposite to the direction of program execution. Used for live variables, etc.

$$\begin{aligned} \text{OUT}[n] &= \sqcap_{s \in \text{succ}(n)} \text{IN}[s] \\ \text{IN}[n] &= f_n(\text{OUT}[n]) \end{aligned}$$

1.4.4 May and Must Analysis

Dataflow analyses can also be categorized based on their certainty:

- **May Analysis**: Computes information that may be true along some execution path. Uses union ($\cup$) as the meet operator. Examples: reaching definitions, available expressions.
- **Must Analysis**: Computes information that must be true along all execution paths. Uses intersection ($\cap$) as the meet operator. Examples: very busy expressions, live variables.

1.4.5 Common Dataflow Analyses

Reaching Definitions

Determines which definitions (assignments) may reach a program point.

- Domain: 2^D where D is the set of all definition sites
- Direction: Forward
- Meet operator: Union ($\cup$)
- Transfer function: $f_n(x) = \text{gen}_n \cup (x - \text{kill}_n)$
  - gen_n: Definitions generated at node n
  - kill_n: Definitions killed at node n (by redefining variables)

Live Variables

Determines which variables may be used before their next definition.

- Domain: 2^V where V is the set of all variables
- Direction: Backward
- Meet operator: Union ($\cup$)
- Transfer function: $f_n(x) = \text{use}_n \cup (x - \text{def}_n)$
  - use_n: Variables used at node n
  - def_n: Variables defined at node n

Available Expressions

Determines which expressions have been computed and not invalidated.

- Domain: 2^E where E is the set of all expressions
- Direction: Forward
- Meet operator: Intersection ($\cap$)
- Transfer function: $f_n(x) = \text{gen}_n \cup (x - \text{kill}_n)$
  - gen_n: Expressions computed at node n
  - kill_n: Expressions invalidated at node n (by redefining variables)

1.4.6 Solving Dataflow Equations

Dataflow equations can be solved using iterative methods:

1. Initialize all IN[] and OUT[] sets based on boundary conditions
2. Repeatedly apply the transfer functions until a fixed point is reached

The iterative algorithm is guaranteed to terminate if:

- The domain has finite height or we use widening
- Transfer functions are monotonic

1.4.7 Monotone Frameworks

Most dataflow analyses can be formulated within a general monotone framework:

Definition 6 (Monotone Framework)**.** A monotone framework consists of:

- A semi-lattice $(L, \sqcap)$ with a top element $\top$
- A set of monotone transfer functions $F : L \to L$
- A control flow graph G
- Boundary conditions

The framework guarantees that the iterative solution method converges to the meet-over-all-paths (MOP) solution for distributive transfer functions.

1.5 Verification Tools

Software verification tools apply the theoretical concepts of semantics, abstract interpretation, and dataflow analysis to verify properties of real-world programs.

1.5.1 Categories of Verification Tools

Static Analyzers

Static analyzers examine program code without execution to identify potential issues:

- **Abstract Interpretation Tools**: Tools like Astrée, Polyspace, and InterProc analyze programs using abstract interpretation to verify absence of runtime errors.

- **Dataflow Analyzers**: Compilers like GCC and LLVM include dataflow analysis for optimization and some error detection.
- **Type-based Analyzers**: Tools that extend type systems to catch more errors at compile time.

Model Checkers

Model checkers verify that a system meets specified properties by exploring all possible states:

- **Explicit-state Model Checkers**: Tools like SPIN explore the concrete state space.
- **Symbolic Model Checkers**: Tools like NuSMV use symbolic representations of states (e.g., BDDs) to handle larger state spaces.
- **Bounded Model Checkers**: Tools like CBMC unroll loops to a bounded depth and convert the program to a satisfiability problem.

Theorem Provers

Theorem provers assist in developing formal proofs of program correctness:

- **Interactive Theorem Provers**: Tools like Coq, Isabelle, and PVS require user guidance to develop proofs.
- **Automatic Theorem Provers**: Tools like Z3, CVC4, and Vampire attempt to find proofs automatically.
- **Program Verifiers**: Tools like Frama-C, Why3, and Dafny combine programming languages with specification and verification capabilities.

1.5.2 Specific Verification Tools

Clousot

Clousot is a static analyzer developed by Microsoft Research that uses abstract interpretation to verify .NET programs:

- Focuses on verifying absence of null dereferences, array bounds violations, and arithmetic overflows
- Uses a combination of abstract domains, including numerical domains and heap analysis
- Integrated with Visual Studio and the Code Contracts framework

InterProc

InterProc is an abstract interpretation-based analyzer for a simple imperative language:

- Supports various abstract domains: intervals, octagons, polyhedra, etc.
- Provides a web interface for interactive analysis
- Developed by INRIA (French National Institute for Research in Computer Science)
- Educational tool that demonstrates the principles of abstract interpretation

Jandom

Jandom is a framework for static analysis based on abstract interpretation:

- Focuses on numerical properties of Java and bytecode programs
- Developed at the University of Pescara
- Implements various abstract domains and fixpoint algorithms
- Available as an API for integration with other tools

1.5.3 Practical Verification Example

Let's consider a simple program analysis using InterProc:

```
var x, y: int;
begin
  x := 0;
  y := 0;
  while x < 10 do
    x := x + 1;
    y := y + x;
  done
end
```

InterProc with the interval domain would analyze this program as follows:

1. Initialize: $x \in [-\infty, +\infty], y \in [-\infty, +\infty]$
2. After $x := 0$: $x \in [0, 0], y \in [-\infty, +\infty]$

3. After $y := 0$: $x \in [0, 0], y \in [0, 0]$
4. Loop entry first iteration: $x \in [0, 0], y \in [0, 0]$ with condition $x < 10$
5. After $x := x + 1$: $x \in [1, 1], y \in [0, 0]$
6. After $y := y + x$: $x \in [1, 1], y \in [1, 1]$
7. Loop entry second iteration: $x \in [1, 1], y \in [1, 1]$ with condition $x < 10$
8. After several iterations and applying widening: $x \in [0, 10], y \in [0, 55]$
9. After the loop (with $x \geq 10$): $x \in [10, 10], y \in [55, 55]$

This analysis proves that at the end of the program, $x = 10$ and $y = 55$ (which is the sum of integers from 1 to 10).

1.5.4 Challenges and Limitations

Verification tools face several challenges:

- **Scalability**: Analysis of large, complex programs can be computationally expensive.
- **Precision vs. Efficiency**: More precise analyses generally require more computational resources.
- **False Positives**: Static analyzers may report potential errors that cannot actually occur.
- **Undecidability**: Some properties are fundamentally undecidable, limiting what can be automatically verified.
- **Complex Language Features**: Features like dynamic dispatch, reflection, and concurrency complicate analysis.

1.6 Conclusion

Software verification represents a critical intersection of theoretical computer science and practical software engineering. By formalizing program semantics, applying abstract interpretation, and implementing dataflow analyses, we can develop tools that help ensure software reliability.

The field continues to evolve, with ongoing research addressing challenges in scalability, precision, and applicability to modern programming languages and paradigms. As software systems grow more complex and are deployed in increasingly critical contexts, the importance of rigorous verification approaches will only increase.

Chapter 2

Denotational Semantics

2.1 Introduction to Denotational Semantics

Denotational semantics is a mathematical approach to modeling the meaning of programs through mapping program phrases to mathematical objects that represent their behavior. Unlike operational semantics, which describes how programs execute, denotational semantics directly relates program constructs to their mathematical meaning, focusing on the input-output relationship rather than execution steps.

2.1.1 Key Principles of Denotational Semantics

Denotational semantics is characterized by several core principles:

- **Compositionality**: The meaning of a compound expression is determined by the meanings of its parts and the rules for combining them.
- **Mathematical functions**: Programs and their components are interpreted as mathematical functions.
- **Order theory and fixed points**: Programs with iteration or recursion are defined using fixed point theory.
- **Abstraction**: Details of actual execution are abstracted away, focusing on input-output behavior.

2.2 Direct Style Denotational Semantics for While

We define a denotational semantics for the While language as a mapping from programs to partial functions on states. This is often called "direct style" because it directly maps statements to their state transformation functions.

2.2.1 Basic Semantic Function

The primary semantic function is:

$$\mathcal{S}_{ds} : \text{Stm} \to (\text{State} \rightharpoonup \text{State}) \tag{2.1}$$

Here $\mathcal{S}_{ds}[S]$ is a partial function on states, meaning it may be undefined for some states (representing non-termination). The notation $\rightharpoonup$ indicates that this is a partial function.

2.2.2 Semantic Equations

The semantic function is defined inductively by the following equations:

$$\mathcal{S}_{ds}[x := a]s = s[x \mapsto \mathcal{A}[a]s] \tag{2.2}$$
$$\mathcal{S}_{ds}[\text{skip}] = \text{id} \tag{2.3}$$
$$\mathcal{S}_{ds}[S_1; S_2] = \mathcal{S}_{ds}[S_2] \circ \mathcal{S}_{ds}[S_1] \tag{2.4}$$
$$\mathcal{S}_{ds}[\text{if } b \text{ then } S_1 \text{ else } S_2] = \text{cond}(\mathcal{B}[b], \mathcal{S}_{ds}[S_1], \mathcal{S}_{ds}[S_2]) \tag{2.5}$$
$$\mathcal{S}_{ds}[\text{while } b \text{ do } S] = \text{FIX } F \tag{2.6}$$

where F is the functional defined by:

$$Fg = \text{cond}(\mathcal{B}[b], g \circ \mathcal{S}_{ds}[S], \text{id}) \tag{2.7}$$

2.2.3 Helper Notations

In these equations, we use the following notation:

$$\text{id } s = s \tag{2.8}$$
$$(f \circ g)s = \begin{cases} f(g\,s) & \text{if } g\,s \neq \text{undef and } f(g\,s) \neq \text{undef} \\ \text{undef} & \text{otherwise} \end{cases} \tag{2.9}$$
$$\text{cond}(p, g_1, g_2)s = \begin{cases} g_1\,s & \text{if } p\,s = \text{tt and } g_1\,s \neq \text{undef} \\ g_2\,s & \text{if } p\,s = \text{ff and } g_2\,s \neq \text{undef} \\ \text{undef} & \text{otherwise} \end{cases} \tag{2.10}$$

2.3 Fixed Points and While Loops

The meaning of while loops is given using fixed points. When we define:

$$\mathcal{S}_{ds}[\text{while } b \text{ do } S] = \text{FIX } F \tag{2.11}$$

The meaning of FIX F requires explanation.

2.3.1 The Fixed Point Problem

When analyzing a while loop:

$$\text{while } b \text{ do } S \tag{2.12}$$

We can rewrite it as:

$$\text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip} \tag{2.13}$$

This gives us a recursive equation:

$$\mathcal{S}_{ds}[\text{while } b \text{ do } S] = \mathcal{S}_{ds}[\text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip}] \tag{2.14}$$

This leads to:

$$\begin{aligned} \mathcal{S}_{ds}[\text{while } b \text{ do } S] &= \text{cond}(\mathcal{B}[b], \mathcal{S}_{ds}[S; \text{while } b \text{ do } S], \mathcal{S}_{ds}[\text{skip}]) && (2.15)\\ &= \text{cond}(\mathcal{B}[b], \mathcal{S}_{ds}[\text{while } b \text{ do } S] \circ \mathcal{S}_{ds}[S], \text{id}) && (2.16)\\ &= F(\mathcal{S}_{ds}[\text{while } b \text{ do } S]) && (2.17) \end{aligned}$$

where $Fg = \text{cond}(\mathcal{B}[b], g \circ \mathcal{S}_{ds}[S], \text{id})$.

This shows that $\mathcal{S}_{ds}[\text{while } b \text{ do } S]$ is a fixed point of F, meaning $F(\mathcal{S}_{ds}[\text{while } b \text{ do } S]) = \mathcal{S}_{ds}[\text{while } b \text{ do } S]$.

2.3.2 Requirements for FIX

The desired fixed point $\text{FIX}\, F$ should be some partial function $g_0 : \text{State} \rightharpoonup \text{State}$ satisfying:

- g_0 is a fixed point of F: $F\, g_0 = g_0$
- If g is another fixed point of F, then g is at least as defined as g_0: if $F\, g = g$ and $g_0\, s = s'$, then $g\, s = s'$ for all choices of s and s'

In other words, $\text{FIX}\, F$ should be the least fixed point of F.

2.4 Order Theory and Fixed Points

To formalize the concept of least fixed points, we need mathematical machinery from order theory.

2.4.1 Partially Ordered Sets

A set D with an ordering $\sqsubseteq$ is a partially ordered set (poset) if $\sqsubseteq$ is:

- **Reflexive**: $d \sqsubseteq d$
- **Transitive**: $d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_3$ imply $d_1 \sqsubseteq d_3$
- **Anti-symmetric**: $d_1 \sqsubseteq d_2$ and $d_2 \sqsubseteq d_1$ imply $d_1 = d_2$

An element d is a **least element** of $(D, \sqsubseteq)$ if $d \sqsubseteq d'$ for all $d' \in D$. If it exists, the least element is unique and denoted $\bot$ (bottom).

2.4.2 Ordering on Partial Functions

For partial functions, we define the ordering $\sqsubseteq$ as:

$$g_1 \sqsubseteq g_2 \text{ if and only if } \forall s, s' : \text{if } g_1\, s = s' \text{ then } g_2\, s = s' \tag{2.18}$$

This means g_1 is less defined than g_2 (or equally defined). The partial function defined by $\bot\, s = \text{undef}$ for all s is the least element in this ordering.

Lemma 3. (State $\rightharpoonup$ State, $\sqsubseteq$) is a partially ordered set with least element $\bot$.

2.4.3 Chains and Upper Bounds

A subset Y of D is called a **chain** if for any two elements d_1 and d_2 in Y, either $d_1 \sqsubseteq d_2$ or $d_2 \sqsubseteq d_1$.

An element d is an **upper bound** of Y if $d' \sqsubseteq d$ for all $d' \in Y$. It is a **least upper bound** of Y if it is an upper bound, and for any other upper bound d' of Y, we have $d \sqsubseteq d'$. The least upper bound of Y, if it exists, is denoted $\bigsqcup Y$.

2.4.4 Chain Complete Partial Orders

A partially ordered set $(D, \sqsubseteq)$ is a **chain complete partial order** (ccpo) if every chain in D has a least upper bound.

Lemma 4. (State $\rightharpoonup$ State, $\sqsubseteq$) is a chain complete partially ordered set. For a chain Y, the least upper bound $\bigsqcup Y$ is given by:

$$(\bigsqcup Y)s = \begin{cases} g\, s & \text{if } g\, s \neq \text{undef for some } g \in Y \\ \text{undef} & \text{otherwise} \end{cases} \tag{2.19}$$

2.5 Continuous Functions and Fixed Point Theorem

2.5.1 Monotone and Continuous Functions

A function $f : D \to D'$ between ccpos is **monotone** if whenever $d_1 \sqsubseteq d_2$, we have $f\, d_1 \sqsubseteq' f\, d_2$.

A function $f : D \to D'$ is **continuous** if:

- f is monotone
- $\bigsqcup'\{f\, d \mid d \in Y\} = f(\bigsqcup Y)$ for all non-empty chains Y of D

2.5.2 The Fixed Point Theorem

Theorem 1 (Fixed Point Theorem)**.** Let $f : D \to D$ be a continuous function on a ccpo $(D, \sqsubseteq)$ with least element $\bot$. Then:

$$\text{FIX } f = \bigsqcup \{f^n \bot \mid n \geq 0\} \tag{2.20}$$

defines an element of D, and this element is the least fixed point of f.

Here, we use the notation $f^0 = \text{id}$ and $f^{n+1} = f \circ f^n$ for $n \geq 0$.

2.5.3 Auxiliary Lemmas for Continuity

For the semantics of While, we need to ensure that the functionals used are continuous. Key lemmas include:

Lemma 5. Let $g_0 : \text{State} \rightharpoonup \text{State}$, $p : \text{State} \to \mathbb{T}$ and define $F\, g = \text{cond}(p, g, g_0)$. Then F is continuous.

Lemma 6. Let $g_0 : \text{State} \rightharpoonup \text{State}$ and define $F\, g = g \circ g_0$. Then F is continuous.

Lemma 7. If $f : D \to D'$ and $f' : D' \to D''$ are continuous functions, then $f' \circ f$ is a continuous function.

2.6 Well-definedness of Denotational Semantics

These theoretical tools allow us to formally establish the well-definedness of our denotational semantics.

The semantic equations for $\mathcal{S}_{ds}$ define a total function in $\text{Stm} \to (\text{State} \rightharpoonup \text{State})$.

The proof relies on showing that all functionals used in the semantic equations are continuous, allowing us to apply the Fixed Point Theorem to define the semantics of while loops.

2.7 Equivalence of Semantic Approaches

One fundamental result is the equivalence between operational and denotational semantics for the While language.

Theorem 2. For every statement S of While, we have $\mathcal{S}_{sos}[S] = \mathcal{S}_{ds}[S]$, where:

$$\mathcal{S}_{sos}[S]s = \begin{cases} s' & \text{if } (S, s) \Rightarrow^* s' \\ \text{undefined} & \text{otherwise} \end{cases} \tag{2.21}$$

The proof involves showing both directions of the inclusion:

Lemma 8. For every statement S of While, we have $\mathcal{S}_{sos}[S] \sqsubseteq \mathcal{S}_{ds}[S]$.

Lemma 9. For every statement S of While, we have $\mathcal{S}_{ds}[S] \sqsubseteq \mathcal{S}_{sos}[S]$.

2.8 Program Equivalence in Denotational Semantics

Denotational semantics provides a powerful framework for establishing program equivalence. Two programs P and Q are equivalent, denoted $P \cong Q$, when $\mathcal{S}_{ds}[P] = \mathcal{S}_{ds}[Q]$.

Examples of equivalent programs include:

- $S; \text{skip} \cong S$
- $S_1; (S_2; S_3) \cong (S_1; S_2); S_3$
- $\text{while } b \text{ do } S \cong \text{if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip}$

More complex equivalences, like $\text{repeat } S \text{ until } b \cong S; \text{while } \neg b \text{ do } S$, require careful reasoning using fixed point theory.

2.9 Extensions and Applications

2.9.1 Denotational Semantics of Additional Constructs

The approach can be extended to additional language constructs:

Repeat-Until Loops

$$\text{repeat } S \text{ until } b \cong S; \text{if } b \text{ then skip else repeat } S \text{ until } b \tag{2.22}$$

This leads to the equation:

$$\mathcal{S}_{ds}[\text{repeat } S \text{ until } b] = \text{cond}(\mathcal{B}[b], \text{id}, \mathcal{S}_{ds}[\text{repeat } S \text{ until } b]) \circ \mathcal{S}_{ds}[S] \tag{2.23}$$

For Loops

$$\text{for } x := a_1 \text{ to } a_2 \text{ do } S \cong x := a_1; \text{while } x \leq a_2 \text{ do } (S; x := x + 1) \tag{2.24}$$

2.9.2 Example: Factorial Program

Consider the factorial program:

$$y := 1; \text{while } \neg(x = 1) \text{ do } (y := y * x; x := x - 1) \tag{2.25}$$

Its denotational semantics is:

$$\mathcal{S}_{ds}[y := 1; \text{while } \neg(x = 1) \text{ do } (y := y * x; x := x - 1)]s = (\text{FIX } F)(s[y \mapsto 1]) \tag{2.26}$$

where:

$$(F\,g)\,s = \begin{cases} g(s[y \mapsto (s\,y * s\,x)][x \mapsto (s\,x) - 1]) & \text{if } s\,x \neq 1 \\ s & \text{if } s\,x = 1 \end{cases} \tag{2.27}$$

Using the Fixed Point Theorem, we can show that:

$$(\text{FIX } F)s = \begin{cases} \text{undef} & \text{if } s\,x < 1 \\ s[y \mapsto (s\,y) * n * \ldots * 2 * 1][x \mapsto 1] & \text{if } s\,x = n \text{ and } n \geq 1 \end{cases} \tag{2.28}$$

2.10 Conclusion and Further Readings

Denotational semantics offers a powerful mathematical framework for understanding program meaning and proving program properties. Its focus on mathematical functions and fixed point theory provides a rigorous foundation for program analysis and verification.

The topics covered in this chapter form the basis for more advanced topics such as abstract interpretation, dataflow analysis, and the development of software verification tools, which we will explore in subsequent chapters.

Chapter 3

Static Program Analysis

3.1 Introduction to Static Program Analysis

Static program analysis comprises automatic techniques designed to infer approximate information about run-time program behaviors at compile-time without executing the program. Unlike dynamic analysis, which examines programs during execution, static analysis works by examining the source code or other artifacts derived from it.

3.1.1 Definition and Applications

Static program analysis can be formally defined as a set of techniques that compute conservative approximations of possible program behaviors. These techniques reason about the program state at various program points without executing the program itself.

The applications of static program analysis include:

- **Bug detection**: Identifying potential runtime errors such as null pointer dereferences, buffer overflows, and memory leaks.
- **Program verification**: Proving that a program satisfies certain properties or specifications.
- **Code optimization**: Identifying redundancies, dead code, and optimization opportunities.
- **Program parallelization**: Determining which parts of a program can be safely executed in parallel.
- **Type inference**: Deducing types of expressions in programming languages with implicit typing.

3.2 Program Representation

To perform static analysis, programs are typically represented in forms amenable to systematic examination, such as program points, control flow graphs, and abstract syntax trees.

3.2.1 Program Points

Program points are locations in a program where the program state may change. They provide reference points for analysis to track information flow through the program.

Example 1. Consider the following program that computes the double factorial of a non-negative number n:

```
{n = k  n  0}
input n;
m := 2;
while n > 1 do
    m := m * n;
    n := n - 1;
output m;
{m = 2(k!)}
```

The same program with program points identified:

```
(1) input n;
(2) m := 2;
while (3) n > 1 do
    (4) m := m * n;
    (5) n := n - 1;
(6) output m;
```

These numbered locations allow us to reason about program states at specific points in the execution.

3.2.2 Control Flow Graph

A control flow graph (CFG) is a directed graph representation of a program where:

- Nodes represent program points or basic blocks (sequences of instructions without branching).
- Edges represent possible control flow transitions between program points.

Formally, a control flow graph G is a tuple (N, E, n_0) where:

- N is a set of nodes representing program points.
- $E \subseteq N \times N$ is a set of directed edges representing control flow.
- $n_0 \in N$ is the entry node.

Example 2. For the program above, the control flow graph would be:

```
input n      (1)
   ↓
m := 2       (2)
   ↓
  n > 1      (3)  → No →  output m  (6)
   ↓ Yes
m := m * n   (4)
   ↓
n := n - 1   (5)
   ↑
```

3.3 Static Analysis Principles

3.3.1 Parity Analysis Example

To illustrate the concepts of static analysis, let us consider a simple parity analysis of the factorial program. The goal is to determine whether variables have even or odd values at different program points.

Example 3. For the factorial program, our parity domain has the values:

- Even: the value is even
- Odd: the value is odd
- DontKnow: the value can be either even or odd

The analysis would propagate these values through the control flow graph:

```
(1) n: DontKnow, m: DontKnow
(2) n: DontKnow, m: DontKnow
(3) n: DontKnow, m: Even     (since m := 2)
(4) n: DontKnow, m: Even
(5) n: DontKnow, m: Even
(6) n: DontKnow, m: Even
```

From this analysis, we can infer that at program point (6), the value of variable m will be even for any input value of n. This is because m is initialized to 2 (an even number) and multiplying an even number by any integer always yields an even number.

Note that for the exact factorial program (without the multiplication by 2), the analysis would provide no useful information on the parity of m at point (6), since factorial values can be either even or odd depending on the input.

3.3.2 Precision and Correctness

A fundamental aspect of static analysis is the trade-off between precision and decidability.

Undecidability and Approximation

Due to Rice's Theorem, most interesting dynamic properties of programs are undecidable. This means that we cannot automatically infer them with complete precision. As a result, static analyses must make approximations.

Theorem 3 (Rice's Theorem)**.** Any non-trivial semantic property of programs is undecidable.

Soundness

Despite approximations, we need to ensure the correctness (soundness) of static analysis:

- If the analysis outputs a positive result (YES), then the property is certainly verified.
- If the analysis does not output a positive result, then it may happen that the property is not verified (DontKnow).

This leads to the concept of sound versus unsound analyses:

- **Sound analysis**: Never produces false negatives (may have false positives).
- **Unsound analysis**: May produce false negatives, which are particularly dangerous for verification.

Example 4. Consider the halting problem, which is undecidable. A sound termination analysis would only claim a program terminates when it can prove it. In contrast, an unsound analysis might incorrectly classify some non-terminating programs as terminating.

Example 5. Imagine a static analysis that attempts to detect divisions by zero by searching for string patterns "/0" in the code. This would generate:

- **True alarm**: $x = 100/0;$
- **False alarm (false positive)**: $print(\text{"}25/09/2016\text{"});$
- **False alarm (false positive)**: $y = 100/02;$
- **False negative (missed error)**: $z = 0; print(5/z);$

The presence of false negatives makes this analysis unsound and unsuitable for verification purposes.

3.4 Static Analysis and Compiler Optimization

One important application of static analysis is in optimizing compilers, which transform program code to improve efficiency without changing the program's input/output behavior.

3.4.1 Optimization Techniques

Common optimization techniques enabled by static analysis include:

- **Common subexpression elimination**: If an expression is computed multiple times, compute it once and reuse the result.
- **Dead code elimination**: Remove code that has no effect on the program's output.
- **Constant folding**: Evaluate constant expressions at compile time.
- **Loop invariant code motion**: Move operations outside of loops when their results don't change within the loop.

3.4.2 Limitations of Optimization

A "fully optimizing compiler" that produces the smallest equivalent program is theoretically impossible due to undecidability constraints. If such a compiler existed, it could solve the halting problem—an undecidable problem—by determining if a program has the same behavior as an infinite loop.

3.5 Designing Static Analyses

To design a sound static analysis, we need:

1. A well-defined reference program semantics

2. A precise specification of what the static analysis computes and how
3. A proof of correctness with respect to the semantics
4. An efficient implementation

3.5.1 Static Analysis Techniques

Several techniques provide frameworks for static analysis:

- **Abstract Interpretation**: A theoretical framework for approximating program semantics.
- **Dataflow Analysis**: A technique for gathering information about the possible values at different points in a program.
- **Model Checking**: Verification of whether a model meets a given specification.
- **Logical Deductive Systems**: Program logics and SMT/SAT solvers.
- **Type Systems**: Formal systems that assign types to expressions to prevent errors.

3.6 Mathematical Foundations: Lattices and Fixpoints

Static analysis techniques, particularly abstract interpretation and dataflow analysis, rely heavily on concepts from order theory such as lattices and fixpoints.

3.6.1 Lattices

Definition 7 (Partially Ordered Set)**.** A partially ordered set (poset) is a pair $(L, \leq)$ where L is a set and $\leq$ is a relation on L that is:

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

Definition 8 (Least Upper Bound and Greatest Lower Bound)**.** For a subset $Y \subseteq L$ of a poset $(L, \leq)$:

- An element $u \in L$ is an upper bound of Y if $\forall y \in Y, y \leq u$.
- The least upper bound (lub) of Y, denoted $\bigsqcup Y$, is an upper bound u of Y such that $\forall u' \in L$ where u' is an upper bound of Y, $u \leq u'$.

- Similarly, a greatest lower bound (glb) of Y, denoted $\bigsqcap Y$, is defined.

Definition 9 (Lattice)**.** A lattice is a poset $(L, \leq)$ such that for any pair of elements $x, y \in L$, both $\bigsqcup\{x, y\}$ (typically written as $x \sqcup y$) and $\bigsqcap\{x, y\}$ (typically written as $x \sqcap y$) exist.

Example 6. The powerset of a set with the subset relation, $(\mathcal{P}(S), \subseteq)$, forms a lattice where:

- The lub is the union: $A \sqcup B = A \cup B$
- The glb is the intersection: $A \sqcap B = A \cap B$

3.6.2 Complete Lattices and CPOs

Definition 10 (Complete Lattice)**.** A lattice $(L, \leq)$ is complete if every subset of L (including the empty set) has both a lub and a glb.

Definition 11 (Chain)**.** A subset $Y \subseteq L$ of a poset $(L, \leq)$ is a chain if $\forall y_1, y_2 \in Y, y_1 \leq y_2 \vee y_2 \leq y_1$ (i.e., Y is totally ordered).

Definition 12 (CPO)**.** A poset $(L, \leq)$ is a chain-complete partial order (CPO) if every chain in L has a lub.

Definition 13 (Ascending Chain)**.** A sequence $(y_n)_{n \in \mathbb{N}}$ of elements in L is an ascending chain if $n \leq m \Rightarrow y_n \leq y_m$.

Definition 14 (Ascending Chain Condition)**.** A poset $(L, \leq)$ satisfies the Ascending Chain Condition (ACC) if every ascending chain in L eventually stabilizes (i.e., becomes constant).

3.6.3 Fixpoints

Definition 15 (Fixpoint)**.** For a function $f : L \to L$ on a poset $(L, \leq)$, a fixpoint of f is an element $x \in L$ such that $f(x) = x$. The set of all fixpoints of f is denoted $\text{Fix}(f) = \{x \in L \mid f(x) = x\}$.

Definition 16 (Least Fixpoint)**.** The least fixpoint of f, denoted $\text{lfp}(f)$, if it exists, is the least element in $\text{Fix}(f)$ with respect to $\leq$.

Definition 17 (Pre-Fixpoint and Post-Fixpoint)**.** For a function $f : L \to L$:

- A pre-fixpoint of f is an element $x \in L$ such that $f(x) \leq x$.
- A post-fixpoint of f is an element $x \in L$ such that $x \leq f(x)$.

Theorem 4 (Fixpoint Theorem for Complete Lattices)**.** Let $f : L \to L$ be a monotonic function on a complete lattice $(L, \leq)$. Then:

- $\text{lfp}(f) = \bigsqcap\{x \in L \mid f(x) \leq x\}$ (the glb of all pre-fixpoints)

- $\text{gfp}(f) = \bigsqcup\{x \in L \mid x \leq f(x)\}$ (the lub of all post-fixpoints)

Theorem 5 (Kleene Fixpoint Theorem)**.** Let $f : L \to L$ be a continuous function on a CPO $(L, \leq)$ with a least element $\bot$. Then the least fixpoint of f exists and is given by:

$$\text{lfp}(f) = \bigsqcup\{f^n(\bot) \mid n \geq 0\}$$

where $f^0(x) = x$ and $f^{n+1}(x) = f(f^n(x))$ for $n \geq 0$.

3.7 Abstract Interpretation

Abstract interpretation is a theoretical framework for approximating program semantics. It was formalized by Patrick and Radhia Cousot in 1977 and has since become a fundamental approach in static program analysis.

3.7.1 Basic Concepts

Abstract interpretation involves:

- A concrete domain representing the actual program behaviors
- An abstract domain representing the properties of interest
- Mappings between the concrete and abstract domains
- Abstract operations that approximate concrete operations

3.7.2 Abstraction and Concretization

The relationship between concrete and abstract domains is formalized through:

Definition 18 (Abstraction and Concretization Functions)**.** Given a concrete domain C and an abstract domain A:

- The abstraction function $\alpha : C \to A$ maps concrete values to their abstract representations.
- The concretization function $\gamma : A \to C$ maps abstract values to the set of concrete values they represent.

Definition 19 (Galois Connection)**.** A Galois connection between posets $(C, \leq_C)$ and $(A, \leq_A)$ is a pair of functions $\alpha : C \to A$ and $\gamma : A \to C$ such that:

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

Definition 20 (Galois Insertion)**.** A Galois connection (α, C, A, γ) is a Galois insertion if α is surjective (or equivalently, γ is injective), which means that:

$$\forall a \in A : \alpha(\gamma(a)) = a$$

3.7.3 Sign Domain Example

The sign domain is a simple abstract domain that tracks whether values are positive, negative, or zero.

Example 7. Consider the sign domain for integer values:

- Abstract domain: Sign $= \{+, -, 0, \top, \bot\}$
- Ordering: $\bot \leq \{+, -, 0\} \leq \top$
- Abstraction function:

$$\alpha_{\text{Sign}}(n) = \begin{cases} + & \text{if } n > 0 \\ 0 & \text{if } n = 0 \\ - & \text{if } n < 0 \end{cases}$$

- Concretization function:

$\gamma_{\text{Sign}}(\bot) = \emptyset$, $\gamma_{\text{Sign}}(+) = \{n \in \mathbb{Z} \mid n > 0\}$, $\gamma_{\text{Sign}}(0) = \{0\}$, $\gamma_{\text{Sign}}(-) = \{n \in \mathbb{Z} \mid n < 0\}$, $\gamma_{\text{Sign}}(\top)$

Multiplication in the sign domain is defined as:

$\times_a$	$-$	0	$+$	$\top$	$\bot$
$-$	$+$	0	$-$	$\top$	$\bot$
0	0	0	0	0	$\bot$
$+$	$-$	0	$+$	$\top$	$\bot$
$\top$	$\top$	0	$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

This example illustrates how concrete operations (integer multiplication) can be approximated by abstract operations (sign multiplication).

3.7.4 Correctness of Abstract Operations

Definition 21 (Correct Abstract Operation)**.** Let $\text{op} : C^n \to C$ be a concrete operation and $\text{op}_a : A^n \to A$ be a corresponding abstract operation. The abstract operation op_a is correct with respect to op if:

$$\forall a_1, \ldots, a_n \in A : \text{op}(\gamma(a_1), \ldots, \gamma(a_n)) \subseteq \gamma(\text{op}_a(a_1, \ldots, a_n))$$

Definition 22 (Best Correct Approximation)**.** For any concrete operation $\text{op} : C^n \to C$, the best correct approximation op_A of op on abstract domain A is:

$$\text{op}_A(a_1, \ldots, a_n) = \alpha(\text{op}(\gamma(a_1), \ldots, \gamma(a_n)))$$

3.8 Abstract Denotational Semantics

We can apply abstract interpretation to denotational semantics to create an abstract denotational semantics for program analysis.

3.8.1 Collecting Semantics

The collecting semantics lifts the standard denotational semantics to operate on sets of states, representing state properties.

Definition 23 (Collecting Denotational Semantics)**.** For a language with arithmetic expressions (Aexp), boolean expressions (Bexp), and statements (While), the collecting semantics are:

$$\begin{aligned} A_c &: \text{Aexp} \to \mathcal{P}(\text{State}) \to \mathcal{P}(\mathbb{Z}) \\ A_c[\![e]\!]T &= \{A[\![e]\!]s \mid s \in T\} \\ B_c &: \text{Bexp} \to \mathcal{P}(\text{State}) \to \mathcal{P}(\text{State}) \\ B_c[\![b]\!]T &= \{s \in T \mid B[\![b]\!]s = \text{tt}\} \\ D &: \text{While} \to \mathcal{P}(\text{State}) \to \mathcal{P}(\text{State}) \end{aligned}$$

For statements, D is defined as:

$$\begin{aligned} D[\![x := e]\!]T &= \{s[x \mapsto A[\![e]\!]s] \mid s \in T\} \\ D[\![\text{skip}]\!]T &= T \\ D[\![S_1; S_2]\!]T &= (D[\![S_2]\!] \circ D[\![S_1]\!])T \\ D[\![\text{if } b \text{ then } S_1 \text{ else } S_2]\!]T &= (D[\![S_1]\!] \circ B_c[\![b]\!])T \cup (D[\![S_2]\!] \circ B_c[\![\neg b]\!])T \end{aligned}$$

For while loops, we use a fixpoint:

$$D[\![\text{while } b \text{ do } S]\!]T = B_c[\![\neg b]\!](\text{lfp}(\lambda T'.T \cup (D[\![S]\!] \circ B_c[\![b]\!])T'))$$

3.8.2 Abstract Domains

To perform abstract interpretation, we need abstract domains that represent properties of concrete values and states.

Definition 24 (Abstract Domains for Denotational Semantics)**.** Given a Galois connection $(\alpha_A, \mathcal{P}(\mathbb{Z}), A, \gamma_A)$ for values and $(\alpha_S, \mathcal{P}(\text{State}), S, \gamma_S)$ for states, we define abstract semantics:

$$\begin{aligned} A^\# &: \text{Aexp} \to S \to A \\ B^\# &: \text{Bexp} \to S \to S \\ D^\# &: \text{While} \to S \to S \end{aligned}$$

3.8.3 Correctness of Abstract Semantics

Theorem 6 (Soundness of Abstract Semantics)**.** The abstract semantics is sound if for all programs $S \in \text{While}$ and all abstract states $s^\# \in S$:

$$D[\![S]\!](\gamma_S(s^\#)) \subseteq \gamma_S(D^\#[\![S]\!]s^\#)$$

3.8.4 Sign Domain Example

Example 8. For the sign domain, abstract arithmetic expressions are defined as:

$$\begin{aligned} A^\#[\![n]\!]s^\# &= \alpha(\{n\}) \\ A^\#[\![x]\!]s^\# &= s^\#(x) \\ A^\#[\![e_1 \text{ op } e_2]\!]s^\# &= A^\#[\![e_1]\!]s^\# \ \text{op}_{\text{Sign}} A^\#[\![e_2]\!]s^\# \end{aligned}$$

For boolean expressions, the abstract semantics handles operations like equality, inequality, and comparison by determining when abstract values definitely satisfy or definitely don't satisfy the condition.

For statements, the abstract semantics follows the structure of the concrete collecting semantics, but operates on abstract states.

3.9 Interval Analysis

Interval analysis is a classic abstract interpretation technique that approximates numeric values using intervals.

3.9.1 Interval Domain

Definition 25 (Interval Domain)**.** The interval domain Int consists of:

- Elements of the form $[a, b]$ where $a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}$ and $a \leq b$
- A special element $\bot$ representing the empty set
- Ordering: $[a, b] \leq [c, d]$ iff $c \leq a$ and $b \leq d$
- Abstraction function: $\alpha(S) = [\min(S), \max(S)]$ if $S \neq \emptyset$, and $\alpha(\emptyset) = \bot$
- Concretization function: $\gamma([a, b]) = \{n \in \mathbb{Z} \mid a \leq n \leq b\}$ and $\gamma(\bot) = \emptyset$

3.9.2 Widening for Interval Analysis

A key challenge in interval analysis is handling loops, which can lead to unbounded growth of intervals. Widening operators address this problem.

Definition 26 (Widening Operator)**.** A widening operator $\nabla : L \times L \to L$ on a complete lattice $(L, \leq_L)$ satisfies:

- $\forall x, y \in L : x \sqcup y \leq_L x \nabla y$ (over-approximation)
- For any ascending chain $(x_n)_{n \geq 0}$, the sequence $y_0 = x_0, y_{i+1} = y_i \nabla x_{i+1}$ eventually stabilizes (termination)

A standard widening operator for intervals is:

$$[a, b] \nabla [c, d] = \begin{cases} [a, +\infty) & \text{if } d > b \\ (-\infty, b] & \text{if } c < a \\ [a, b] & \text{otherwise} \end{cases}$$

Example 9. Consider the program:

```
int x = 1;
while (x <= 100) do
    x := x + 1;
```

Without widening, interval analysis would require 101 iterations to reach the fixpoint $[1, 101]$. With widening, we might compute:

$$\begin{aligned} X_0 &= \bot \\ X_1 &= X_0 \nabla F_a(X_0) = \bot \nabla [1, 1] = [1, 1] \\ X_2 &= X_1 \nabla F_a(X_1) = [1, 1] \nabla [1, 2] = [1, +\infty) \end{aligned}$$

After applying the loop condition, we conclude $x \in [101, +\infty)$ after the loop, correctly indicating that $x > 100$.

3.9.3 Practical Applications of Abstract Interpretation

Abstract interpretation has been successfully applied in various tools:

- **Astrée**: An abstract interpretation-based static analyzer that proved the absence of runtime errors in the Airbus A380 fly-by-wire system.
- **Frama-C**: A platform for analyzing C programs using abstract interpretation.
- **Clang Static Analyzer**: Uses abstract interpretation techniques to find bugs in C, C++, and Objective-C programs.
- **IKOS**: An open-source framework for static analysis based on abstract interpretation.

3.10 Conclusion

Static program analysis provides powerful techniques to reason about program behaviors without execution. Through abstract interpretation and related approaches, we can develop sound analyses that identify potential issues, verify properties, and guide optimizations. While the fundamental limitations of undecidability mean that these analyses must make approximations, carefully designed abstract domains and operations allow us to extract valuable information about programs at compile time.

The fields of abstract interpretation and static analysis continue to evolve, with ongoing research addressing challenges in scalability, precision, and applicability to modern programming languages and paradigms.

Chapter 4

Abstract Interpretation with Control Flow Graphs

4.1 Introduction

Abstract interpretation is a theoretical framework for constructing sound approximations of program semantics. It provides a formal basis for static analysis, allowing us to derive information about program behavior without executing the program itself. This chapter focuses on abstract interpretation techniques applied to control flow graphs, specifically for numerical programs.

The key insight of abstract interpretation is that we can work with abstract representations of program states that preserve the properties we care about while discarding irrelevant details. This allows us to make analysis tractable, even when the concrete semantics is undecidable or computationally infeasible to analyze fully.

4.2 Generalities and Notations

4.2.1 Syntax

We begin by defining a simple toy language that we'll use throughout this chapter:

- Fixed, finite set of variables $\mathcal{V}$
- One datatype: scalars in $\mathbb{I}$, with $\mathbb{I} \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$ (and later, floating-point numbers $\mathbb{F}$)
- No procedures

Expression Syntax

Arithmetic expressions in our language are defined by the following grammar:

$$
\begin{aligned}
exp ::= V &\quad \text{variable } V \in \mathcal{V} && (4.1)\\
| -exp &\quad \text{negation} && (4.2)\\
| exp \odot exp &\quad \text{binary operation: } \odot \in \{+, -, \times, /\} && (4.3)\\
| [c, c'] &\quad \text{constant range, } c, c' \in \mathbb{I} \cup \{\pm\infty\} && (4.4)
\end{aligned}
$$

Note that c is a shorthand for $[c, c]$.

Programs as Structured Syntax

Programs can be represented as syntax trees:

$$
\begin{aligned}
prog ::= V := exp &\quad \text{assignment} && (4.5)\\
| \textit{if } exp \bowtie 0 \textit{ then } prog \textit{ else } prog \textit{ fi} &\quad \text{test} && (4.6)\\
| \textit{while } exp \bowtie 0 \textit{ do } prog \textit{ done} &\quad \text{loop} && (4.7)\\
| prog; prog &\quad \text{sequence} && (4.8)\\
| \epsilon &\quad \text{no-op} && (4.9)
\end{aligned}
$$

Where comparison operators: $\bowtie \in \{=, <, >, \leq, \geq, \neq\}$.

Programs as Control Flow Graphs

Alternatively, programs can be represented as control flow graphs. The commands in this representation are:

$$
\begin{aligned}
com ::= V := exp &\quad \text{assignment into } V \in \mathcal{V} && (4.10)\\
| exp \bowtie 0 &\quad \text{test, } \bowtie \in \{=, <, >, \leq, \geq, \neq\} && (4.11)
\end{aligned}
$$

Programs as control flow graphs are formally defined as:

$$\mathcal{P} \stackrel{def}{=} (\mathcal{L}, e, x, \mathcal{A}) \tag{4.12}$$

Where:

- $\mathcal{L}$ are program points (labels)
- e is the entry point: $e \in \mathcal{L}$
- x is the exit point: $x \in \mathcal{L}$
- $\mathcal{A}$ are arcs: $\mathcal{A} \subseteq \mathcal{L} \times com \times \mathcal{L}$

Example

Consider the following program:

```
X:=[0,10];
Y:=100;
while X>=0 do
    X:=X-1;
    Y:=Y+10
done
```

This program can be represented as a control flow graph:

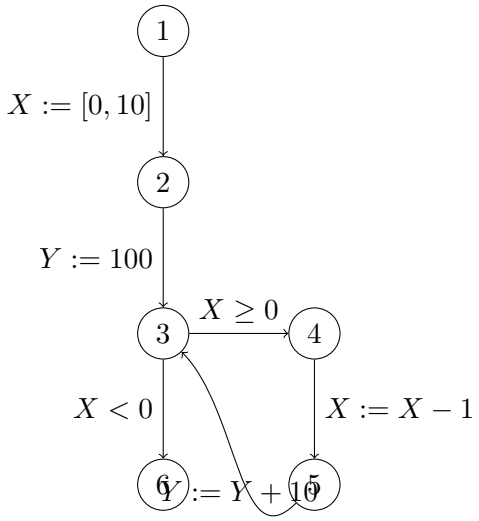


Figure 4.1: Control flow graph example

4.3 Concrete Semantics

4.3.1 Forward Concrete Semantics

The semantics of expressions $\mathcal{E}[\![e]\!] : (\mathcal{V} \to \mathbb{I}) \to \mathcal{P}(\mathbb{I})$ gives a set of values when evaluating an expression e in an environment ρ:

$$\mathcal{E}[\![[c,c']]\!]\rho \stackrel{def}{=} \{x \in \mathbb{I} \mid c \leq x \leq c'\} \tag{4.13}$$

$$\mathcal{E}[\![V]\!]\rho \stackrel{def}{=} \{\rho(V)\} \tag{4.14}$$

$$\mathcal{E}[\![-e]\!]\rho \stackrel{def}{=} \{-v \mid v \in \mathcal{E}[\![e]\!]\rho\} \tag{4.15}$$

$$\mathcal{E}[\![e_1 + e_2]\!]\rho \stackrel{def}{=} \{v_1 + v_2 \mid v_1 \in \mathcal{E}[\![e_1]\!]\rho, v_2 \in \mathcal{E}[\![e_2]\!]\rho\} \tag{4.16}$$

$$\mathcal{E}[\![e_1 - e_2]\!]\rho \stackrel{def}{=} \{v_1 - v_2 \mid v_1 \in \mathcal{E}[\![e_1]\!]\rho, v_2 \in \mathcal{E}[\![e_2]\!]\rho\} \tag{4.17}$$

$$\mathcal{E}[\![e_1 \times e_2]\!]\rho \stackrel{def}{=} \{v_1 \times v_2 \mid v_1 \in \mathcal{E}[\![e_1]\!]\rho, v_2 \in \mathcal{E}[\![e_2]\!]\rho\} \tag{4.18}$$

$$\mathcal{E}[\![e_1/e_2]\!]\rho \stackrel{def}{=} \{v_1/v_2 \mid v_1 \in \mathcal{E}[\![e_1]\!]\rho, v_2 \in \mathcal{E}[\![e_2]\!]\rho, v_2 \neq 0\} \tag{4.19}$$

Note that $\mathcal{E}[\![e_i]\!] = \emptyset \Rightarrow \mathcal{E}[\![e_1 \odot e_2]\!] = \emptyset$, where the empty set models run-time errors or non-termination.

The semantics of commands $\mathcal{C}[\![c]\!] : \mathcal{P}(\mathcal{V} \to \mathbb{I}) \to \mathcal{P}(\mathcal{V} \to \mathbb{I})$ defines a transfer function for commands:

$$\mathcal{C}[\![V := e]\!]X \stackrel{def}{=} \{\rho[V \mapsto v] \mid \rho \in X, v \in \mathcal{E}[\![e]\!]\rho\} \tag{4.20}$$

$$\mathcal{C}[\![e \bowtie 0]\!]X \stackrel{def}{=} \{\rho \mid \rho \in X, \exists v \in \mathcal{E}[\![e]\!]\rho, v \bowtie 0\} \tag{4.21}$$

This is a complete join morphism: $\mathcal{C}[\![c]\!]X = \bigcup_{\rho \in X} \mathcal{C}[\![c]\!]\{\rho\}$.

Note that:

$$\mathcal{C}[\![x := 1/0]\!]X = \emptyset \tag{4.22}$$

$$\mathcal{C}[\![1/0 < 1]\!]X = \emptyset \tag{4.23}$$

where $\emptyset$ models run-time error or non-termination. Also,

$$\mathcal{C}[\![[2,2] > 0]\!]X = X \tag{4.24}$$

The semantics of programs $\mathcal{P}[\![(\mathcal{L}, e, x, \mathcal{A})]\!] : \mathcal{L} \to \mathcal{P}(\mathcal{V} \to \mathbb{I})$ computes the most precise invariant at each program point $\ell \in \mathcal{L}$.

This is the smallest solution of a recursive equation system $(X_\ell)_{\ell \in \mathcal{L}}$:

$$X_e = \text{(given initial state)} \tag{4.25}$$

$$X_{\ell \neq e} = \bigcup_{(\ell', c, \ell) \in \mathcal{A}} \mathcal{C}[\![c]\!]X_{\ell'} \text{ (transfer function)} \tag{4.26}$$

By Tarski's Theorem, this smallest solution exists and is unique. The domain $\mathcal{D} \stackrel{def}{=} (\mathcal{P}(\mathcal{V} \to \mathbb{I}), \subseteq, \cup, \cap, \emptyset, (\mathcal{V} \to \mathbb{I}))$ is a complete lattice, and each $M_\ell : X_\ell \mapsto \bigcup_{(\ell', c, \ell) \in \mathcal{A}} \mathcal{C}[\![c]\!]X_{\ell'}$ is monotonic in $\mathcal{D}$.

Therefore, the solution is the least fixpoint of $(M_\ell)_{\ell \in \mathcal{L}}$.

4.3.2 Resolution

Resolution by increasing iterations:

$$\begin{cases} X_e^0 \stackrel{def}{=} X_e \\ X_{\ell \neq e}^0 \stackrel{def}{=} \emptyset \end{cases} \tag{4.27}$$

$$\begin{cases} X_e^{n+1} \stackrel{def}{=} X_e \\ X_{\ell \neq e}^{n+1} \stackrel{def}{=} \bigcup_{(\ell', c, \ell) \in \mathcal{A}} \mathcal{C}[\![c]\!] X_{\ell'}^n \end{cases} \tag{4.28}$$

This converges in ω iterations to a least solution because each $\mathcal{C}[\![c]\!]$ is continuous in the CPO $\mathcal{D}$ (by the Kleene-Knaster-Tarski theorem).

4.3.3 Resolution Example

For our example program, we get the following equation system:

$$\begin{cases} X_1 = (\{\mathrm{X}, \mathrm{Y}\} \to \mathbb{Z}) \\ X_2 = \mathcal{C}[\![\mathrm{X} := [0, 10]]\!] X_1 \\ X_3 = \mathcal{C}[\![\mathrm{Y} := 100]\!] X_2 \cup \mathcal{C}[\![\mathrm{Y} := \mathrm{Y} + 10]\!] X_5 \\ X_4 = \mathcal{C}[\![\mathrm{X} \geq 0]\!] X_3 \\ X_5 = \mathcal{C}[\![\mathrm{X} := \mathrm{X} - 1]\!] X_4 \\ X_6 = \mathcal{C}[\![\mathrm{X} < 0]\!] X_3 \end{cases} \tag{4.29}$$

After iteration, the loop invariant is:

$$X_3 = \{\rho \mid \rho(\mathrm{X}) \in [0, 10], 10\rho(\mathrm{X}) + \rho(\mathrm{Y}) \in [100, 200] \cap 10\mathbb{Z}\} \tag{4.30}$$

4.3.4 Limit to Automation

We would like to perform automatic numerical invariant discovery, but face theoretical and practical problems:

- Elements of $\mathcal{P}(\mathcal{V} \to \mathbb{I})$ are not computer-representable
- Transfer functions $\mathcal{C}[\![c]\!]$, $\overline{\mathcal{C}}[\![c]\!]$ are not computable
- Lattice iterations in $\mathcal{P}(\mathcal{V} \to \mathbb{I})$ are transfinite
- Finding the best invariant is an undecidable problem

Note that even when $\mathbb{I}$ is finite, a concrete analysis is not tractable:

- Representing elements in $\mathcal{P}(\mathcal{V} \to \mathbb{I})$ in extension is expensive
- Computing $\mathcal{C}[\![c]\!]$, $\overline{\mathcal{C}}[\![c]\!]$ explicitly is expensive
- The lattice $\mathcal{P}(\mathcal{V} \to \mathbb{I})$ has a large height (many iterations)

4.4 Abstraction

4.4.1 Numerical Abstract Domains

A numerical abstract domain is given by:

- A subset of $\mathcal{P}(\mathcal{V} \to \mathbb{I})$ (a set of environment sets)
- Together with a machine encoding
- Effective and sound abstract operators
- An iteration strategy ensuring convergence in finite time

Numerical Abstract Domain Examples

Abstract domains can be classified by their expressiveness:

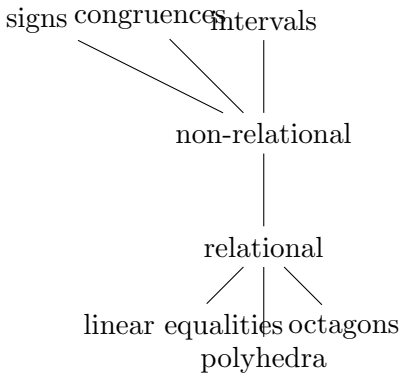


Figure 4.2: Hierarchy of numerical abstract domains

Representation

An abstract domain is given by:

- A set $\mathcal{D}^\sharp$ of machine-representable abstract values
- A partial order $(\mathcal{D}^\sharp, \sqsubseteq, \bot^\sharp, \top^\sharp)$ relating the amount of information given by abstract values
- A concretization function $\gamma : \mathcal{D}^\sharp \to \mathcal{P}(\mathcal{V} \to \mathbb{I})$ giving a concrete meaning to each abstract element

Required algebraic properties:

- γ should be monotonic for $\sqsubseteq$: $X^\sharp \sqsubseteq Y^\sharp \Rightarrow \gamma(X^\sharp) \subseteq \gamma(Y^\sharp)$
- $\gamma(\bot^\sharp) = \emptyset$

- $\gamma(\top^\sharp) = \mathcal{V} \to \mathbb{I}$

Note: γ need not be one-to-one.

Abstract Operators

We require:

- Sound, effective, abstract transfer functions $\mathcal{C}^\sharp[\![c]\!]$, $\overline{\mathcal{C}}^\sharp[\![c]\!]$ for all commands c
- Sound, effective, abstract set operators $\sqcup^\sharp$, $\sqcap^\sharp$
- An algorithm to decide the ordering $\sqsubseteq$

Soundness criterion: $F^\sharp$ is a sound abstraction of a n-ary operator F if:

$$\forall X_1^\sharp, \ldots, X_n^\sharp \in \mathcal{D}^\sharp, F(\gamma(X_1^\sharp), \ldots, \gamma(X_n^\sharp)) \subseteq \gamma(F^\sharp(X_1^\sharp, \ldots, X_n^\sharp)) \tag{4.31}$$

This concerns both semantic and algorithmic aspects.

4.4.2 Abstract Semantics

The abstract semantic equation system is:

$$X^\sharp : \mathcal{L} \to \mathcal{D}^\sharp \tag{4.32}$$

$$X_\ell^\sharp \supseteq \begin{cases} X_e^\sharp & \text{if } \ell = e \text{ (where } X_e \subseteq \gamma(X_e^\sharp)) \\ \bigsqcup^\sharp_{(\ell',c,\ell)\in\mathcal{A}} \mathcal{C}^\sharp[\![c]\!] X_{\ell'}^\sharp & \text{if } \ell \neq e \text{ (abstract transfer function)} \end{cases} \tag{4.33}$$

Soundness Theorem: Any solution $(X_\ell^\sharp)_{\ell\in\mathcal{L}}$ is a sound over-approximation of the concrete collecting semantics:

$$\forall \ell \in \mathcal{L}, \gamma(X_\ell^\sharp) \supseteq X_\ell \tag{4.34}$$

where X_ℓ is the smallest solution of

$$\begin{cases} X_e & \text{given} \\ X_\ell = \bigcup_{(\ell',c,\ell)\in\mathcal{A}} \mathcal{C}[\![c]\!] X_{\ell'} & \text{if } \ell \neq e \end{cases} \tag{4.35}$$

4.4.3 Iteration Strategy

To effectively solve the abstract system, we require:

- An iteration ordering on abstract equations (which equation(s) are applied at a given iteration)
- A widening operator ∇ to speed up the convergence, if there are infinite strictly increasing chains in $\mathcal{D}^\sharp$

$\nabla : (\mathcal{D}^\sharp \times \mathcal{D}^\sharp) \to \mathcal{D}^\sharp$ is a widening if:

- It is sound: $\gamma(X^\sharp) \cup \gamma(Y^\sharp) \subseteq \gamma(X^\sharp \nabla Y^\sharp)$
- It enforces termination: $\forall$ sequence $(Y_i^\sharp)_{i\in\mathbb{N}}$ the sequence $X_0^\sharp = Y_0^\sharp$, $X_{i+1}^\sharp = X_i^\sharp \nabla Y_{i+1}^\sharp$ stabilizes in finite time: $\exists n < \omega, X_{n+1}^\sharp = X_n^\sharp$ (note: $\exists n, \forall m \leq n, X_{m+1}^\sharp = X_m^\sharp$ is not required)

4.4.4 Abstract Analysis

$W \subseteq \mathcal{L}$ is a set of widening points if every CFG cycle has a point in W.

Forward analysis:

$$X_e^{\sharp 0} \stackrel{def}{=} X_e^\sharp \text{ given, such that } X_e \subseteq \gamma(X_e^\sharp) \tag{4.36}$$

$$X_{\ell\neq e}^{\sharp 0} \stackrel{def}{=} \bot^\sharp \tag{4.37}$$

$$X_\ell^{\sharp n+1} \stackrel{def}{=} \begin{cases} X_e^\sharp & \text{if } \ell = e \\ \bigsqcup^\sharp_{(\ell',c,\ell)\in\mathcal{A}} \mathcal{C}^\sharp[\![c]\!] X_{\ell'}^{\sharp n} & \text{if } \ell \notin W, \ell \neq e \\ X_\ell^{\sharp n} \nabla \bigsqcup^\sharp_{(\ell',c,\ell)\in\mathcal{A}} \mathcal{C}^\sharp[\![c]\!] X_{\ell'}^{\sharp n} & \text{if } \ell \in W, \ell \neq e \end{cases} \tag{4.38}$$

Termination: for some ω, $\forall \ell, X_\ell^{\sharp\omega+1} = X_\ell^{\sharp\omega}$

Soundness: $\forall \ell \in \mathcal{L}, X_\ell \subseteq \gamma(X_\ell^{\sharp\omega})$

This can be refined by decreasing iterations with narrowing $\triangle$ (presented later).

Here, we apply every equation at each step, but other iteration schemes are possible (worklist, chaotic iterations).

4.4.5 Exact and Best Abstractions

Galois connection: $(D, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftarrow\!\!\rightarrow}} (D^\sharp, \sqsubseteq)$

α, γ monotonic and $\forall X, Y^\sharp, \alpha(X) \sqsubseteq Y^\sharp \Leftrightarrow X \subseteq \gamma(Y^\sharp)$

This implies elements X have a best abstraction: $\alpha(X)$, and operators F have a best abstraction: $F^\sharp = \alpha \circ F \circ \gamma$.

Sometimes, no α exists:

- $\{\gamma(Y^\sharp) \mid X \subseteq \gamma(Y^\sharp)\}$ has no greatest lower bound
- Abstract elements with the same γ have no best representation
- $\alpha \circ F \circ \gamma$ may still be defined for some F (partial α)

Concretization-based optimality:

- Sound abstraction: $\gamma \circ F^\sharp \supseteq F \circ \gamma$
- Exact abstraction: $\gamma \circ F^\sharp = F \circ \gamma$
- Optimal abstraction: $\gamma(X^\sharp)$ minimal in $\{\gamma(Y^\sharp) \mid X \subseteq \gamma(Y^\sharp)\}$

4.5 Non-Relational Domains

Non-relational domains "forget" all relationships between variables. They cannot distinguish between sets with the same variable projections.

4.5.1 Value Abstraction

The idea is to start from an abstraction of values $\mathcal{P}(\mathbb{I})$.

A numerical value abstract domain consists of:

- $\mathcal{B}^\sharp$ abstract values, machine-representable
- $\gamma_b : \mathcal{B}^\sharp \to \mathcal{P}(\mathbb{I})$ concretization
- $\sqsubseteq_b$ partial order
- $\bot_b^\sharp$, $\top_b^\sharp$ represent $\emptyset$ and $\mathbb{I}$
- $\sqcup_b^\sharp$, $\sqcap_b^\sharp$ abstractions of $\cup$ and $\cap$
- ∇_b extrapolation operator (widening)
- $\alpha_b : \mathcal{P}(\mathbb{I}) \to \mathcal{B}^\sharp$ abstraction (optional)

Derived Abstract Domain

We define $\mathcal{D}^\sharp \stackrel{def}{=} (\mathcal{V} \to (\mathcal{B}^\sharp \setminus \{\bot_b^\sharp\})) \cup \{\bot^\sharp\}$

This is a point-wise extension: $X^\sharp \in \mathcal{D}^\sharp$ is a vector of elements in $\mathcal{B}^\sharp$ (e.g., using arrays of size $|\mathcal{V}|$) with a smashed $\bot^\sharp$ (to avoid redundant representations of $\emptyset$).

Definitions on $\mathcal{D}^\sharp$ derived from $\mathcal{B}^\sharp$:

$$\gamma(X^\sharp) \stackrel{def}{=} \begin{cases} \emptyset & \text{if } X^\sharp = \bot^\sharp \\ \{\rho \mid \forall V, \rho(V) \in \gamma_b(X^\sharp(V))\} & \text{otherwise} \end{cases} \tag{4.39}$$

$$\alpha(X) \stackrel{def}{=} \begin{cases} \bot^\sharp & \text{if } X = \emptyset \\ V.\alpha_b(\{\rho(V) \mid \rho \in X\}) & \text{otherwise} \end{cases} \tag{4.40}$$

$$\top^\sharp \stackrel{def}{=} V.\top_b^\sharp \tag{4.41}$$

$$X^\sharp \sqsubseteq Y^\sharp \stackrel{def}{\Leftrightarrow} X^\sharp = \bot^\sharp \vee (X^\sharp, Y^\sharp \neq \bot^\sharp \wedge \forall V, X^\sharp(V) \sqsubseteq_b Y^\sharp(V)) \tag{4.42}$$

$$X^\sharp \sqcup^\sharp Y^\sharp \stackrel{def}{=} \begin{cases} Y^\sharp & \text{if } X^\sharp = \bot^\sharp \\ X^\sharp & \text{if } Y^\sharp = \bot^\sharp \\ V.X^\sharp(V) \sqcup_b^\sharp Y^\sharp(V) & \text{otherwise} \end{cases} \tag{4.43}$$

$$X^\sharp \nabla Y^\sharp \stackrel{def}{=} \begin{cases} Y^\sharp & \text{if } X^\sharp = \bot^\sharp \\ X^\sharp & \text{if } Y^\sharp = \bot^\sharp \\ V.X^\sharp(V) \nabla_b Y^\sharp(V) & \text{otherwise} \end{cases} \tag{4.44}$$

$$X^\sharp \sqcap^\sharp Y^\sharp \stackrel{def}{=} \begin{cases} \bot^\sharp & \text{if } X^\sharp = \bot^\sharp \text{ or } Y^\sharp = \bot^\sharp \\ \bot^\sharp & \text{if } \exists V, X^\sharp(V) \sqcap_b^\sharp Y^\sharp(V) = \bot_b^\sharp \\ V.X^\sharp(V) \sqcap_b^\sharp Y^\sharp(V) & \text{otherwise} \end{cases} \tag{4.45}$$

Cartesian Abstraction

Non-relational domains "forget" all relationships between variables. This can be formalized with the Cartesian abstraction:

Upper closure operator $\rho_c : \mathcal{P}(\mathcal{V} \to \mathbb{I}) \to \mathcal{P}(\mathcal{V} \to \mathbb{I})$:

$$\rho_c(X) \stackrel{def}{=} \{\rho \in \mathcal{V} \to \mathbb{I} \mid \forall V \in \mathcal{V}, \exists \rho' \in X, \rho(V) = \rho'(V)\} \tag{4.46}$$

A domain is non-relational if $\rho \circ \gamma = \gamma$, i.e., it cannot distinguish between X and X' if $\rho_c(X) = \rho_c(X')$.

Example: $\rho_c(\{(X, Y) \mid X \in \{0, 2\}, Y \in \{0, 2\}, X + Y \leq 2\}) = \{0, 2\} \times \{0, 2\}$

Generic Non-Relational Abstract Assignments

Given sound abstract versions in $\mathcal{B}^\sharp$ of all arithmetic operators:

$$[c, c']_b^\sharp : \{x \mid c \leq x \leq c'\} \subseteq \gamma_b([c, c']_b^\sharp) \tag{4.47}$$
$$-_b^\sharp : \{-x \mid x \in \gamma_b(X_b^\sharp)\} \subseteq \gamma_b(-_b^\sharp X_b^\sharp) \tag{4.48}$$
$$+_b^\sharp : \{x + y \mid x \in \gamma_b(X_b^\sharp), y \in \gamma_b(Y_b^\sharp)\} \subseteq \gamma_b(X_b^\sharp +_b^\sharp Y_b^\sharp) \tag{4.49}$$
$$\ldots \tag{4.50}$$

We can define an abstract semantics of expressions $\mathcal{E}^\sharp[\![e]\!] : \mathcal{D}^\sharp \to \mathcal{B}^\sharp$:

$$\mathcal{E}^\sharp[\![e]\!]\bot^\sharp \stackrel{def}{=} \bot_b^\sharp \tag{4.51}$$

If $X^\sharp \neq \bot^\sharp$:

$$\mathcal{E}^\sharp[\![[c, c']]\!]X^\sharp \stackrel{def}{=} [c, c']_b^\sharp \tag{4.52}$$
$$\mathcal{E}^\sharp[\![V]\!]X^\sharp \stackrel{def}{=} X^\sharp(V) \tag{4.53}$$
$$\mathcal{E}^\sharp[\![-e]\!]X^\sharp \stackrel{def}{=} -_b^\sharp \mathcal{E}^\sharp[\![e]\!]X^\sharp \tag{4.54}$$
$$\mathcal{E}^\sharp[\![e_1 + e_2]\!]X^\sharp \stackrel{def}{=} \mathcal{E}^\sharp[\![e_1]\!]X^\sharp +_b^\sharp \mathcal{E}^\sharp[\![e_2]\!]X^\sharp \tag{4.55}$$
$$\ldots \tag{4.56}$$

We can then define an abstract assignment:

$$\mathcal{C}^\sharp[\![V := e]\!]X^\sharp \stackrel{def}{=} \begin{cases} \bot^\sharp & \text{if } V_b^\sharp = \bot_b^\sharp \\ X^\sharp[V \mapsto V_b^\sharp] & \text{otherwise} \end{cases} \tag{4.57}$$

where $V_b^\sharp = \mathcal{E}^\sharp[\![e]\!]X^\sharp$

Using a Galois connection (α_b, γ_b), we can define best abstract arithmetic operators:

$$[c, c']_b^\sharp \stackrel{def}{=} \alpha_b(\{x \mid c \leq x \leq c'\}) \tag{4.58}$$
$$-_b^\sharp X_b^\sharp \stackrel{def}{=} \alpha_b(\{-x \mid x \in \gamma(X_b^\sharp)\}) \tag{4.59}$$
$$X_b^\sharp +_b^\sharp Y_b^\sharp \stackrel{def}{=} \alpha_b(\{x + y \mid x \in \gamma(X_b^\sharp), y \in \gamma(Y_b^\sharp)\}) \tag{4.60}$$
$$\ldots \tag{4.61}$$

Note: in general, $\mathcal{E}^\sharp[\![e]\!]$ is less precise than $\alpha_b \circ \mathcal{E}[\![e]\!] \circ \gamma$. For example, with $e = V - V$ and $\gamma_b(X^\sharp(V)) = [0, 1]$, we have $\mathcal{E}^\sharp[\![V - V]\!]X^\sharp = [-1, 1]$ while $\mathcal{E}[\![V - V]\!]_{\gamma_b(X^\sharp(V))} = [0, 0]$.

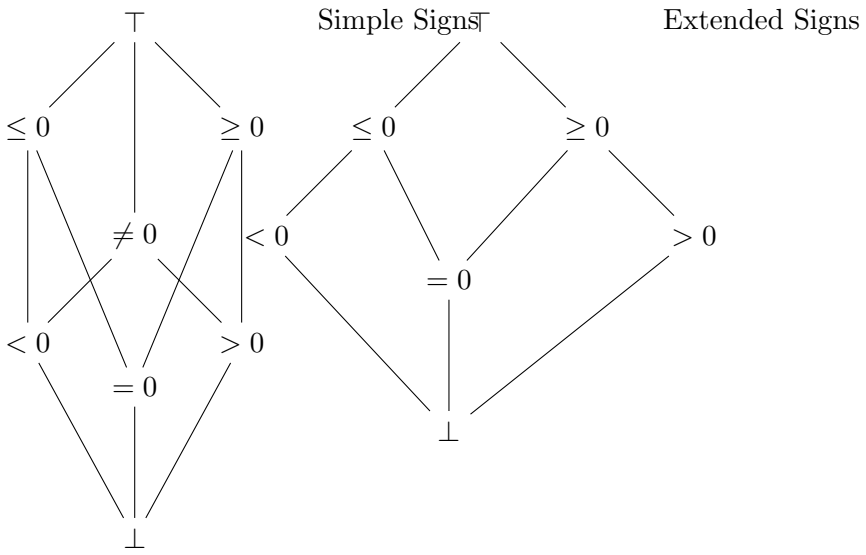


Figure 4.3: Sign lattices

4.5.2 The Sign Domain

The sign domain tracks whether values are positive, negative, or zero. There are two variants:

The extended sign domain is a refinement of the simple sign domain. The diagram implicitly defines $\sqcup^\sharp$ and $\sqcap^\sharp$ as the least upper bound and greatest lower bound for $\sqsubseteq$.

Operations on Simple Signs

Abstraction α forms a Galois connection between $\mathcal{B}^\sharp$ and $\mathcal{P}(\mathbb{I})$:

$$\alpha_b(S) \stackrel{def}{=} \begin{cases} \bot_b^\sharp & \text{if } S = \emptyset \\ = 0 & \text{if } S = \{0\} \\ \leq 0 & \text{else if } \forall s \in S, s \leq 0 \\ \geq 0 & \text{else if } \forall s \in S, s \geq 0 \\ \top_b^\sharp & \text{otherwise} \end{cases} \tag{4.62}$$

Derived abstract arithmetic operators:

$$c_b^\sharp \stackrel{def}{=} \alpha_b(\{c\}) = \begin{cases} = 0 & \text{if } c = 0 \\ < 0 & \text{if } c < 0 \\ > 0 & \text{if } c > 0 \end{cases} \tag{4.63}$$

$$X^\sharp +_b^\sharp Y^\sharp \stackrel{def}{=} \alpha_b(\{x+y \mid x \in \gamma_b(X^\sharp), y \in \gamma_b(Y^\sharp)\}) = \begin{cases} \bot_b^\sharp & \text{if } X^\sharp \text{ or } Y^\sharp = \bot_b^\sharp \\ = 0 & \text{if } X^\sharp = Y^\sharp == 0 \\ \leq 0 & \text{else if } X^\sharp \text{ and } Y^\sharp \in \{= 0, \leq 0\} \\ \geq 0 & \text{else if } X^\sharp \text{ and } Y^\sharp \in \{= 0, \geq 0\} \\ \top_b^\sharp & \text{otherwise} \end{cases} \tag{4.64}$$

Abstract Test Examples

$$\mathcal{C}^\sharp[\![X = 0]\!]X^\sharp \stackrel{def}{=} \begin{cases} X^\sharp[X \mapsto = 0] & \text{if } X^\sharp(X) \in \{= 0, \leq 0, \geq 0, \top_b^\sharp\} \\ \bot^\sharp & \text{otherwise} \end{cases} \tag{4.65}$$

$$\mathcal{C}^\sharp[\![X - c = 0]\!]X^\sharp \stackrel{def}{=} \begin{cases} \mathcal{C}^\sharp[\![X = 0]\!]X^\sharp & \text{if } c = 0 \\ X^\sharp & \text{otherwise} \end{cases} \tag{4.66}$$

$$\mathcal{C}^\sharp[\![X - Y = 0]\!]X^\sharp \stackrel{def}{=} \begin{cases} \mathcal{C}^\sharp[\![X = 0]\!]X^\sharp & \text{if } X^\sharp(Y) \in \{= 0, \leq 0, \geq 0\} \\ X^\sharp & \text{otherwise} \end{cases} \tag{4.67}$$

$$\sqcap^\sharp \begin{cases} \mathcal{C}^\sharp[\![Y = 0]\!]X^\sharp & \text{if } X^\sharp(X) \in \{= 0, \leq 0, \geq 0\} \\ X^\sharp & \text{otherwise} \end{cases} \tag{4.68}$$

Other cases: $\mathcal{C}^\sharp[\![\text{expr} \bowtie 0]\!]X^\sharp \stackrel{def}{=} X^\sharp$ is always a sound abstraction.

Simple Sign Analysis Example

Example analysis using the simple sign domain:

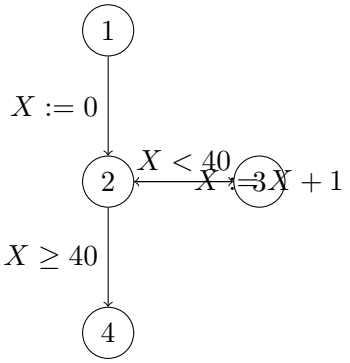
```
X:=0;
while X<40 do
    X:=X+1
done
```

Program rewritten as a control flow graph:
With the sign domain, the iterations will produce the following:
After several iterations, we derive that at program point 2, $X \geq 0$.

4.5.3 The Constant Domain

The constant domain tracks when variables have specific constant values.
The lattice is flat but infinite:

$$\mathcal{B}^\sharp = \mathbb{I} \cup \{\top_b^\sharp, \bot_b^\sharp\} \tag{4.69}$$



ℓ	$X_\ell^{\sharp 0}$	$X_\ell^{\sharp 1}$	$X_\ell^{\sharp 2}$	$X_\ell^{\sharp 3}$	$X_\ell^{\sharp 4}$
1	$\top$	$\top$	$\top$	$\top$	$\top$
2	$\bot$	$X = 0$	$X = 0$	$X \geq 0$	$X \geq 0$
3	$\bot$	$\bot$	$X = 0$	$X = 0$	$X \geq 0$
4	$\bot$	$\bot$	$X = 0$	$X = 0$	$X \geq 0$

Table 4.1: Sign domain iterations

Operations on Constants

Abstraction α forms a Galois connection:

$$\alpha_b(S) \stackrel{def}{=} \begin{cases} \bot_b^\sharp & \text{if } S = \emptyset \\ c & \text{if } S = \{c\} \\ \top_b^\sharp & \text{otherwise} \end{cases} \tag{4.70}$$

Derived abstract arithmetic operators:

$$c_b^\sharp \stackrel{def}{=} c \tag{4.71}$$

$$\gamma(X^\sharp) +_b^\sharp \gamma(Y^\sharp) \stackrel{def}{=} \begin{cases} \bot_b^\sharp & \text{if } X^\sharp \text{ or } Y^\sharp = \bot_b^\sharp \\ \top_b^\sharp & \text{else if } X^\sharp \text{ or } Y^\sharp = \top_b^\sharp \\ X^\sharp + Y^\sharp & \text{otherwise} \end{cases} \tag{4.72}$$

$$\gamma(X^\sharp) \times_b^\sharp \gamma(Y^\sharp) \stackrel{def}{=} \begin{cases} \bot_b^\sharp & \text{if } X^\sharp \text{ or } Y^\sharp = \bot_b^\sharp \\ 0 & \text{else if } X^\sharp \text{ or } Y^\sharp = 0 \\ \top_b^\sharp & \text{else if } X^\sharp \text{ or } Y^\sharp = \top_b^\sharp \\ X^\sharp \times Y^\sharp & \text{otherwise} \end{cases} \tag{4.73}$$

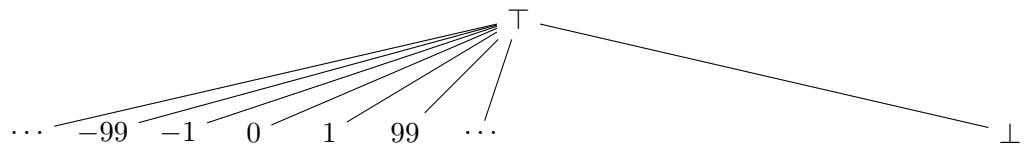


Figure 4.4: Constant domain lattice

Abstract Test Examples

$$\mathcal{C}^\sharp[\![X - c = 0]\!]X^\sharp \stackrel{def}{=} \begin{cases} \bot^\sharp & \text{if } X^\sharp(X) \notin \{c, \top_b^\sharp\} \\ X^\sharp[X \mapsto c] & \text{otherwise} \end{cases} \tag{4.74}$$

$$\mathcal{C}^\sharp[\![X - Y - c = 0]\!]X^\sharp \stackrel{def}{=} \begin{cases} \mathcal{C}^\sharp[\![X - (X^\sharp(Y) + c) = 0]\!]X^\sharp & \text{if } X^\sharp(Y) \notin \{\bot_b^\sharp, \top_b^\sharp\} \\ X^\sharp & \text{otherwise} \end{cases} \tag{4.75}$$

$$\sqcap^\sharp \begin{cases} \mathcal{C}^\sharp[\![Y - (X^\sharp(X) - c) = 0]\!]X^\sharp & \text{if } X^\sharp(X) \notin \{\bot_b^\sharp, \top_b^\sharp\} \\ X^\sharp & \text{otherwise} \end{cases} \tag{4.76}$$

Constant Analysis Example

$\mathcal{B}^\sharp$ has finite height, so the iterations $(X_\ell^{\sharp i})$ converge in finite time (even though $\mathcal{B}^\sharp$ is infinite).

Analysis example:

```
X:=0; Y:=10;
while X<100 do
    Y:=Y-3;
    X:=X+Y;
    Y:=Y+3
done
```

The constant analysis finds, at the point marked •, the invariant:

$$\begin{cases} X = \top_b^\sharp \\ Y = 7 \end{cases} \tag{4.77}$$

Note: the analysis can find constants that do not appear syntactically in the program.

4.5.4 The Interval Domain

The interval domain tracks ranges of possible values for each variable. It was introduced by [**?**].

$$\mathcal{B}^\sharp \stackrel{def}{=} \{[a,b] \mid a \in \mathbb{I} \cup \{-\infty\}, b \in \mathbb{I} \cup \{+\infty\}, a \leq b\} \cup \{\bot_b^\sharp\} \tag{4.78}$$

The partial ordering is:

$$[a,b] \sqsubseteq_b [c,d] \stackrel{def}{\Leftrightarrow} a \leq c \text{ and } b \leq d \tag{4.79}$$

The least upper bound and greatest lower bound are:

$$[a,b] \sqcup_b^\sharp [c,d] \stackrel{def}{=} [\min(a,c), \max(b,d)] \tag{4.80}$$

$$[a,b] \sqcap_b^\sharp [c,d] \stackrel{def}{=} \begin{cases} [\max(a,c), \min(b,d)] & \text{if } \max(a,c) \leq \min(b,d) \\ \bot_b^\sharp & \text{otherwise} \end{cases} \tag{4.81}$$

The top element is $\top_b^\sharp \stackrel{def}{=} [-\infty, +\infty]$.
Note: intervals are open at infinite bounds $+\infty, -\infty$.

Interval Abstract Arithmetic Operators

$$[c,c']_b^\sharp \stackrel{def}{=} [c,c'] \tag{4.82}$$

$$-_b^\sharp [a,b] \stackrel{def}{=} [-b,-a] \tag{4.83}$$

$$[a,b] +_b^\sharp [c,d] \stackrel{def}{=} [a+c, b+d] \tag{4.84}$$

$$[a,b] -_b^\sharp [c,d] \stackrel{def}{=} [a-d, b-c] \tag{4.85}$$

$$[a,b] \times_b^\sharp [c,d] \stackrel{def}{=} [\min(ac,ad,bc,bd), \max(ac,ad,bc,bd)] \tag{4.86}$$

Division requires special care to handle division by zero:

$$[a,b]/_b^\sharp[c,d] \stackrel{def}{=} \begin{cases} \bot_b^\sharp & \text{if } c = d = 0 \\ [\min(a/c, a/d, b/c, b/d), & \text{else if } 0 \leq c \\ \max(a/c, a/d, b/c, b/d)] & \\ [-b,-a]/_b^\sharp[-d,-c] & \text{else if } d \leq 0 \\ ([a,b]/_b^\sharp[c,0]) \sqcup_b^\sharp ([a,b]/_b^\sharp[0,d]) & \text{otherwise} \end{cases} \tag{4.87}$$

where $\pm\infty \times 0 = 0$, $0/0 = 0$, $\forall x, x/\pm\infty = 0$, $\forall x > 0, x/0 = +\infty$, $\forall x < 0, x/0 = -\infty$.

Operators are strict: $-_b^\sharp \bot_b^\sharp = \bot_b^\sharp$, $[a,b] +_b^\sharp \bot_b^\sharp = \bot_b^\sharp$, etc.

Interval Abstract Tests

If $X^\sharp(X) = [a, b]$ and $X^\sharp(Y) = [c, d]$, we can define:

$$\mathcal{C}^\sharp[\![X - c \leq 0]\!]X^\sharp \stackrel{def}{=} \begin{cases} \bot^\sharp & \text{if } a > c \\ X^\sharp[X \mapsto [a, \min(b, c)]] & \text{otherwise} \end{cases} \tag{4.88}$$

$$\mathcal{C}^\sharp[\![X - Y \leq 0]\!]X^\sharp \stackrel{def}{=} \begin{cases} \bot^\sharp & \text{if } a > d \\ X^\sharp[X \mapsto [a, \min(b, d)], Y \mapsto [\max(c, a), d]] & \text{otherwise} \end{cases} \tag{4.89}$$

$$\mathcal{C}^\sharp[\![e \bowtie 0]\!]X^\sharp \stackrel{def}{=} X^\sharp \text{ otherwise} \tag{4.90}$$

Note: fallback operators $\mathcal{C}^\sharp[\![e \bowtie 0]\!]X^\sharp = X^\sharp$ and $\mathcal{C}^\sharp[\![X := e]\!]X^\sharp = X^\sharp[X \mapsto \top_b^\sharp]$ are always sound.

Generic Non-Relational Abstract Test

A more sophisticated approach associates an abstract value in $\mathcal{B}^\sharp$ to each expression node using two traversals of the expression tree:

- First, a bottom-up evaluation using forward operators $\odot_b^\sharp$
- Apply $\overline{\bowtie 0}_b^\sharp$ to the root
- Then, a top-down refinement using backward operators $\overline{\odot}_b^\sharp$

For each expression leaf, we get an abstract value $V_b^\sharp$:

- For a variable V, replace $X^\sharp(V)$ with $X^\sharp(V) \sqcap_b^\sharp V_b^\sharp$
- For a constant $[c, c']$, check that $[c, c']_b^\sharp \sqcap_b^\sharp V_b^\sharp \neq \bot_b^\sharp$

Return $\bot^\sharp$ if some $\sqcap_b^\sharp V_b^\sharp$ returns $\bot_b^\sharp$.
This approach can be refined further with local iterations.

Interval Test Example

Example: $\mathcal{C}^\sharp[\![X + Y - Z \leq 0]\!]X^\sharp$ with $X^\sharp = \{X \mapsto [0, 10], Y \mapsto [2, 10], Z \mapsto [3, 5]\}$

Bottom-up evaluation:

$$X \mapsto [0, 10] \tag{4.91}$$
$$Y \mapsto [2, 10] \tag{4.92}$$
$$X + Y \mapsto [2, 20] \tag{4.93}$$
$$Z \mapsto [3, 5] \tag{4.94}$$
$$(X + Y) - Z \mapsto [-3, 17] \tag{4.95}$$

Test ≤ 0: $[-3,17] \cap [-\infty,0] = [-3,0]$
Top-down backward refinement:

$$(X+Y)-Z \mapsto [-3,0] \tag{4.96}$$
$$Z \mapsto [3,5] \tag{4.97}$$
$$X+Y \mapsto [3-0, 5-(-3)] = [3,8] \tag{4.98}$$
$$X \mapsto [0,3] \tag{4.99}$$
$$Y \mapsto [2,5] \tag{4.100}$$

Interval Widening

Widening on non-relational domains extends a value widening $\nabla_b : \mathcal{B}^\sharp \times \mathcal{B}^\sharp \to \mathcal{B}^\sharp$ point-wise:

$$X^\sharp \nabla Y^\sharp \stackrel{def}{=} V.(X^\sharp(V) \nabla_b Y^\sharp(V)) \tag{4.101}$$

Interval widening example:

$$\bot_b^\sharp \nabla_b X^\sharp \stackrel{def}{=} X^\sharp \tag{4.102}$$
$$[a,b]\nabla_b[c,d] \stackrel{def}{=} \begin{cases} a & \text{if } a \leq c \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ +\infty & \text{otherwise} \end{cases} \tag{4.103}$$

Unstable bounds are set to $\pm\infty$.

Analysis with Widening Example

Analysis example with widening points $W = \{2\}$:

```
X:=0;
while X<40 do
    X:=X+1
done
```

ℓ	$X_\ell^{\sharp 0}$	$X_\ell^{\sharp 1}$	$X_\ell^{\sharp 2}$	$X_\ell^{\sharp 3}$	$X_\ell^{\sharp 4}$	$X_\ell^{\sharp 5}$
1	$\top^\sharp$	$\top^\sharp$	$\top^\sharp$	$\top^\sharp$	$\top^\sharp$	$\top^\sharp$
2	$\bot^\sharp \nabla = 0$	$= 0$	$= 0$	$\in [0,\infty)$	$\in [0,\infty)$	$\in [0,\infty)$
3	$\bot^\sharp$	$\bot^\sharp$	$= 0$	$= 0$	$\in [0,39]$	$\in [0,39]$
4	$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$	≥ 40	≥ 40

Table 4.2: Interval analysis with widening

More precisely, at the widening point:

$$X_2^{\sharp 1} = \bot^\sharp \nabla_b([0,0] \sqcup_b^\sharp \bot^\sharp) = \bot^\sharp \nabla_b [0,0] = [0,0] \tag{4.104}$$

$$X_2^{\sharp 2} = [0,0] \nabla_b([0,0] \sqcup_b^\sharp \bot^\sharp) = [0,0] \nabla_b [0,0] = [0,0] \tag{4.105}$$

$$X_2^{\sharp 3} = [0,0] \nabla_b([0,0] \sqcup_b^\sharp [1,1]) = [0,0] \nabla_b [0,1] = [0,+\infty) \tag{4.106}$$

$$X_2^{\sharp 4} = [0,+\infty) \nabla_b([0,0] \sqcup_b^\sharp [1,40]) = [0,+\infty) \nabla_b [0,40] = [0,+\infty) \tag{4.107}$$

Note that the most precise interval abstraction would be $X \in [0,40]$ at point 2, and $X = 40$ at point 4.

Narrowing

Using a widening makes the analysis less precise. Some precision can be retrieved using a narrowing operator $\triangle$.

Definition: A narrowing $\triangle$ is a binary operator $\mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ such that:

- $(X^\sharp \sqcap^\sharp Y^\sharp) \sqsubseteq (X^\sharp \triangle Y^\sharp) \sqsubseteq X^\sharp$
- For all sequences $(X_i^\sharp)$, the decreasing sequence $(Y_i^\sharp)$ defined by $Y_0^\sharp \stackrel{def}{=} X_0^\sharp$, $Y_{i+1}^\sharp \stackrel{def}{=} Y_i^\sharp \triangle X_{i+1}^\sharp$ is stationary

This is not the dual of a widening!

Examples of narrowing:

- Trivial narrowing: $X^\sharp \triangle Y^\sharp \stackrel{def}{=} X^\sharp$
- Finite-time intersection narrowing:

$$X^{\sharp i} \triangle Y^\sharp \stackrel{def}{=} \begin{cases} X^{\sharp i} \sqcap^\sharp Y^\sharp & \text{if } i \leq N \\ X^{\sharp i} & \text{if } i > N \end{cases} \tag{4.108}$$

- Interval narrowing:

$$[a,b] \triangle_b [c,d] \stackrel{def}{=} \begin{cases} c & \text{if } a = -\infty \\ a & \text{otherwise} \end{cases}, \begin{cases} d & \text{if } b = +\infty \\ b & \text{otherwise} \end{cases} \tag{4.109}$$

Point-wise extension to $\mathcal{D}^\sharp$: $X^\sharp \triangle Y^\sharp \stackrel{def}{=} V.(X^\sharp(V) \triangle_b Y^\sharp(V))$

Iterations with Narrowing

Let $X_\ell^\sharp$ be the result after widening stabilization:

$$X_\ell^\sharp \sqsupseteq \begin{cases} \top^\sharp & \text{if } \ell = e \\ \bigsqcup_{(\ell',c,\ell) \in \mathcal{A}}^\sharp \mathcal{C}^\sharp[\![c]\!] X_{\ell'}^\sharp & \text{if } \ell \neq e \end{cases} \tag{4.110}$$

The following sequence is computed:

$$Y_\ell^{\sharp 0} \stackrel{def}{=} X_\ell^\sharp \tag{4.111}$$

$$Y_\ell^{\sharp i+1} \stackrel{def}{=} \begin{cases} \top^\sharp & \text{if } \ell = e \\ \bigsqcup^\sharp_{(\ell',c,\ell)\in\mathcal{A}} \mathcal{C}^\sharp[\![c]\!] Y_{\ell'}^{\sharp i} & \text{if } \ell \notin W \\ Y_\ell^{\sharp i} \triangle \bigsqcup^\sharp_{(\ell',c,\ell)\in\mathcal{A}} \mathcal{C}^\sharp[\![c]\!] Y_{\ell'}^{\sharp i} & \text{if } \ell \in W \end{cases} \tag{4.112}$$

The sequence $(Y_\ell^{\sharp i})$ is decreasing and converges in finite time, and all $(Y_\ell^{\sharp i})$ are solutions of the abstract semantic system.

Analysis with Narrowing Example

Example with $W = \{2\}$:

```
X:=0;
while X<40 do
    X:=X+1
done
```

ℓ	$Y_\ell^{\sharp 0}$	$Y_\ell^{\sharp 1}$	$Y_\ell^{\sharp 2}$	$Y_\ell^{\sharp 3}$
1	$\top^\sharp$	$\top^\sharp$	$\top^\sharp$	$\top^\sharp$
2	$\in [0,\infty)\triangle$	$\in [0,40]$	$\in [0,40]$	$\in [0,40]$
3	$\in [0,39]$	$\in [0,39]$	$\in [0,39]$	$\in [0,39]$
4	≥ 40	≥ 40	$= 40$	$= 40$

Table 4.3: Interval analysis with narrowing

Narrowing at point 2 gives:

$$Y_2^{\sharp 1} = [0,+\infty)\triangle_b([0,0] \sqcup_b^\sharp [1,40]) = [0,+\infty)\triangle_b[0,40] = [0,40] \tag{4.113}$$

$$Y_2^{\sharp 2} = [0,40]\triangle_b([0,0] \sqcup_b^\sharp [1,40]) = [0,40]\triangle_b[0,40] = [0,40] \tag{4.114}$$

Then $Y_2^{\sharp 2} : X \in [0,40]$ gives $Y_4^{\sharp 3} : X = 40$.
We've found the most precise invariants!

Improving Widening

Widening can sometimes lead to imprecise results. For example:

```
X:=40;
while X<>0 do
    X:=X-1
done
```

The interval domain with standard widening cannot prove that $X \geq 0$ at the loop point, while the simpler sign domain can! This happens because the interval widening jumps immediately to $[-\infty, 40]$.

We can improve the interval widening to check the stability of specific values like 0:

$$[a,b]\nabla'_b[c,d] \stackrel{def}{=} \begin{cases} a & \text{if } a \leq c \\ 0 & \text{if } 0 \leq c < a \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ 0 & \text{if } 0 \geq b > d \\ +\infty & \text{otherwise} \end{cases} \tag{4.115}$$

This extended widening checks the stability of 0, enhancing precision in many practical cases.

Widening with Thresholds

Another approach is to use widening with thresholds. Given a finite set T of thresholds containing $+\infty$ and $-\infty$:

$$[a,b]\nabla^T_b[c,d] \stackrel{def}{=} \begin{cases} a & \text{if } a \leq c \\ \max\{x \in T \mid x \leq c\} & \text{otherwise} \end{cases}, \begin{cases} b & \text{if } b \geq d \\ \min\{x \in T \mid x \geq d\} & \text{otherwise} \end{cases} \tag{4.116}$$

The widening tests thresholds and stops at the first stable bound in T.

This approach is useful when:

- It's easy to find a "good" set T (e.g., array bound-checking, arithmetic overflow checking)
- An over-approximation of the bound is sufficient

But it only works if some non-∞ bound in T is stable.

4.5.5 The Congruence Domain

The congruence domain tracks whether variables have values that satisfy congruence properties (e.g., being even, odd, or congruent to some value modulo some number).

The domain is:

$$\mathcal{B}^\sharp \stackrel{def}{=} \{(a\mathbb{Z} + b) \mid a \in \mathbb{N}, b \in \mathbb{Z}\} \cup \{\bot^\sharp_b\} \tag{4.117}$$

Where $a\mathbb{Z}+b$ is the congruence class of b modulo a, i.e., $x \in a\mathbb{Z}+b$ when a divides $x - b$.

The partial order is:

$$(a\mathbb{Z} + b) \sqsubseteq_b (a'\mathbb{Z} + b') \stackrel{def}{\Leftrightarrow} a'/a \text{ and } b \equiv b'[a'] \tag{4.118}$$

Where a/a' means a divides a', and $b \equiv b'[a']$ means b is congruent to b' modulo a'.

The top element is $\top_b^\sharp \stackrel{def}{=} (1\mathbb{Z} + 0)$.

This lattice satisfies the Ascending Chain Condition (ACC) but not the Descending Chain Condition (DCC).

Congruence Operators

Abstract arithmetic operators:

$$[c, c']_b^\sharp \stackrel{def}{=} \begin{cases} 0\mathbb{Z} + c & \text{if } c = c' \\ \top_b^\sharp & \text{otherwise} \end{cases} \tag{4.119}$$

$$-_b^\sharp(a\mathbb{Z} + b) \stackrel{def}{=} a\mathbb{Z} + (-b) \tag{4.120}$$

$$(a\mathbb{Z} + b) +_b^\sharp (a'\mathbb{Z} + b') \stackrel{def}{=} (a \wedge a')\mathbb{Z} + (b + b') \tag{4.121}$$

$$(a\mathbb{Z} + b) -_b^\sharp (a'\mathbb{Z} + b') \stackrel{def}{=} (a \wedge a')\mathbb{Z} + (b - b') \tag{4.122}$$

$$(a\mathbb{Z} + b) \times_b^\sharp (a'\mathbb{Z} + b') \stackrel{def}{=} (aa' \wedge ab' \wedge a'b)\mathbb{Z} + bb' \tag{4.123}$$

Division is more complex and not always optimal.

Congruence Analysis Example

Example:

```
X:=0; Y:=2;
while X<40 do
    X:=X+2;
    if X<5 then Y:=Y+18 fi;
    if X>8 then Y:=Y-30 fi
done
```

We find the loop invariant:

$$\begin{cases} X \in 2\mathbb{Z} \\ Y \in 6\mathbb{Z} + 2 \end{cases} \tag{4.124}$$

This shows that Y is always congruent to 2 modulo 6, which is the greatest common divisor of 18 and 30.

4.6 Reduced Products of Domains

4.6.1 Non-Reduced Product of Domains

Product representation: Cartesian product $\mathcal{D}^\sharp_{1\times 2}$ of $\mathcal{D}^\sharp_1$ and $\mathcal{D}^\sharp_2$:

$$\mathcal{D}^\sharp_{1\times 2} \stackrel{def}{=} \mathcal{D}^\sharp_1 \times \mathcal{D}^\sharp_2 \tag{4.125}$$

$$\gamma_{1\times 2}(X^\sharp_1, X^\sharp_2) \stackrel{def}{=} \gamma_1(X^\sharp_1) \cap \gamma_2(X^\sharp_2) \tag{4.126}$$

$$\alpha_{1\times 2}(X) \stackrel{def}{=} (\alpha_1(X), \alpha_2(X)) \tag{4.127}$$

$$(X^\sharp_1, X^\sharp_2) \sqsubseteq_{1\times 2} (Y^\sharp_1, Y^\sharp_2) \stackrel{def}{\Leftrightarrow} X^\sharp_1 \sqsubseteq_1 Y^\sharp_1 \text{ and } X^\sharp_2 \sqsubseteq_2 Y^\sharp_2 \tag{4.128}$$

Abstract operators are performed in parallel on both components:

$$(X^\sharp_1, X^\sharp_2) \sqcup^\sharp_{1\times 2} (Y^\sharp_1, Y^\sharp_2) \stackrel{def}{=} (X^\sharp_1 \sqcup^\sharp_1 Y^\sharp_1, X^\sharp_2 \sqcup^\sharp_2 Y^\sharp_2) \tag{4.129}$$

$$\mathcal{C}^\sharp[\![c]\!]_{1\times 2}(X^\sharp_1, X^\sharp_2) \stackrel{def}{=} (\mathcal{C}^\sharp[\![c]\!]_1(X^\sharp_1), \mathcal{C}^\sharp[\![c]\!]_2(X^\sharp_2)) \tag{4.130}$$

Non-Reduced Product Example

The product analysis is no more precise than two separate analyses:

```
X:=1;
while X-10<=0 do
    X:=X+2
done;
•
if X-12>=0 then
    ⟨
    X:=0
    ⟩
fi
```

	interval	congruence	product
•	$X \in [11, 12]$	$X \equiv 1[2]$	$X = 11$
$\langle$	$X = 12$	$X \equiv 1[2]$	$\emptyset$
$\rangle$	$X = 0$	$X = 0$	$X = 0$

Table 4.4: Non-reduced product example

The product domain can prove that the if branch is never taken because $X = 11$ from the interval-congruence combination, and $11 < 12$.

4.6.2 Fully-Reduced Product

The fully-reduced product uses a reduction operator ρ that propagates information between domains:

$$\rho : \mathcal{D}^\sharp_{1\times 2} \to \mathcal{D}^\sharp_{1\times 2} \tag{4.131}$$

$$\rho(X^\sharp_1, X^\sharp_2) \stackrel{def}{=} (\alpha_1(\gamma_1(X^\sharp_1) \cap \gamma_2(X^\sharp_2)), \alpha_2(\gamma_1(X^\sharp_1) \cap \gamma_2(X^\sharp_2))) \tag{4.132}$$

We can reduce the result of each abstract operator, except widening:

$$(X^\sharp_1, X^\sharp_2) \sqcup^\sharp_{1\times 2} (Y^\sharp_1, Y^\sharp_2) \stackrel{def}{=} \rho(X^\sharp_1 \sqcup^\sharp_1 Y^\sharp_1, X^\sharp_2 \sqcup^\sharp_2 Y^\sharp_2) \tag{4.133}$$

$$\mathcal{C}^\sharp[\![c]\!]_{1\times 2}(X^\sharp_1, X^\sharp_2) \stackrel{def}{=} \rho(\mathcal{C}^\sharp[\![c]\!]_1(X^\sharp_1), \mathcal{C}^\sharp[\![c]\!]_2(X^\sharp_2)) \tag{4.134}$$

We refrain from reducing after a widening ∇, as this may jeopardize convergence.

Fully-Reduced Product Example

Reduction example between the interval and congruence domains:

$$\rho_b([a,b], c\mathbb{Z}+d) \stackrel{def}{=} \begin{cases} (\bot^\sharp_b, \bot^\sharp_b) & \text{if } a' > b' \\ ([a',a'], 0\mathbb{Z}+a') & \text{if } a' = b' \\ ([a',b'], c\mathbb{Z}+d) & \text{if } a' < b' \end{cases} \tag{4.135}$$

where $a' \stackrel{def}{=} \min\{x \geq a \mid x \equiv d[c]\}$ and $b' \stackrel{def}{=} \max\{x \leq b \mid x \equiv d[c]\}$.
Extended point-wise to ρ on $\mathcal{D}^\sharp$.
Application:

$$\rho_b([10,11], 2\mathbb{Z}+1) = ([11,11], 0\mathbb{Z}+11) \tag{4.136}$$

$$\rho_b([1,3], 4\mathbb{Z}) = (\bot^\sharp_b, \bot^\sharp_b) \tag{4.137}$$

4.7 Relational Domains

4.7.1 Linear Equality Domain

The linear equality domain tracks linear equality relationships between variables. It was proposed by Karr [**?**].

We look for invariants of the form:

$$\bigwedge_j \left(\sum_{i=1}^{n} \alpha_{ij} V_i = \beta_j \right), \alpha_{ij}, \beta_j \in \mathbb{I} \tag{4.138}$$

We use a domain of affine subspaces of $\mathcal{V} \to \mathbb{I}$:

$$\mathcal{D}^\sharp \stackrel{def}{=} \{\text{affine subspaces of } \mathcal{V} \to \mathbb{I}\} \tag{4.139}$$

Linear Equality Analysis Example

Forward analysis example:

```
X:=10; Y:=100;
while X<>0 do
    X:=X-1;
    Y:=Y+10
done
```

ℓ	$X_\ell^{\sharp 0}$	$X_\ell^{\sharp 1}$	$X_\ell^{\sharp 2}$	$X_\ell^{\sharp 3}$	$X_\ell^{\sharp 4}$
1	$\top^\sharp$	$\top^\sharp$	$\top^\sharp$	$\top^\sharp$	$\top^\sharp$
2	$\bot^\sharp$	$(10, 100)$	$(10, 100)$	$10X + Y = 200$	$10X + Y = 200$
3	$\bot^\sharp$	$\bot^\sharp$	$(10, 100)$	$(10, 100)$	$10X + Y = 200$
4	$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$	$\bot^\sharp$	$(0, 200)$

Table 4.5: Linear equality analysis

Note that:

$$X_2^{\sharp 3} = \{(10, 100)\} \sqcup^\sharp \{(9, 110)\} = \{(X, Y) \mid 10X + Y = 200\} \tag{4.140}$$

This represents a line including $(10, 100)$ and $(9, 110)$.

4.7.2 Polyhedron Domain

The polyhedron domain tracks linear inequality relationships between variables. It was proposed by Cousot and Halbwachs [**?**].

We look for invariants of the form:

$$\bigwedge_j \left(\sum_{i=1}^{n} \alpha_{ij} V_i \leq \beta_j \right) \tag{4.141}$$

We use the polyhedron domain:

$$\mathcal{D}^\sharp \stackrel{def}{=} \{\text{closed convex polyhedra of } \mathcal{V} \to \mathbb{I}\} \tag{4.142}$$

Note: polyhedra need not be bounded (polytopes).

Polyhedron Example Analysis

Example program:

```
X:=2; I:=0;
while • I<10 do
    if [0,1]=0 then X:=X+2 else X:=X-3 fi;
    I:=I+1
done⟨
```

Using widening and narrowing at $\bullet$:

$$X_\bullet^{\sharp 1} = \{X = 2, I = 0\} \tag{4.143}$$
$$X_\bullet^{\sharp 2} = \{X = 2, I = 0\} \nabla (\{X = 2, I = 0\} \sqcup^\sharp \{X \in [-1, 4], I = 1\}) \tag{4.144}$$
$$= \{X = 2, I = 0\} \nabla \{I \in [0, 1], 2 - 3I \leq X \leq 2I + 2\} \tag{4.145}$$
$$= \{I \geq 0, 2 - 3I \leq X \leq 2I + 2\} \tag{4.146}$$
$$X_\bullet^{\sharp 3} = \{I \geq 0, 2 - 3I \leq X \leq 2I + 2\} \triangle \tag{4.147}$$
$$(\{X = 2, I = 0\} \sqcup^\sharp \{I \in [1, 10], 2 - 3I \leq X \leq 2I + 2\}) \tag{4.148}$$
$$= \{I \in [0, 10], 2 - 3I \leq X \leq 2I + 2\} \tag{4.149}$$

At $\langle$ we find: $I = 10 \wedge X \in [-28, 22]$.

4.7.3 Zone Domain

The zone domain tracks constraints of the form $V_i - V_j \leq c$ or $\pm V_i \leq c$ with $c \in \mathbb{I}$. It was introduced by Miné [**?**].

A subset of $\mathbb{I}^n$ bounded by such constraints is called a zone.

Representation is via potential graphs or Difference Bound Matrices (DBMs):

- A DBM m is square with size $n \times n$ and elements in $\mathbb{I} \cup \{+\infty\}$
- $m_{ij} = c < +\infty$ denotes the constraint $V_j - V_i \leq c$
- $m_{ij} = +\infty$ if there is no upper bound on $V_j - V_i$

Concretization: $\gamma(m) \stackrel{def}{=} \{(v_1, \ldots, v_n) \in \mathbb{I}^n \mid \forall i, j, v_j - v_i \leq m_{ij}\}$

Unary constraints add a constant null variable V_0:

- m has size $(n + 1) \times (n + 1)$
- $V_i \leq c$ is denoted as $V_i - V_0 \leq c$, i.e., $m_{i0} = c$
- $-V_i \leq c$ is denoted as $V_0 - V_i \leq c$, i.e., $m_{0i} = c$

Normal form uses shortest-path closure: $m^*_{ij} \stackrel{def}{=} \min_{hi=i_1,\ldots,i_N=ji} \sum_{k=1}^{N-1} m_{i_k i_{k+1}}$

This exists only when m has no cycle with strictly negative weight.

The octagon domain, as we began to explain, tracks constraints of the form $\pm V_i \pm V_j \leq c$ with $c \in \mathbb{I}$. The constraint encoding uses a variable change to get back to potential constraints:

- Let $\mathcal{V}' \stackrel{def}{=} V'1, \ldots, V'2n$ where $V_i \mapsto V'2i - 1$ and $-V_i \mapsto V'2i$
- The constraint $V_i - V_j \leq c$ (for $i \neq j$) is encoded as $V'2i - 1 - V'2j - 1 \leq c$ and $V'2j - V'2i \leq c$

- The constraint $V_i + V_j \leq c$ (for $i \neq j$) is encoded as $V'2i - 1 - V'2j \leq c$ and $V'2j - 1 - V'2i \leq c$
- The constraint $-V_i - V_j \leq c$ (for $i \neq j$) is encoded as $V'2j - V'2i - 1 \leq c$ and $V'2i - V'2j - 1 \leq c$
- The constraint $V_i \leq c$ is encoded as $V'2i - 1 - V'2i \leq 2c$
- The constraint $-V_i \leq c$ is encoded as $V'2i - V'2i - 1 \leq 2c$

We use a matrix m of size $(2n) \times (2n)$ with elements in $\mathbb{I} \cup +\infty$, and define:

$$\gamma_{\pm}(m) \stackrel{def}{=} (v_1, \ldots, v_n) \mid (v_1, -v_1, \ldots, v_n, -v_n) \in \gamma(m) \tag{4.150}$$

Note: To ensure a coherent representation, we impose that $\forall i, j, m_{ij} = m_{\overline{j}\overline{i}}$ where $\bar{\imath} = i \oplus 1$ (i.e., flip the parity of i).

Example of Octagon Analysis

As an example, consider the following program:

```
var X: int, Y: int;
begin
X=10;
Y=0;
while (X>=0) do
X = X-1;
if brandom then Y = Y+1; endif;
done;
end
```

Using the octagon domain, we can derive the loop invariant:

$$-1 \leq X \leq 10, ; 0 \leq Y \leq 11, ; -12 \leq X - Y \leq 10, ; -1 \leq X + Y \leq 10 \tag{4.151}$$

This provides more precise relational information than what would be possible with just interval analysis.

4.8 Summary and Comparison of Domains

The selection of an appropriate domain involves a trade-off between precision and computational complexity:

- **Non-relational domains** (sign, constant, interval, congruence) are efficient but cannot capture relationships between variables.

Domain	Non-relational	Linear equalities	Polyhedra	Octagons
$V \in \mathcal{B}_b^\sharp$	$\sum_i \alpha_i V_i = \beta$	$\sum_i \alpha_i V_i \leq \beta$	$\pm V_i \pm V_j \leq c$ heightMemory cost	
$O(n^2)$	$O(2^n)$	$O(n^2)$ heightTime cost	$O(n)$	
$O(2^n)$	$O(n^3)$ height			

Table 4.6: Summary of numerical abstract domains

- **Weakly relational domains** (zone, octagon) offer a good compromise, capturing some relationships with reasonable efficiency.
- **Fully relational domains** (linear equalities, polyhedra) can express rich relationships but are computationally expensive.

Many modern analyzers use combinations of domains (via reduced products) or domain switching strategies to balance precision and cost for different program constructs.

4.9 Practical Considerations

4.9.1 Widening Strategies

The choice of widening strategy can significantly impact analysis precision:

- **Widening points**: Selecting an optimal set of widening points can reduce precision loss. Typically, one point per cycle in the control flow graph is sufficient.
- **Widening with thresholds**: Using constants from the program as thresholds often improves precision significantly.
- **Delayed widening**: Applying standard iteration for a few steps before starting to widen can preserve important invariants.
- **Narrowing**: Always follow widening with narrowing iterations to recover precision.

4.9.2 Domain Selection Guidelines

Selecting an appropriate abstract domain depends on the program properties of interest:

- For simple range analysis (e.g., array bounds checking), the interval domain is often sufficient.
- For analyzing modular arithmetic or bit-level operations, the congruence domain is valuable.

- When loop invariants involve simple variable relationships (e.g., $x \leq y$, $x + y \leq c$), octagons offer a good trade-off.
- For sophisticated numerical invariants with arbitrary linear constraints, the polyhedron domain is appropriate, despite its cost.
- When analyzing data structures, specialized domains for shapes, heaps, or arrays may be necessary.

In practice, domain combinations (via reduced products) often provide the best results.

4.9.3 Implementation Considerations

When implementing abstract interpretation:

- **Floating-point arithmetic**: Real-world analyzers must handle floating-point issues carefully, often using sound approximations or interval arithmetic.
- **Sparse representations**: For large programs, sparse matrix representations can significantly reduce memory consumption for relational domains.
- **Incremental computations**: Recomputing only affected parts of abstract states after small program changes improves analyzer responsiveness.
- **Parallelization**: Some abstract domains support parallel computations, which can improve analysis speed on modern hardware.

4.10 Real-World Applications

Abstract interpretation has been successfully applied in various software verification contexts:

- **ASTRÉE**: An abstract interpretation-based analyzer that proved the absence of runtime errors in the Airbus A380 flight control software.
- **CodeHawk**: A commercial tool targeting security vulnerabilities in C and Java code.
- **Clousot**: A static analyzer for .NET programs that verifies absence of runtime errors.
- **Frama-C**: An open-source framework for analyzing C programs with various abstract domains.

- **IKOS**: An LLVM-based framework for static analysis through abstract interpretation.

Modern compilers like GCC and LLVM also incorporate abstract interpretation principles for their optimization passes.

4.11 Advanced Topics

4.11.1 Handling Advanced Language Features

Real-world programming languages include features that require specialized abstract domains:

- **Pointers and aliasing**: Memory abstractions such as shape analysis and separation logic.
- **Dynamic allocation**: Domains that track heap properties and detect memory leaks.
- **Arrays and containers**: Specialized domains like array regions or array segmentation.
- **Concurrency**: Thread-modular analysis, rely-guarantee reasoning, or analysis of synchronization primitives.
- **Objects and inheritance**: Class hierarchy analysis and properties preserved by polymorphism.

4.11.2 Trace Partitioning

Trace partitioning improves precision by separately analyzing program paths:

- **Path-sensitive analysis**: Maintaining distinct abstract states for different execution paths.
- **Context-sensitive analysis**: Differentiating function calls based on their calling context.
- **Value-based partitioning**: Separating analysis based on specific variable values.

This approach reduces the loss of precision from merging control flows but increases the analysis cost.

4.11.3 Modular Analysis

For large programs, modular analysis is essential:

- **Procedure summaries**: Computing reusable abstract transformers for functions.
- **Contract-based verification**: Using pre/post-conditions to analyze components independently.
- **Bottom-up analysis**: Analyzing called procedures before their callers to build summaries.
- **Top-down analysis with placeholders**: Using hypotheses about called procedures during the analysis of callers.

As seen in our slides, relational domains are particularly important for creating precise procedure summaries that capture input-output relationships.

4.12 Conclusion

Abstract interpretation provides a powerful theoretical framework for static program analysis. By systematically approximating the semantics of programs, it enables sound verification of properties that would be undecidable in general. The key advantages of abstract interpretation include:

- **Soundness**: When properly implemented, it guarantees the absence of false negatives.
- **Automation**: Once set up, analyses require minimal user intervention.
- **Scalability**: With appropriate domains and widening strategies, it can scale to large programs.
- **Adaptability**: The framework accommodates diverse program properties through custom abstract domains.

The main challenges involve selecting appropriate abstract domains, implementing efficient algorithms, and managing the precision-performance trade-off. As software systems become increasingly complex and deployed in safety-critical contexts, abstract interpretation-based verification tools play a vital role in ensuring reliability, security, and correctness.

4.13 Exercises

The following exercises can help solidify understanding of abstract interpretation concepts:

Exercise 1. Consider the program:

```
X := 0;
Y := 0;
while X < 10 do
X := X + 1;
Y := Y + X;
done
```

Analyze this program using:

1. The interval domain
2. The linear equality domain

What invariants can each domain establish at the loop exit?

Exercise 2. Design a widening operator for the sign domain that preserves more information than the standard widening. Apply it to an example program to demonstrate its effectiveness.

Exercise 3. Consider a reduced product of the interval and congruence domains. Show how the reduction operator would process the abstract states $X \in [1, 10]$ and $X \in 2\mathbb{Z}$ to obtain a more precise result.

Exercise 4. Implement a simple interval analysis for a small imperative language, including widening and narrowing operations. Test it on programs with loops to verify its effectiveness.

Exercise 5. Consider the octagon constraints for two variables X and Y. Manually construct the DBM representation for the constraints $X \leq 10$, $Y \leq 5$, $X + Y \leq 12$, $X - Y \leq 7$. Then apply the shortest-path closure algorithm to compute the normal form.

Exercise 6. For the following program:

```
X := 0;
Y := 50;
while X < 100 do
X := X + 1;
if [0,1] = 0 then
Y := Y + 1;
else
Y := Y - 1;
endif;
done
```

What properties can be established using the interval domain? What about using the polyhedron domain? Explain the differences in precision.

Chapter 5

Advanced Topics in Software Verification

This chapter extends our exploration of software verification with advanced techniques for computing fixpoints and the emerging field of neural network verification. We begin by examining efficient algorithms for fixpoint computation, which form the basis of many static analyses. Then, we delve into the formal verification of neural networks, examining how abstract interpretation techniques can be applied to ensure their reliability and robustness.

5.1 Efficient Fixpoint Computation

5.1.1 Fixpoint Algorithms

Fixpoint computation is a fundamental operation in static program analysis, underpinning dataflow analysis, abstract interpretation, and many other verification techniques. Let $F : C^n \to C^n$ be a monotone operator on the product complete lattice C^n. According to the Kleene-Knaster-Tarski Theorem, its least fixpoint $\text{lfp}(F) = \bigvee_{i \in \mathbb{N}} F^i(\bot)$ can be computed by iteratively applying F until convergence.

However, naive implementations of this iterative process can be inefficient. We present three progressively more efficient algorithms for fixpoint computation.

Naive Iteration

The naive algorithm directly implements the Kleene-Knaster-Tarski Theorem:

$(x_1, \ldots, x_n) := \langle \bot, \ldots, \bot \rangle$ $(t_1, \ldots, t_n) := (x_1, \ldots, x_n)$ $(x_1, \ldots, x_n) := F(x_1, \ldots, x_n)$ $(x_1, \ldots, x_n) = (t_1, \ldots, t_n)$

This algorithm recomputes all components in each iteration, even when some components may not change.

Chaotic Iteration

Chaotic iteration improves on the naive approach by exploiting the product structure of the lattice and updating components in a specific order:

$x_1 := \bot; \ldots; x_n := \bot$ $t_1 := x_1; \ldots; t_n := x_n$ $x_1 := F_1(x_1, \ldots, x_n); \ldots; x_n := F_n(x_1, \ldots, x_n)$ $x_1 \neq t_1 \vee \ldots \vee x_n \neq t_n$

With this approach, the update of x_k can exploit the already updated values of $x_1, \ldots, x_{k-1}$. The ordering of variables can significantly affect efficiency, and different orderings may be appropriate for different problems.

Worklist Algorithm

Both previous algorithms recompute all components in each iteration, even when some components cannot change. The worklist algorithm addresses this by tracking dependencies between components and only recomputing when necessary:

First, we define a dependency function between components:

$$\text{dep}(x_j) \triangleq \{x_k \mid \text{the } k\text{-th equation defining } F_k(x_1, \ldots, x_n) \text{ depends on the variable } x_j\} \tag{5.1}$$

For example, in the system:

$$F_1(x_1, \ldots, x_5) = x_1 \cup x_3 \tag{5.2}$$
$$F_2(x_1, \ldots, x_5) = \emptyset \tag{5.3}$$
$$F_3(x_1, \ldots, x_5) = x_2 \cap x_5 \tag{5.4}$$
$$F_4(x_1, \ldots, x_5) = x_4 \tag{5.5}$$
$$F_5(x_1, \ldots, x_5) = F_3(x_1, x_1, x_1, x_1, x_4) \tag{5.6}$$

We have:

$$\text{dep}(x_1) = \{x_1, x_5\} \tag{5.7}$$
$$\text{dep}(x_2) = \{x_3\} \tag{5.8}$$
$$\text{dep}(x_3) = \{x_1\} \tag{5.9}$$
$$\text{dep}(x_4) = \{x_4, x_5\} \tag{5.10}$$
$$\text{dep}(x_5) = \{x_3\} \tag{5.11}$$

The worklist algorithm uses these dependencies to focus computation:

$x_1 := \bot; \ldots; x_n := \bot$ $W := [v_1, \ldots, v_n]$ v_i represents the i-th component $W \neq []$ $v_k := \text{head}(W)$ x_k is the component to update $W := \text{tail}(W)$ $y := F_k(x_1, \ldots, x_n)$ $y \neq x_k$ if x_k has been updated $v \in \text{dep}(v_k)$ $W := [v].\text{append}(W)$ add dependent components to worklist $x_k := y$

This algorithm significantly reduces the number of component updates required to reach a fixpoint.

5.1.2 Example: Sign Analysis with Different Algorithms

Consider the following simple program and its associated control flow graph:

```
X := 0;
while X < 40 do
    X := X + 1
done
```

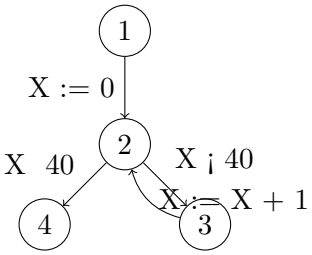


Figure 5.1: Control flow graph for the example program

Using the sign domain, the dataflow equations are:

$$X_2^{i+1} = C^\sharp[\![X := 0]\!]X_1^i \cup C^\sharp[\![X := X + 1]\!]X_3^i \tag{5.12}$$
$$X_3^{i+1} = C^\sharp[\![X < 40]\!]X_2^i \tag{5.13}$$
$$X_4^{i+1} = C^\sharp[\![X \geq 40]\!]X_2^i \tag{5.14}$$

The dependency function is:

$$\mathrm{dep}(X_1^\sharp) = \{X_2^\sharp\} \tag{5.15}$$
$$\mathrm{dep}(X_2^\sharp) = \{X_3^\sharp, X_4^\sharp\} \tag{5.16}$$
$$\mathrm{dep}(X_3^\sharp) = \{X_2^\sharp\} \tag{5.17}$$
$$\mathrm{dep}(X_4^\sharp) = \emptyset \tag{5.18}$$

With naive iteration, six iterations are needed to reach the fixpoint:

ℓ	$X_\ell^{\sharp 0}$	$X_\ell^{\sharp 1}$	$X_\ell^{\sharp 2}$	$X_\ell^{\sharp 3}$	$X_\ell^{\sharp 4}$	$X_\ell^{\sharp 5}$
1	$\top^\sharp$	$\top^\sharp$	$\top^\sharp$	$\top^\sharp$	$\top^\sharp$	$\top^\sharp$
2	$\bot^\sharp$	$X = 0$	$X = 0$	$X \geq 0$	$X \geq 0$	$X \geq 0$
3	$\bot^\sharp$	$\bot^\sharp$	$X = 0$	$X = 0$	$X \geq 0$	$X \geq 0$
4	$\bot^\sharp$	$\bot^\sharp$	$X = 0$	$X = 0$	$X \geq 0$	$X \geq 0$

Using chaotic iteration with ordering $1, 2, 3, 4$, only three iterations are needed:

ℓ	$X_\ell^{\sharp 0}$	$X_\ell^{\sharp 1}$	$X_\ell^{\sharp 2}$
1	$\top^\sharp$	$\top^\sharp$	$\top^\sharp$
2	$\bot^\sharp$	$X = 0$	$X \geq 0$
3	$\bot^\sharp$	$X = 0$	$X \geq 0$
4	$\bot^\sharp$	$\bot^\sharp$	$X \geq 0$

The worklist algorithm would further reduce computations by prioritizing components that actually change.

5.2 Neural Network Verification

5.2.1 Introduction to Neural Networks

A neural network is a directed graph where each node performs an operation. Overall, the network represents a function $f : \mathbb{R}^n \to \mathbb{R}^m$. Consider the following simple neural network:

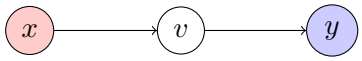


Figure 5.2: A simple neural network

The red node is an input node; it passes input x to node v. Node v performs some operation and produces a value for the output node y. For example, v might compute $f_v(x) = 2x + 1$, and y might apply a ReLU activation function $f_y(x) = \max(0, x)$.

Together, this network computes:

$$f(x) = f_y(f_v(x)) = \max(0, 2x + 1) \tag{5.19}$$

Transformations and Activations

Neural networks typically consist of two types of operations:

- **Affine transformations**: Functions that multiply inputs by constants and add constant values (e.g., $f(x) = 2x + 1$)
- **Activation functions**: Non-linear functions that introduce non-linearity into the network

Common activation functions include:

- **ReLU (Rectified Linear Unit)**: $\text{ReLU}(x) = \max(0, x)$
- **Sigmoid**: $\sigma(x) = \frac{1}{1+e^{-x}}$

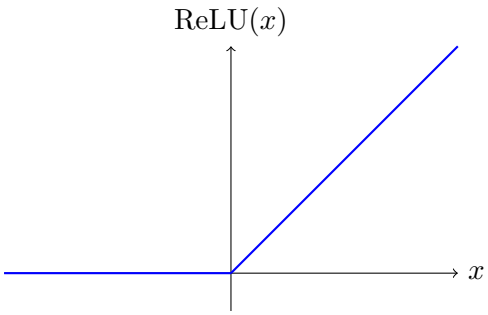


Figure 5.3: ReLU activation function

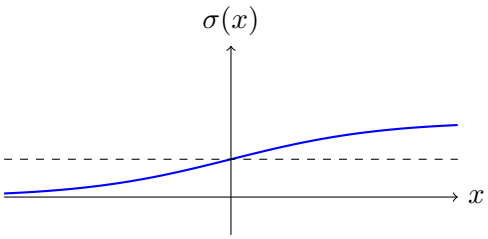


Figure 5.4: Sigmoid activation function

5.2.2 The Need for Neural Network Verification

Despite their impressive performance, neural networks can be vulnerable to adversarial examples—inputs that are imperceptibly different from normal inputs but cause the network to make incorrect predictions.

[Image of MNIST digit with adversarial perturbation]

Original image Perturbed image Difference

Figure 5.5: Example of an adversarial attack on an MNIST digit classification network

This vulnerability raises significant concerns in safety-critical applications like autonomous vehicles, medical diagnosis systems, and security applications. Formal verification techniques aim to mathematically prove that neural networks satisfy certain robustness properties, such as:

Definition 27 (Local Robustness)**.** A classifier $C : X \to L$ is robust on an input vector x for an adversarial region $P(x) \subseteq X$ when $\forall x' \in P(x), C(x') = C(x)$.

Here, $P(x)$ typically represents a small perturbation around x, such as:

$$P_\epsilon^\infty(x) = \{x' \in \mathbb{R}^n \mid \|x' - x\|_\infty \leq \epsilon\} = \{x' \in \mathbb{R}^n \mid \forall i. x'_i \in [x_i - \epsilon, x_i + \epsilon]\} \tag{5.20}$$

5.2.3 Abstract Interpretation for Neural Networks

Abstract interpretation offers a promising approach for neural network verification. We now adapt the abstract interpretation framework to neural networks.

The Interval Domain for Neural Networks

The interval domain is a natural starting point for neural network verification. We define abstract transformers for different neural network operations:

Affine Functions For an affine function $f(x_1, \ldots, x_n) = \sum_i c_i x_i$, we define:

$$f^\sharp([l_1, u_1], \ldots, [l_n, u_n]) = \left[\sum_i l'_i, \sum_i u'_i\right] \tag{5.21}$$

where $l'_i = \min(c_i l_i, c_i u_i)$ and $u'_i = \max(c_i l_i, c_i u_i)$.

Example 10. Consider $f(x, y) = 3x + 2y$. Then:

$$f^\sharp([5, 10], [20, 30]) = [3 \cdot 5 + 2 \cdot 20, 3 \cdot 10 + 2 \cdot 30] = [55, 90] \tag{5.22}$$

Monotonic Functions For a monotonically increasing function $f : \mathbb{R} \to \mathbb{R}$ (like ReLU or sigmoid), we define:

$$f^\sharp([l, u]) = [f(l), f(u)] \tag{5.23}$$

Example 11. For the ReLU function on interval $[3, 5]$:

$$\text{ReLU}^\sharp([3, 5]) = [\text{ReLU}(3), \text{ReLU}(5)] = [3, 5] \tag{5.24}$$

Composing Abstract Transformers For a function composition $h(x) = f(g(x))$, we define:

$$h^\sharp([l, u]) = f^\sharp(g^\sharp([l, u])) \tag{5.25}$$

Example 12. Let $g(x) = 3x$, $f(x) = \text{ReLU}(x)$, and $h(x) = f(g(x))$. Then:

$$h^\sharp([2, 3]) = f^\sharp(g^\sharp([2, 3])) \tag{5.26}$$

$$= f^\sharp([6, 9]) \tag{5.27}$$

$$= [6, 9] \tag{5.28}$$

Abstract Transformers for Neural Networks

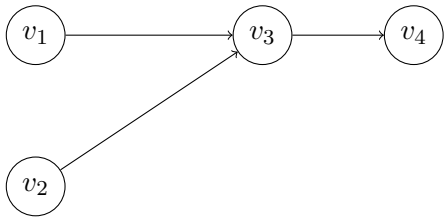
Given a neural network defined as a graph $G = (V, E)$ representing a function $f_G : \mathbb{R}^n \to \mathbb{R}^m$, we define $f_G^\sharp([l_1, u_1], \ldots, [l_n, u_n])$ as follows:

- For each input node v_i, define $\text{out}^\sharp(v_i) = [l_i, u_i]$
- For each non-input node v with incoming edges $(v_1, v), \ldots, (v_k, v)$:

$$\text{out}^\sharp(v) = f_v^\sharp(\text{out}^\sharp(v_1), \ldots, \text{out}^\sharp(v_k)) \tag{5.29}$$

  where $f_v^\sharp$ is the abstract transformer for the function f_v
- The output of $f_G^\sharp$ is the set of intervals $\text{out}^\sharp(v_1), \ldots, \text{out}^\sharp(v_m)$, where $v_1, \ldots, v_m$ are the output nodes

Example 13. Consider the neural network:



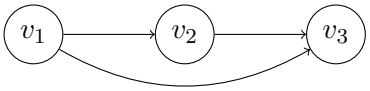
Where $f_{v_3}(x) = 2x_1 + x_2$ and $f_{v_4}(x) = \text{ReLU}(x)$.
To evaluate $f_G^\sharp([0, 1], [2, 3])$:

$$\text{out}^\sharp(v_1) = [0, 1] \tag{5.30}$$
$$\text{out}^\sharp(v_2) = [2, 3] \tag{5.31}$$
$$\text{out}^\sharp(v_3) = [2 \cdot 0 + 2, 2 \cdot 1 + 3] = [2, 5] \tag{5.32}$$
$$\text{out}^\sharp(v_4) = [\text{ReLU}(2), \text{ReLU}(5)] = [2, 5] \tag{5.33}$$

Limitations of the Interval Domain

The interval domain, while simple to implement, has limitations due to its non-relational nature. It cannot capture relationships between different values, which leads to imprecision.

Example 14. Consider the following network:



Where $f_{v_2}(x) = -x$ and $f_{v_3}(x) = x_1 + x_2$.

Clearly, for any input x, $f_G(x) = 0$ since f_{v_3} receives x and $-x$ as inputs. However, using the interval domain:

$$\text{out}^\sharp(v_1) = [0, 1] \tag{5.34}$$

$$\text{out}^\sharp(v_2) = [-1, 0] \tag{5.35}$$

$$\text{out}^\sharp(v_3) = [0, 1] + [-1, 0] = [-1, 1] \tag{5.36}$$

The interval domain cannot capture the relationship between the inputs to v_3, leading to an imprecise result.

5.3 Advanced Abstract Domains for Neural Networks

5.3.1 Beyond Intervals: The Zonotope Domain

While the interval domain is simple and efficient, it often produces overly conservative approximations. More sophisticated domains like zonotopes can provide better precision while maintaining reasonable efficiency.

A zonotope in $\mathbb{R}^n$ is a centrally symmetric convex polytope represented as the Minkowski sum of a center point and a finite set of line segments (generators):

Definition 28 (Zonotope)**.** A zonotope Z is defined as:

$$Z = \left\{ c + \sum_{i=1}^{m} \alpha_i g_i \mid \alpha_i \in [-1, 1] \right\} \tag{5.37}$$

where $c \in \mathbb{R}^n$ is the center and $g_i \in \mathbb{R}^n$ are the generators.

For example, a zonotope in $\mathbb{R}^2$ with center $(0, 0)$ and generators $(1, 0)$ and $(0, 1)$ represents the square $[-1, 1] \times [-1, 1]$.

Operations on Zonotopes

The zonotope domain supports several important operations:

- **Affine transformations**: Given a zonotope Z with center c and generators $g_1, \ldots, g_m$, and an affine function $f(x) = Ax + b$:

$$f(Z) = \{Ac + b + \sum_{i=1}^{m} \alpha_i A g_i \mid \alpha_i \in [-1, 1]\} \tag{5.38}$$

- **Join**: The join of two zonotopes is approximated by a new zonotope that contains both.

- **ReLU transformation**: For ReLU functions, a sound approximation is computed by considering different cases based on whether inputs can be positive, negative, or both.

Example of Neural Network Verification with Zonotopes

Consider a simple neural network with a single hidden layer and ReLU activations, as shown in Figure 5.6.

[Image showing zonotope propagation through a neural network]

The green shapes represent zonotope abstractions at each layer

Figure 5.6: Illustration of zonotope propagation through a neural network

The verification process consists of:

1. Converting the input region (e.g., L_∞ ball) to a zonotope
2. Propagating this zonotope through the network using abstract transformers
3. Checking if the output zonotope satisfies the desired property

For example, to verify robustness, we check if the output zonotope contains points with different classifications than the original input.

5.3.2 The AI² Framework

The AI² framework (Abstract Interpretation for neural network Analysis) combines multiple abstract domains to achieve both efficiency and precision in neural network verification. It handles realistic neural networks, including convolutional networks with ReLU activations and max pooling layers.

The key components of AI² include:

- Sound abstract transformers for common neural network operations
- A flexible combination of different abstract domains
- Domain-specific optimizations for neural networks

Example 15. For a convolutional neural network trained on MNIST, AI² can verify that small perturbations (L_∞ norm ≤ 0.1) to a correctly classified digit do not change the classification.

5.4 Applications of Formal Verification for Neural Networks

5.4.1 Robustness Certification

One of the primary applications of neural network verification is to certify robustness against adversarial examples.

Definition 29 (Certified Robustness)**.** A sound robustness verifier $\langle C^\sharp, P^\sharp \rangle$ for a classifier C with respect to perturbation P certifies that for all $x \in X$, if $C^\sharp(P^\sharp(x)) = \{C(x)\}$ then $\forall x' \in P(x), C(x') = C(x)$.

This enables us to provide formal guarantees about neural network behavior under specified perturbations.

5.4.2 Safety Verification

Beyond robustness to adversarial examples, verification techniques can ensure that neural networks satisfy critical safety properties, such as:

- Ensuring autonomous vehicles maintain safe distances
- Verifying that medical diagnosis systems correctly identify critical conditions
- Checking that control systems remain within operational limits

5.4.3 Evaluating Defense Mechanisms

Formal verification can also assess the effectiveness of proposed defenses against adversarial attacks, providing mathematical guarantees rather than empirical evaluations.

5.5 Conclusion and Future Directions

This chapter has explored advanced topics in software verification, focusing on efficient fixpoint computation algorithms and neural network verification. We have seen how abstract interpretation provides a powerful framework for analyzing complex systems, including the increasingly important domain of neural networks.

As neural networks continue to be deployed in safety-critical applications, the need for formal verification techniques will only grow. Future research directions include:

- Developing more precise abstract domains tailored to neural network properties
- Improving the scalability of verification techniques to handle larger networks
- Extending verification to other types of neural networks, including recurrent and attention-based architectures
- Integrating formal verification into the neural network training process

The intersection of formal methods and machine learning presents both significant challenges and opportunities for ensuring the reliability and safety of AI systems.

5.6 Exercises

Exercise 7. Consider the following system of equations:

$$X_1 = X_2 \cup X_3 \tag{5.39}$$
$$X_2 = \{a\} \cup X_4 \tag{5.40}$$
$$X_3 = \{b\} \cup X_5 \tag{5.41}$$
$$X_4 = \{c\} \tag{5.42}$$
$$X_5 = \{d\} \cup X_1 \tag{5.43}$$

Compute the least fixpoint using (a) naive iteration, (b) chaotic iteration, and (c) the worklist algorithm. Compare the number of updates required by each algorithm.

Exercise 8. Consider a simple neural network with one hidden layer of two neurons and ReLU activations:

$$f(x_1, x_2) = w_3 \cdot \text{ReLU}(w_1 \cdot [x_1, x_2] + b_1) + w_4 \cdot \text{ReLU}(w_2 \cdot [x_1, x_2] + b_2) + b_3 \tag{5.44}$$

where $w_1 = [1, 2]$, $w_2 = [2, -1]$, $b_1 = 0$, $b_2 = 1$, $w_3 = 1$, $w_4 = 2$, and $b_3 = -1$.

Use the interval domain to verify that for inputs in the region $[0, 1] \times [0, 1]$, the output is positive.

Exercise 9. Explain why the interval domain fails to precisely capture the behavior of the network in Example 8.I. Design a more precise abstract domain that would accurately represent the relationship between the inputs to node v_3.

Exercise 10. Consider a neural network used for binary classification of images. Formalize the property that the network is robust against brightness perturbations (where all pixel values are increased by a small amount δ) and describe how you would verify this property using abstract interpretation.

Exercise 11. Research and describe a real-world application where formal verification of neural networks would be particularly important. Discuss the specific properties that would need to be verified and the challenges in applying existing verification techniques.

Chapter 6

Completeness in Abstract Interpretation

6.1 Introduction to Completeness

In the realm of software verification and static analysis, the concepts of soundness and completeness play crucial roles in determining the quality and reliability of analytical results. While soundness has traditionally been the primary focus—ensuring that analyses do not miss potential errors—completeness is equally important for eliminating false alarms and providing precise results.

Completeness in abstract interpretation refers to the property where an abstract analysis captures exactly the same information as the concrete semantics, just represented at a higher level of abstraction. When an analysis is complete, every property expressible in the abstract domain corresponds perfectly to a property in the concrete domain.

This chapter explores the theoretical foundations of completeness, its relationship with soundness, practical applications, and techniques for proving completeness for various program constructs and domains.

6.2 Soundness versus Completeness

6.2.1 Defining Soundness and Completeness

For an abstract operation to be sound, it must over-approximate the concrete operation. Formally, if α is the abstraction function, γ the concretization function, and f a concrete operation with $f^\sharp$ as its abstract counterpart, soundness is expressed as:

$$\alpha(f(x)) \sqsubseteq f^\sharp(\alpha(x)) \tag{6.1}$$

Completeness, on the other hand, requires that the abstract operation precisely captures the concrete operation:

$$\alpha(f(x)) = f^\sharp(\alpha(x)) \tag{6.2}$$

6.2.2 The Value of Completeness

The distinction between soundness and completeness directly impacts the precision of static analysis:

- **Sound over-approximations** are valuable for proving correctness, ensuring no true errors are missed.
- **Complete over-approximations** are excellent for proving that all reported alarms are genuine, eliminating false positives.

In essence, completeness provides precision. A complete analysis will never report a false alarm—if it flags a potential issue, that issue genuinely exists in the concrete semantics.

6.3 Concrete and Abstract Models

6.3.1 The Concrete Model

The concrete model represents the actual execution behavior of programs, typically expressed as sets of program states or traces. These models are generally undecidable, meaning we cannot algorithmically determine all properties of interest for arbitrary programs.

For a program P, its concrete semantics $[\![P]\!]$ represents all possible executions or states. This semantics is precise but difficult to compute or represent finitely.

6.3.2 Abstraction Approaches

Various approaches to abstraction exist, not all of which constitute abstract interpretation:

Partial Execution

Executing the program on some inputs can detect bugs but is unsound—it cannot prove the absence of bugs since it doesn't explore all execution paths.

Testing

Similar to partial execution, testing is efficient but unsound, as it cannot exhaustively cover all possible program behaviors.

Model Abstraction

Abstracting the model itself by generalizing sets of states yields $\alpha([\![P]\!])$. This approach, while potentially more efficient, remains undecidable for many properties.

Abstract Interpretation

The core of abstract interpretation is to compute an abstract semantics $[\![P]\!]^\alpha$ that approximates the concrete semantics. The fundamental relationship is:

$$\alpha([\![P]\!]) \subseteq [\![P]\!]^\alpha \tag{6.3}$$

When equality holds ($\alpha([\![P]\!]) = [\![P]\!]^\alpha$), the abstraction is complete for program P.

6.4 Theoretical Foundations of Completeness

6.4.1 Galois Connections and Best Correct Approximations

A core concept in abstract interpretation is the Galois connection between concrete and abstract domains. Given concrete domain C and abstract domain A, a Galois connection consists of an abstraction function $\alpha : C \to A$ and a concretization function $\gamma : A \to C$ such that:

$$\forall c \in C, a \in A : \alpha(c) \sqsubseteq a \iff c \sqsubseteq \gamma(a) \tag{6.4}$$

When a Galois connection exists, the best correct approximation of a concrete operation f is:

$$f^\sharp = \alpha \circ f \circ \gamma \tag{6.5}$$

6.4.2 Abstract Join Completeness

An important property of Galois connections is that abstract joins are always complete:

$$\alpha(c_1 \sqcup c_2) = \alpha(\gamma(\alpha(c_1)) \sqcup \gamma(\alpha(c_2))) = \alpha(c_1) \sqcup_\alpha \alpha(c_2) \tag{6.6}$$

This property implies that incompleteness in abstract interpretation is never due to abstract joins but comes from other operations such as tests and assignments.

6.5 Completeness Classes

6.5.1 Definition of Completeness Classes

For an abstraction α, we define the completeness classes for programs, arithmetic expressions, and Boolean expressions:

$$\mathbb{C}(\alpha) = \{P \text{ program} \mid \alpha([\![P]\!]) = [\![P]\!]^\alpha\} \tag{6.7}$$
$$\mathbb{A}(\alpha) = \{a \text{ arith.exp.} \mid \alpha([\![a]\!]) = [\![a]\!]^\alpha\} \tag{6.8}$$
$$\mathbb{B}(\alpha) = \{b \text{ Bool.exp.} \mid \alpha([\![b]\!]) = [\![b]\!]^\alpha\} \tag{6.9}$$

6.5.2 Properties of Completeness Classes

Completeness classes have several important properties:

- For trivial abstractions ($\alpha = \lambda x.x$ or $\alpha = \lambda x.\top$), $\mathbb{C}(\alpha)$ includes all programs.
- For any non-trivial abstraction, there exist incomplete programs.
- Completeness is not extensional: programs P and Q might have the same concrete semantics ($[\![P]\!] = [\![Q]\!]$), but one could be complete while the other is not.
- Both $\mathbb{C}(\alpha)$ and $\overline{\mathbb{C}(\alpha)}$ (its complement) are not recursively enumerable, making automated completeness verification challenging.

The non-extensional nature of completeness is particularly significant, as it means that program transformations that preserve concrete semantics might not preserve completeness.

6.6 Completeness Analysis for Program Constructs

6.6.1 Boolean Guards and Tests

Boolean guards are a major source of incompleteness. Even when a guard like $x > 0$ is exactly representable in an abstract domain (such as intervals), the abstract operation may not be complete:

$$\alpha_{\text{Int}}([\![x > 0]\!]\{0, 2, 3\}) = \alpha_{\text{Int}}(\{2, 3\}) = [2, 3] \subsetneq [\![x > 0]\!]^{\alpha_{\text{Int}}} \alpha_{\text{Int}}(\{0, 2, 3\}) = [\![x > 0]\!]^{\alpha_{\text{Int}}}[0, 3] = [1, 3] \tag{6.10}$$

6.6.2 Assignments

Assignments present particular challenges for completeness analysis. The completeness of an assignment depends on both the expression being assigned and the abstract domain in use.

Non-relational Domains

For non-relational domains (those that don't track relationships between variables), assignments of the form $x := a$ are complete if $a \in \mathbb{A}(\alpha)$. This rule is sound because non-relational domains consider each variable independently.

Example:

```
x := null;
if (x = null) then x := new Int;
```

In a simple nullness domain, this program is complete because both the assignment and the test are representable in the domain.

Relational Domains

For relational domains like octagons, the situation is more complex. An expression can be complete while an assignment using that expression is not.

Example: In the octagon domain, the expression $x + y$ is complete, but the assignment $z := x + y$ is not complete. This incompleteness arises because the octagon domain can express the constraint $x + y$ but cannot fully capture how this expression relates to other variables after assignment.

6.7 Proving Completeness

6.7.1 Core Proof System

A core proof system for verifying completeness can be defined with rules such as:

$$\frac{}{\vdash_\alpha \text{skip}} \quad \frac{\vdash_\alpha P \quad \vdash_\alpha Q}{\vdash_\alpha P; Q} \quad \frac{\vdash_\alpha C \quad b \in \mathbb{B}(\alpha) \quad \neg b \in \mathbb{B}(\alpha)}{\vdash_\alpha \text{if } b \text{ then } C} \quad \frac{\vdash_\alpha C \quad b \in \mathbb{B}(\alpha) \quad \neg b \in \mathbb{B}(\alpha)}{\vdash_\alpha \text{while } b \text{ do } C} \tag{6.11}$$

This system is sound (if $\vdash_\alpha P$ then $P \in \mathbb{C}(\alpha)$) but not complete (there exist programs in $\mathbb{C}(\alpha)$ for which $\vdash_\alpha P$ cannot be derived).

6.7.2 Challenges in Automating Completeness Proofs

Automating completeness proofs is challenging because:

- Completeness classes are not recursively enumerable.
- Completeness is harder to prove than termination.
- Domain-specific analyses are needed, especially for assignments in relational domains.

Despite these challenges, certain under-approximations of completeness classes can be computed, providing partial verification of completeness.

6.8 Examples of Completeness Analysis

6.8.1 Complete and Incomplete Loop Examples

Consider two similar programs:

Complete:

```
x := 9;
while (x > 0)
    x := x - 1;
// query: x = 0?
```

Incomplete:

```
x := 9;
while (x > 0)
    x := x - 2;
// query: x = -1?
```

Both programs appear similar, and each individual operation seems complete for the interval domain:

- Assignment to a constant is complete in intervals
- Both tests $x > 0$ and $x \leq 0$ are exactly represented in intervals
- The decrements $x - 1$ and $x - 2$ are complete in intervals
- Abstract join is always complete

However, the second program is incomplete due to the interaction between operations, specifically how the test interacts with the state transformations across loop iterations.

6.8.2 Relational Domain Example

The octagon domain example illustrates the challenges with relational domains:

$$S = \{(x/2, y/1, z/0), (x/1, y/4, z/2)\} \tag{6.12}$$

$$\text{Oct}([\![z := x + y]\!]S) \subsetneq \text{Oct}([\![z := x + y]\!]\text{Oct}(S)) \tag{6.13}$$

A specific element $(x/2, y/3, z/5)$ belongs to $[\![z := x + y]\!]\text{Oct}(S)$ but not to $\text{Oct}([\![z := x + y]\!]S)$, demonstrating incompleteness.

6.9 Conclusion

Completeness in abstract interpretation provides a powerful framework for verifying the precision of static analyses. While challenging to achieve and prove, complete abstractions offer significant benefits by eliminating false alarms and providing exact information within the constraints of the abstract domain.

Understanding when and why completeness fails helps analysts design better abstractions and verification systems. Though completeness classes are generally not computable, domain-specific techniques and proof systems can help establish completeness for important program fragments and properties.

Future research in this area may focus on developing more powerful proof techniques, identifying broader classes of programs for which completeness can be guaranteed, and creating domain-specific refinements that achieve completeness for critical properties.

Bibliography

[1] Winskel, G. (1993). *The Formal Semantics of Programming Languages: An Introduction*. MIT Press.

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

[3] Nielson, F., Nielson, H.R., & Hankin, C. (1999). *Principles of Program Analysis*. Springer.

[4] Muchnick, S.S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

[5] Miné, A. (2006). The octagon abstract domain. *Higher-order and symbolic computation*, 19(1), 31-100.

[6] Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., & Stata, R. (2002). Extended static checking for Java. *ACM SIGPLAN Notices*, 37(5), 234-245.

[7] Hoare, C.A.R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 576-580.

[8] Fähndrich, M., & Logozzo, F. (2010). Static contract checking with abstract interpretation. In *Formal Verification of Object-Oriented Software* (pp. 10-30). Springer.

[9] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., & Rival, X. (2005). The ASTRÉE analyzer. In *Programming Languages and Systems* (pp. 21-30). Springer.

[10] Kildall, G.A. (1973). A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (pp. 194-206).

[11] Winskel, G. (1993). *The Formal Semantics of Programming Languages: An Introduction*. MIT Press.

[12] Nielson, F., Nielson, H.R., & Hankin, C. (1999). *Principles of Program Analysis*. Springer.

[13] Schmidt, D.A. (1986). *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon.

[14] Stoy, J.E. (1977). *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press.