Università
Ca'Foscari
Venezia

Dipartimento di Scienze Ambientali, Informatica e Statistica

Master's Degree in Computer Science

Stability Abstract Domain: determining Covariance or Contra-variance trough Static Analysis

MASTER DEGREE THESIS

Supervisor
Ch. Prof. Agostino Cortesi

Graduand
Sofia Presotto
870762

Academic Year
2023/2024

Contents

1 Introduction

The increasing complexity of software systems necessitates robust methods for ensuring their reliability and security. *Anomaly detection* consists in the analysis of program behaviors in order to determine whether aberrant executions states can occur.

In this thesis, we propose a *static* and *formal* contribution to anomaly detection, based on *abstract interpretation* theory. The main novelty is the formal definition and complete implementation of the *Stability* domain for anomaly detection, noticing that no other domain in the literature already captures this property, as far as we know.

Our solution is built on the concept of covariance and contra-variance of variables as relevant properties in the identification of anomalous states.

When operating a vehicle, for example, we expect that if the left turn signal is activated, then the turning wheel will rotate to the left. The two conditions are independent from one another, as it is possible to signal a right turn and then turn left, but from our understanding this is clearly an anomaly.

In a program, we may have that variables syntactically independent from one-another are expected to behave as correlated variables, either increasing and decreasing together, or maintaining opposite trends. If it is possible to identify such variables, then any state in which the property is not satisfied may correspond to an anomalous state of execution. On the other hand, a static analyser providing a proof that covariance or contra-variance is verified throughout the program or in critical code blocks, is providing a proof of correct behavior.

The key idea for expressing covariance and contra-variance is the notion of stability as the detection of monotonicity of a variable in the sequence of states of any tree of execution for a program. From stability information of a set of variables it is possible to determine their correlation.

The *Stability* domain is fully implemented in the LiSA generic static analyser developed by the SSV research group at Ca' Foscari University and has been tested on a generic imperative language, though we note it can already be applied to any other programming language supported by LiSA (namely Python and Go, at the time of writing).

The effectiveness of the proposal is discussed by considering in detail a representative use case.

Background The background section lays the foundational concepts necessary for understanding the thesis, introducing static analysis, abstract interpretation and anomaly detection in the context of program verification. It also discusses order theory to provide the mathematical framework underlying the abstraction of program properties.

The chapter concludes with an introduction to the LiSA static analysis framework and an overview of components, interfaces and classes necessary for implementing and running an analysis.

The Stability Domain Chapter 3 introduces the *Stability* abstract domain. It first defines concepts of increasing, decreasing and stable variables and discusses how these properties can be inferred from semantics and auxiliary information. Then, the chapter provides the formal definition of the domain with its concretization and abstraction functions, mapping variables to their trend in correspondence of a statement.

The last section shows the LiSA implementation of *Stability* through the `Trend` and `Stability` classes, as well as the results obtained from running the analysis on a method.

Covariance and Contra-variance This chapter explores the concepts of covariance and contra-variance, showing how they can be determined from the results of the stability analysis and providing their formal definitions.

Then it discusses the implementation of these concepts in LiSA as a `SemanticCheck`.

Study: the Scale Problem Chapter 5 illustrates a practical application of the correlation analysis, demonstrating the results that can be reached on a program computing the scale transformation of rectangles. This example provides a clear understanding of how covariance and contra-variance can influence software analysis and the benefits they offer when integrated into the LiSA framework.

Related Work This chapter provides an overview of existing research and developments in the fields of abstract interpretation and anomaly detection, contextualising this thesis within the broader landscape and highlighting significant contributions from previous studies.

Appendices The appendices include additional technical details and support the implementation discussions from the main chapters. Appendix A presents the logic of operations discussed in Section 3.2.1. Appendix B covers the `Trend` class, Appendix C the `Stability` class, and Appendix D the `CoContravarianceCheck` class. Finally, Appendix E presents two examples of methods running the stability and correlation analyses.

2 Background

This chapter provides the foundation for understanding the concepts discussed in this thesis.

It introduces static analysis as a method for automated analysis of program source code independently from execution, outlining its main features and limitations. It delves into abstract interpretation theory, explaining key concepts like *concrete* and *abstract* semantics, properties, and domains. Order theory is introduced to provide the mathematical foundations necessary to model these concepts, namely *partially ordered sets*, *lattices*, *Galois connections*, and *fixpoint abstraction*.

Anomaly detection is discussed as a key application scenario, highlighting the relevance of a formal and static approach.

Finally, the chapter introduces LiSA, a Java framework for the development of abstract interpretation static analyzers, presenting its main components and providing practical guidance on its use.

2.1 Static Analysis

Static analysis is an automatic technique to verify that a program satisfies some semantic property, *semantic* meaning it pertains to the run-time behavior of a program [1]. It belongs to the class of formal methods, mathematical techniques used in program analysis for the verification of software systems. They provide a formal framework to reason about the correctness, behavior, and properties of programs. By symbolically examining the entire state space, formal methods can establish correctness or safety properties that apply to all possible inputs of the analyzed program.

Static analysis is automated through specialised tools, called static analyzers. These are programs that take another program as input and produce information about its behavior independently from execution, as the word *static* suggests.

A static approach, while being typically harder to design and implement than a dynamic one, has the key advantage of allowing for the detection and removal of issues before the program runtime. Moreover, dynamic analyses often introduce performance overhead and, as pointed out by [1], certain properties, such as termination, cannot be verified dynamically.

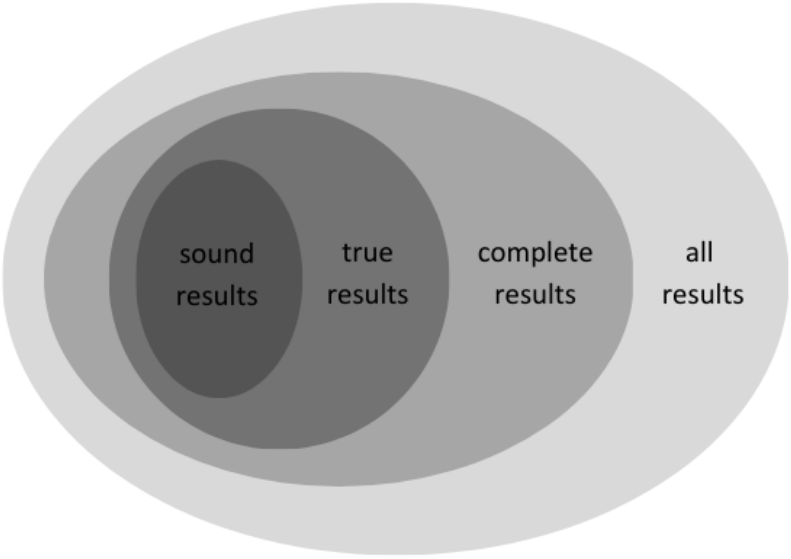

FIGURE 2.1: Venn representation of sound and complete analysis results.

2.1.1 Sound and Complete analyses

The two following properties are critical in order to understand the results of an analysis.

Definition 2.1. (Soundness) The analysis is *sound* with respect to property P whenever, for any program p, $analysis(p) = true$ implies that p satisfies property P.

Soundness means that it is possible to have a program which does satisfy P and for which the analysis returns *false*, but there will never be a program for which the result is *true* and P does not hold. A sound analysis will only accept programs for which it can guarantee that they satisfy the property, ensuring that all programs that do not satisfy P are rejected.

Definition 2.2. (Completeness) The analysis is *complete* with respect to property P whenever, for every program p such that p satisfies P, $analysis(p) = true$.

Therefore, completeness means it is possible to encounter a program which does not satisfy P and for which the analysis returns *true*, but no program for which P holds and the analysis returns *false*. Complete analyses will accept every program that satisfies P and ensure that all programs which are rejected do not satisfy P.

A *sound* analysis may be interpreted as over-approximating the behavior of the program, while a *complete* analysis as underapproximating it, as exemplified by Figure 2.1.

In practice, any alarm raised by a complete analysis (by returning *false*) corresponds to an actual violation of the property. On the other hand, an alarm reported by a sound analysis may be a false alarm and the property might actually hold. However, if a sound program analysis reports no errors, then the program is guaranteed to satisfy the property being verified.

Static analysis is sound but not complete, the reason for which is discussed in more detail in the following section.

2.1.2 Limitations of Static Analysis

As proved by H.G. Rice, all non-trivial properties of program behaviour in a Turing-complete programming language are undecidable [2]. *Non-trivial* properties, i.e., "there are programs that satisfy [them] and programs that do not" [1], are exactly what is necessary to verify.

A consequence of Rice's Theorem is that there can be no verification method that is automated, sound and complete. Therefore, any possible analysis presents some kind of limitation, either by giving up automation, by targeting only a restricted class of programs, or by not always being able to provide an exact answer (i.e., giving up soundness or completeness) [1].

Testing, for example, is complete but not sound: as it only observes a finite set of executions, it cannot guarantee that all programs that do not satisfy the property will be identified.

Static analysis, on the other hand, is automated and the soundness of the results can be ensured by the use of *abstract interpretation theory*, as discussed in Section 2.2. The price to be payed is the possible loss of accuracy. In fact, as static analysis employs algorithms that enforce termination of the analysis even when the program may have infinite executions, the results are not complete, as we may have 'false alarms' [1].

2.2 Abstract Interpretation

Abstract interpretation [3] is a semantics approximation theory which provides formal tools for the sound abstraction of a program's source code into a set a properties.

The central idea is to construct two different meanings of a programming language: one, the *concrete semantics*, gives the usual meaning of programs in the language, and the other, the *abstract semantics*, can be used to verify properties of programs in the language (i.e., answer certain questions about runtime behavior). The abstract semantics produced through abstract interpretation denote a *sound over-approximation* of the concrete semantics, ensuring that any property proven true for the program on an abstract domain does indeed hold for the original concrete domain (soundness). The mathematical foundation for this is provided by *order theory* and is presented in Section 2.3. The abstraction also provides scalability, by abstracting away irrelevant details and making it feasible to analyze complex programs.

Then, given an abstraction of the program, the analysis applies an iterative fixed-point algorithm to determine a superset of states which verify a property. The intuition is that properties of the program "flow" through the statements, with each statement adding new information and "killing" information that is no longer true. An algorithm analysing the program requires transfer functions, describing how information changes as it flows through a basic block (i.e., what information is added and eliminated), and merge functions, describing how information from multiple pats is combined. As discussed in section 2.3.2, in this case the domain itself describes both transfer and *meet* functions.

2.2.1 Abstract domains

An *abstract domain* is a set of *abstract properties*, which represent partial information about program states or about execution [1].

This domain encodes weaker properties and operations than the concrete domain it abstracts [4]. As a result, analysis on the abstract domain over-approximates program behavior, generating a superset of reachable states. By manipulating these abstract logical properties, the analysis can infer sound conclusions about all possible program executions.

Example 2.1. The *Sign* abstract domain is a set of properties representing the sign of program variables at a program state. We can consider the abstract properties: *"is positive"* ('+'), *"is negative"* ('−'), *"is zero"* ('0'), *"is non negative"* ('$+^0$'), *"is non positive"* ('$-^0$'), *"is different from zero"* ('$\neq 0$') (i.e., either positive or negative); as well as: *"unknown sign"* ('$\top$') (i.e., either positive, negative, or zero) and *"no sign"* ('$\bot$'), the meaning for which is discussed later on.

Suppose, for some state of a program, $\{x \mapsto 2\}$ in the concrete domain. The abstract value of variable x in the *Sign* domain is *"is positive"*, meaning that for that same state $\{x \mapsto \text{'}> 0\text{'}\}$ in the *Sign* domain. Note that abstract property P'="*x is positive*" is indeed weaker than property P = "$x = 2$", since P implies P'.

The relation between the concrete and abstract domains is formalized by a *concretization* function, defined along with the abstract domain, which maps abstract properties to the set of concrete elements that satisfy them. An *abstraction* function operates the mapping of concrete properties into the ones abstracting them in the abstract domain. The relation between these two functions, when respecting criteria discussed in Section 2.3.3, can equate to a proof of the soundness of the results of the analysis.

Abstract interpretation provides the mathematical tools for constructing an abstract domain for which these properties hold. Section 2.3 goes into detail of the mathematical structures representing domains and their relation.

2.3 Order Theory

Order theory provides the formal mathematical framework for representing programs and properties, necessary for abstract interpretation. It is a branch of mathematics that deals with the study of *partially ordered sets* (*posets* for short), which are sets equipped with a *partial order*, i.e., a binary relation that is reflexive, antisymmetric, and transitive [4]. A particular case of poset is the *lattice* structure, which in abstract interpretation can be used to represent abstract domains, with its elements being abstract properties. Order theory also provides the mathematical foundation for understanding fixpoint computations, proving their correctness, and ensuring their convergence properties.

This section presents the formal definitions of order theory structures relevant to abstract interpretation theory.

2.3.1 Posets

Definition 2.3. (Poset) A *poset* $\langle \mathbb{P}, \sqsubseteq \rangle$ is a set $\mathbb{P}$ equipped with a partial order $\sqsubseteq$ that is:

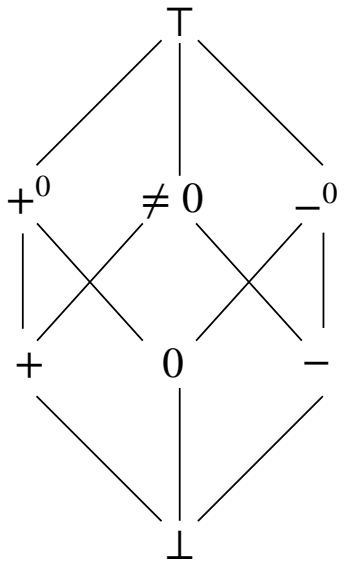

FIGURE 2.2: Hasse diagram of the Sign abstract domain.

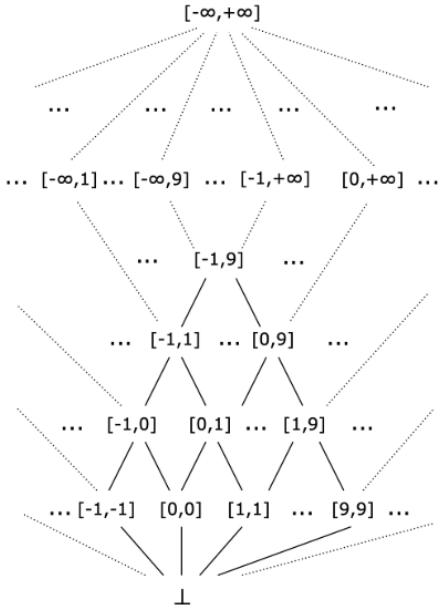

FIGURE 2.3: Hasse diagram of the lattice of intervals

- *Reflexive*: $\forall x \in \mathbb{P},\ x \sqsubseteq x$;
- *Antisymmetric*: $\forall x, y \in \mathbb{P},\ ((x \sqsubseteq y) \wedge (y \sqsubseteq x)) \Rightarrow (x = y)$;
- *Transitive*: $\forall x, y, z \in \mathbb{P},\ ((x \sqsubseteq y) \wedge (y \sqsubseteq z)) \Rightarrow (x \sqsubseteq z)$.

Two elements x and y are said to be *comparable* when either $x \sqsubseteq y$ or $y \sqsubseteq x$, and *incomparable* when neither $x \sqsubseteq y$ nor $y \sqsubseteq x$. A partially ordered set can contain both comparable and incomparable elements, providing a way to compare elements within the set without necessarily requiring every pair of elements to be comparable.

Posets $\langle \mathbb{P}, \sqsubseteq \rangle$ can be represented by *Hasse diagrams*, as shown in Figures 2.2 and 2.3. In a Hasse diagram, a node x is below node y if $x \sqsubset y$, and a segment connects x and y if $x \sqsubset y$ and there exists no other node z *s.t.* $x \sqsubset z \wedge z \sqsubset y$ (i.e., we exclude transitive edges, as well as reflexive edges). The edges have no direction, but the position of the two connected nodes allows to infer the orientation.

An abstract domain can be represented by a poset $\langle \mathbb{A}, \sqsubseteq \rangle$, where elements of $\mathbb{A}$ are abstract properties and $P \sqsubseteq P'$ indicates that property P' is weaker than property P, i.e., P implies P'. For example, for the *Sign* poset from Figure 2.2, property *"is positive"* ('+') implies property *"is non negative"* ('$+^0$'), meaning '$+^0$' is the weaker, more general property and '+' $\sqsubseteq$ '$+^0$'. For the *Interval* domain from Figure 2.3, property *"is between 0 and 1"* ('[0,1]') implies property *"is between -1 and 1"* ('[-1,1]'), so '[0,1]' $\sqsubseteq$ '[-1,1]'.

2.3.2 Lattices

Lattice structures are often used in abstract interpretation to represent abstract domains. A lattice is a poset for which any pair of elements has a *least upper bound* and a *greatest lower bound*.

Definition 2.4. (Upper (lower) bound) Given S subset of $\langle \mathbb{P}, \sqsubseteq \rangle$, an *upper (lower) bound*, if it exists, is an element $b \in \mathbb{P}$ such that for all $x \in S$, $x \sqsubseteq b$ ($b \sqsubseteq x$).

Definition 2.5. (Least upper bound) The *least upper bound* (*lub* or *join*) for S, indicated with $\sqcup S$, is the smallest among the upper bounds of S.

Definition 2.6. (Greatest lower bound) The *greatest lower bound* (*glb* or *meet*) for S, indicated with $\sqcap S$, is the greatest among lower bounds of S.

Note that *lub* and *glb* are not guaranteed to exist in every poset; however, when they do exist, they are unique.

Example 2.2. Consider again the *Sign* poset of properties from Figure 2.2. Subset $S = \{\text{'+'}, \text{'0'}\}$ has upper bounds $\{\text{'}+^0\text{'}, \text{'}\top\text{'}\}$, since '+' $\sqsubset$ '$+^0$' and '0' $\sqsubset$ '$+^0$', and '+' $\sqsubset$ '$\top$' and '0' $\sqsubset$ '$\top$'. The *lub* (or *join*) of S is '$+^0$', because '$+^0$' $\sqsubset$'$\top$'. Consider the *Interval* lattice of properties from Figure 2.3. Subset {'[-1,1]', '[0,9]'} has infinite upper bounds {'[-1,9]', '[-2,9]', '[-1,10]'...}. The *lub* is '[-1,9]', because it is the least in this infinite set.

Definition 2.7. (Lattice): *Lattices* are posets in the form $\langle \mathbb{L}, \sqsubseteq, \sqcup, \sqcap \rangle$, with set $\mathbb{L}$ equipped with partial order $\sqsubseteq$ and *lub* and *glb* operators $\sqcup$ and $\sqcap$, with the following property:

$$\forall x, y \in \mathbb{L},\ x \sqcup y \text{ and } x \sqcap y \text{ exist in } \mathbb{L}.$$

A lattice is *complete* if any subset S of $\mathbb{L}$ has a *lub* and a *glb* (not only the finite ones). A complete lattice has a *top element* $\top = \sqcup\mathbb{L}$ (*join* of all elements of $\mathbb{L}$) and an *bottom element* $\bot = \sqcap\mathbb{L}$ (*meet* of all elements of $\mathbb{L}$), and is denoted by $\langle \mathbb{L}, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$. The *Sign* domain from Figure 2.2 and the *Interval* domain from Figure 2.3 are complete lattices.

In the context of abstract interpretation, the *lub* operator corresponds to the merge function for converging branches, introduced in Section 2.2. This means that during the analysis, when encountering a statement with more than one predecessor, its entry state will be the *lub* of the exit states of its predecessors. If the abstract domain is a complete lattice, we are ensured this *lub* exists.

2.3.3 Galois connections

The relation required between concrete and abstract domains is the *concretization function* (γ), expressing the meaning of abstract properties in terms of concrete properties. An abstract property P' is a sound over-approximation of a concrete property P whenever $P \sqsubseteq \gamma(P')$.

The *abstraction function* (α), instead, assigns the corresponding abstractions to concrete properties (e.g., $\alpha(P) = P'$).

Definition 2.8. (Galois connection) Given posets $\langle \mathbb{C}, \leq \rangle$ (the concrete domain) and $\langle \mathbb{A}, \sqsubseteq \rangle$ (the abstract domain), the pair $\langle \alpha, \gamma \rangle$ of abstraction function $\alpha \in \mathbb{C} \to \mathbb{A}$ and concretization function $\gamma \in \mathbb{A} \to \mathbb{C}$ is a *Galois connection* iff:

$$\forall P \in \mathbb{C}\ , \forall P' \in \mathbb{A}\ , \alpha(P) \sqsubseteq P' \iff P \leq \gamma(P')$$

A Galois connection establishes a correspondence between concrete and abstract domains, enabling the transfer of properties and operations between

them and formalizing the correspondence between concrete and abstract properties.

Under the assumption that $\langle \mathbb{C}, \leq \rangle$ and $\langle \mathbb{A}, \sqsubseteq \rangle$ form a Galois connection (with abstraction function α and concretization function γ) the following properties are true:

- α and γ are monotone functions, meaning they map logically comparable inputs into logically comparable outputs (e.g., if $P_1 \leq P_2$ then $\alpha(P_1) \sqsubseteq \alpha(P_2)$ and if $P'_1 \sqsubseteq P'_2$ then $\gamma(P'_1) \leq \gamma(P'_2)$);
- applying the abstraction function to a concrete property and then applying the concretization function to the result yields back a less precise property than the original one (i.e., $\forall P \in \mathbb{C}, P \leq \gamma(\alpha(P))$);
- applying the concretization function to an abstract property and then abstracting the result refines the information available in the initial abstract element, resulting in a more precise property (i.e., $\forall P' \in \mathbb{A}, P' \sqsubseteq \alpha(\gamma(P'))$).

These properties of Galois connections are the base for ensuring soundness, guaranteeing that analyses over abstract domains yield correct results relative to the concrete semantics of programs. For the purposes of the present work, it is sufficient to note that if a concrete domain and the corresponding abstract domain form a Galois connection, then the results of the analysis are sound.

2.3.4 Fixpoint Abstraction

Fixpoint abstraction is essential for defining the semantics of programs, in particular for enforcing termination of the analysis algorithms. Programs often contain loops or recursive functions, where the behavior depends on the accumulation of effects over multiple iterations. The analysis in such cases requires to determine the eventual outcome of these iterations, which can be seen as reaching a fixpoint where further iterations do not change the state. Fixpoint abstraction consists in computing these fixpoints in the abstract domain, ensuring that the fixpoint computation is always terminating.

To ensure termination, the algorithm applies a *widening* operator, which accelerates the convergence to a fixpoint by over-approximating the abstract values, preventing infinite loops in the analysis.

In this section we provide definitions for the main concepts of fixpoint computation for abstract interpretation.

Definition 2.9. (Fixpoint) Consider monotone function $f : S \to S$ on a partial order $\mathbb{P}$. An element x of S is a *fixpoint* of f if $f(x) = x$.

In the context of abstract interpretation, such fixpoints represent stable states in the program's execution where certain properties no longer change.

If S is a complete lattice, then the set of its fixpoints $fp(f) \subseteq S$ is also a complete lattice, with:

- a *least fixpoint*: $lfp(f) = glb(fp(f))$;
- a *greatest fixpoint*: $gfp(f) = lub(fp(f))$;

with $lfp(f) \in fp(f)$ and $gfp(f) \in fp(f)$.

Definition 2.10. (Widening) A binary operator $\nabla : \mathbb{A} \times \mathbb{A} \to \mathbb{A}$ is a *widening operator* in abstract domain $\langle \mathbb{A}, \sqsubseteq \rangle$ if it:

1. Abstracts the *lub* operator: for abstract elements a_0, a_1, $\gamma(a_0) \cup \gamma(a_1) \subseteq \gamma(a_0 \nabla a_1)$.
2. Enforces convergence: for ascending chain $(a_i)_{i \in \mathbb{N}}$ in $\mathbb{A}$, the corresponding sequence $(a_i^*)_{i \in \mathbb{N}}$ computed as: $\begin{cases} a_0^* = a_0 \\ a_{n+1}^* = a_n^* \nabla a_{n+1} \end{cases}$ is ultimately stationary.

The analysis is able to enforce termination by detecting lattice values that may be part of infinitely ascending chain and then artificially raising the value of the analysis to an approximation of the *lub* of the chain (i.e., $a_n^* \nabla a_{n+1}$), abstracting the reaching of a fixpoint.

2.4 Anomaly Detection

Anomaly detection consists in the analysis of program behaviors in order to determine whether aberrant executions states caused by attacks, misconfigurations, program bugs, and unusual usage patterns can occur. Anomalous execution states can be caused by program bugs, inappropriate program logic, or insecure system designs and can lead to malfunctions, as well as leave software open to subversion or exploitation by malicious users [5].

Practical applications often consist in dynamically monitoring the program execution and comparing it with some model of the normal system-call behavior in order to detect anomalous system calls. This can be more or less effective depending on the application, on the user and on the attacker. Nevertheless, since the underlining issue is not addressed, the defences can be circumvented and the program exploited [5]. Added to this, is the significant tracing overhead, as well as the issue of safely modeling normal behavior [6].

A static and formal approach to anomaly detection, instead, would have the benefits of addressing the core issue, identifying potentially problematic patterns, structures, or behaviors before the deployment of software, as well as removing the necessity to model normal behavior and the overhead due to the real-time tracking of program execution.

The focus of this thesis is the development of an abstract domain for tracking the trends of variables trough a program. This information is applied for anomaly detection based on the idea that, for some programs, certain variables are covariant (or contra-variant) for any correct trace of execution. Then, it is possible to run a static analysis on a program's source code in order to determine trends of variables, and, based on this information, infer their covariance or contra-variance. Then it is possible to verify whether a property in the form *"x and y are covariant (or contra-variant)"* holds.

Note that, for the results of this analysis to be relevant, we need a formal guarantee of the covariance or contra-variance of these variables. Put simply, the automatic analyser may determine if we can be sure that some variables have always the same (or opposite) trend within the code, but it is up to the user running the analysis to determine (formally) which variables must be covariant or contra-variant in order for the program's behavior to be correct (i.e., free of anomalies).

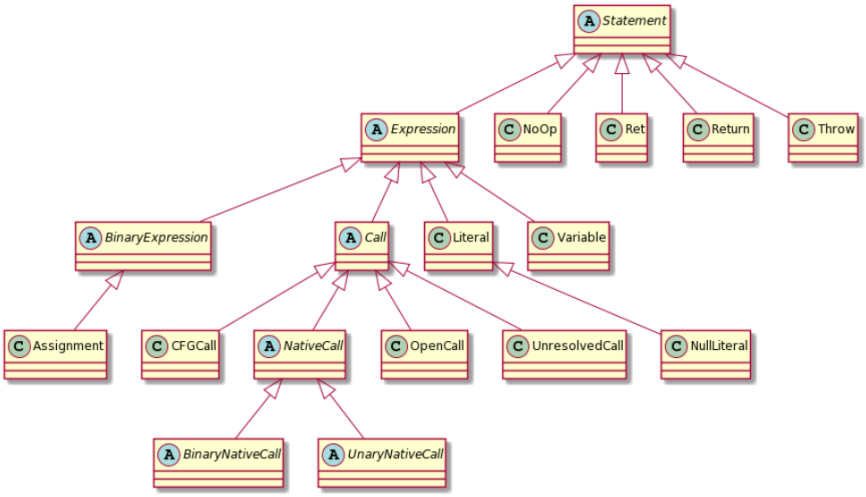

FIGURE 2.4: LiSA `Statement` class hierarchy

2.5 LiSA

LiSA (Library for Static Analysis) is a Java framework designed for the creation and implementation of static analyzers based on abstract interpretation for imperative languages. The library implements the infrastructure necessary, allowing users to focus mainly on the design of abstract properties and on the sound approximation of code semantics [7].

LiSA translates input programs into an internal control-flow graph (*CFG*, for short) representation. Then, it runs a twofold analysis: first an heap abstraction, then a value abstraction. The result of the analysis is an entry and an exit abstract state for each node in the CFG.

A domain is able to determine how abstract information evolves thanks to the semantics of statements. In LiSA, implementing a new abstract domain equates to defining a value domain, which must provide the logic of lattice operations and operations for reasoning on statement semantics, while the infrastructure responsible for mapping variables to abstract values is encapsulated by the concept of *environment*.

This section only provides a superficial understanding of the library, presenting an overview of LiSA features and classes relevant to the discussion in the following chapters.

2.5.1 Front-end component and CFG representation

An input program is translated into a set of CFGs, one for each program method, by a language-specific front-end component. The main language used for testing in LiSA is IMP, an imperative language handling arithmetic expressions inspired by Java. The examples presented in Sections 3.4.3 and 5.3 take as input IMP programs, which are processed by the `IMPFrontend` class.

The CFG structure offered by LiSA is flexible and extensible, capable of representing the syntax of any programming language by directly encoding control flow structures in its own structure. Single nodes of the CFG represent single statements which directly affect the program state, expressed as sets of symbolic expressions, written in the internal language of LiSA.

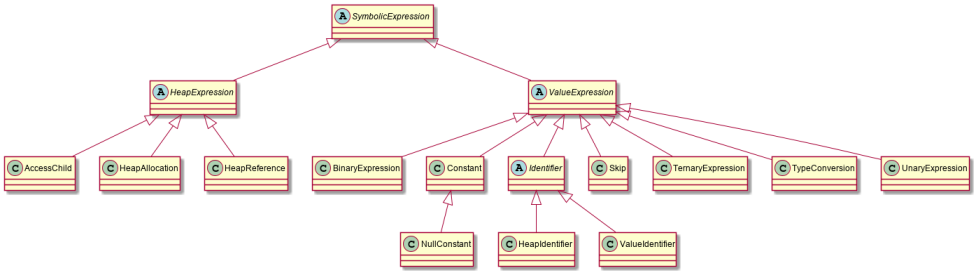

FIGURE 2.5: LiSA `SymbolicExpression` class hierarchy

The `CFG` class is the LiSA implementation of CFGs, where edges are represented by the `Edge` class, while nodes by the `Statement` class, whose hierarchy is shown in Figure 2.4.

2.5.2 `SymbolicExpression` and `Statement` classes

`SymbolicExpression`s are a set of expressions which represent the semantics of CFG nodes, written in the internal language of LiSA and defined over a small set of operations. They are separated into `HeapExpression`s, modeling operations on the heap, and `ValueExpression`s, that instead concern only constant values and identifiers. Figure 2.5 shows the `SymbolicExpression`'s class hierarchy. Abstract domains reason on semantics and therefore their implementations in LiSA operate on these `SymbolicExpression`s.

Nodes of LiSA CFGs, which are represented by `Statement`s, must rewrite themselves into `SymbolicExpression`s that represent their semantics, that are then passed on to abstract domains.

Note that even though function calls are `Statement`s, they are not included as `SymbolicExpression`s. This is because CFG calls are handled by the `CallGraph` interface, which resolves the calls and provides their results in the abstract domain, allowing the rest of the analysis infrastructure to abstract away the calls themselves.

2.5.3 `Lattice` and `SemanticDomain` interfaces

In LiSA, lattice operations and semantic operations are handled by two separate interfaces: `Lattice` and `SemanticDomain`, respectively.

The generic `Lattice` interface represents elements of a lattice. It is parametric to the concrete instance `L`, which must implement bottom and top elements, *lub*, *glb* and widening operations, and the partial order. For example, the `lub()` method will be called by the analysis whenever encountering two converging branches of the CFG to compute the entry state of their successor node.

LiSA provides the `BaseLattice` class, which overrides these lattice operations and handles their base cases, delegating the implementation of the specific logic to auxiliary methods that must be implemented by its subclasses. Take, for example, the `lub()` method implementation of `BaseLattice`:

```
1 default L lub(L other) throws SemanticException {
2     if (other == null || other.isBottom() || this.isTop()
3             || this == other || this.equals(other))
4         return (L) this;
5
```

```
6       if (this.isBottom() || other.isTop())
7           return other;
8
9       return lubAux(other);
10  }
```

The `SemanticDomain` interface represents domains capable of reasoning about semantics of symbolic expressions. It is parametric on concrete instance `D`, symbolic expression type `E` and identifier type `I`. An implementation of `D` must implement methods:

- `D assign(I id, E exp)`: yields a copy of the domain where `id` has been assigned to the abstract value obtained from the evaluation of `exp`;
- `D assume(E exp)`: yields a copy of a domain modified by assuming that `exp` holds;
- `D forgetIdentifier(I id)`: forgets all gathered information about identifier `id`;
- `D forgetIdentifiers(Collection<I> ids)`: forgets all information about all identifiers `ids`;
- `Satisfiability satisfies(E exp)`: returns a `Satisfiability` object (`SAT`, `UNSAT`, `UNKNOWN` or `BOTTOM`) representing whether `exp` is satisfied in the program state represented by the domain;
- `D smallStepSemantics(E exp)`: yields a copy of a domain modified accordingly to the semantics of `exp`.

2.5.4 `ValueDomain` and `HeapDomain` interfaces

`ValueDomain` and `HeapDomain` are two interfaces, each parametric on its own concrete type (`V` and `H`, respectively), that extend both `Lattice` (`Lattice<V>` and `Lattice<H>`, respectively) and `SemanticDomain`. Their difference comes down to the parameters of this last interface: `ValueDomain` extends `SemanticDomain<V,ValueExpression,Identifier>`, while `HeapDomain` extends `SemanticDomain<H,SymbolicExpression,Identifier>`. `ValueDomain` represents the value abstraction and `HeapDomain` the heap abstraction.

2.5.5 `ValueEnvironment` interface

The `ValueEnvironment` interface is parametric on type `D`, implements `ValueDomain<ValueEnvironment<D>` and represents the mapping of `Identifiers` (i.e., individual variables) to instances of `D` (i.e., abstract values).

Type `D` must implement `NonRelationalValueDomain`, an interface modeling a non-relational value domain capable of computing the value of a `ValueExpression` from the values of all program variables and to check if an expression is satisfied.

`BaseNonRelationalValueDomain` is an abstract class provided by LiSA that presents a base implementation for non-relational value domains.

2.5.6 `AbstractState` class

A LiSA abstract state, represented in the `AbstractState` class, wraps together a `ValueDomain` and an `HeapDomain`. `AbstractState` is parametric on the type `H` of `HeapDomain`, the type `V` of `ValueDomain` and on the type `T` of `TypeDomain`. The `TypeDomain` interface, parametric on its own concrete type `T`, extends `ValueDomain<T>` and represent a domain for handling dynamic typing inferences in the analysis. `AbstractState` also implements `Lattice<AbstractState<H,V,T>` and `SemanticDomain<AbstractState<H, V,T>,SymbolicExpression,Identifier>`.

Any expression to be evaluated is first passed to the `HeapDomain`, which updates itself according to its semantics and rewrites it into an expression free of heap references. Then, the rewritten expression is passed to the `ValueDomain` for its evaluation. This separation between domains ensures a high degree of modularity, as all components of the analysis can be modified or replaced without the need for any modifications to other components. This also means that the implementation of a new domain can be carried out without any knowledge of heap access semantics, for which the standard implementations provided by LiSA (e.g., `MonolithicHeap`, `FieldSensitivePointBasedHeap`) can be applied, simplifying the development of analysers.

2.5.7 Running LiSA

After the definition of an appropriate `ValueDomain`, the analysis is run by creating a `LiSA` object, configuring it, and, finally, calling the `run()` method on a `Program` object. The following line of code defines a `Program` by invoking the IMP front-end on the file located at "filePath".

```
Program program =
    IMPFrontend.processFile("filePath");
```

One possible way of configuring the `LiSA` object is by passing directly to the constructor a `LiSAConfiguration` object. The following code shows an example:

```
1  // specify directory for generated files
2  conf.workdir = "output/sign";
3
4  // specify the visual format of the analysis results
5  conf.analysisGraphs = LiSAConfiguration.GraphType.HTML;
6
7  // indicate to produce JSON files with the serialised results
8  conf.serializeResults = true;
9
10 // specify the analysis that we want to execute
11 conf.abstractState = new SimpleAbstractState<>(
12         // heap domain
13         new MonolithicHeap(),
14         // value domain
15         new ValueEnvironment<>(new Sign()),
16         // type domain
17         new TypeEnvironment<>(new InferredTypes()));
18
19 conf.interproceduralAnalysis = new ContextBasedAnalysis<>();
20
21 // instantiate LiSA with the configuration
```

```
22 LiSA lisa = new LiSA(conf);
23
24 // analyze the program
25 lisa.run(program);
```

As shown by the code above, the different fields of a `LiSAConfiguration` are set to the desired implementations to be used in the analysis.

The `stabiltiyTest()` method from Appendix E shows a test running LiSA with the *Stability* domain implementation, `Stability`.

3 The Stability Domain

This chapter presents the *Stability* domain, a novel abstract domain specifically designed to track variations of the trends of variables within software programs. This analysis is crucial for identifying variables that should remain constant or follow specific trends. By tracking how variables change over time, it is possible to detect patterns that may indicate potential issues or to prove properties that ensure correct behavior. In the scope of this thesis, as explored in Chapter 4, *Stability* information aims to aid in the detection of potential anomalies by identifying correlation between syntactically independent variables as correlation of their trends.

Overall, this section provides a comprehensive examination of the *Stability* domain, from the intuitions behind its development and its proposed definition, to its implementation and application.

We discuss how stability information can be inferred from code statements' semantics and refined by interrogating an auxiliary abstract domain, with accuracy depending on the chosen domain.

Next, we define the *Stb* lattice to express variables' possible trends as abstract properties and their relations. We then provide the formal definition of the *Stability* abstract domain as the STB lattice, mapping program variables to elements of *Stb*, along with its concretization and abstraction functions.

The final part of this chapter focuses on the implementation of the *Stability* domain within the LiSA framework. We describe `Trend` class, representing the *Stb* lattice, and `Stability` class, which implements the analysis by wrapping together the *Stability* and auxiliary domains and at each program statement computes the new trend of a variable by reasoning on semantics and querying the auxiliary domain. This section concludes with a practical example, running the analyser on a simple IMP method and showing the results produced.

3.1 Increasing and decreasing variables

At each statement in the program, any variable has a *trend* representing how its value will be modified by that statement's execution. The *Stability* domain analysis aims to determine this trend for each variable at each statement. For

statements that do not modify variable values, all trends are simply propagated forward. For example, after the statement $st : if(a > 0)$, the trends of all variables will remain the same as they were before (they will not become '*stable*').

An *assignment* is a statement in the form $st : x = e$, where x is a variable and e is an expression. In particular, e can be: a constant, another variable, or an expression in the form $a\ OP\ b$, where a and b are expressions and OP is an operator (i.e., $+$, $-$, $\times$, or $\div$). When reaching an assignment, the analysis needs to determine the new trend of variable x.

Definition 3.1. (Increasing (decreasing) variables) Given a statement st which modifies the value of variable x, we say that x is *increasing* (*decreasing*) at st if it is true that the value of x after st is greater than (less than) the value of x before st.

Example 3.1. If $x = 0$, then at statement $st : x = 1$, x is increasing.

Example 3.2. For statement $st : x = x + 1$, whatever the value of x before st, we can be certain that x is increasing at st.

Definition 3.2. (Stable variables) We say that a variable is *stable* at statement st if its value before and after st is unchanged.

Example 3.3. Variable x is stable at statement $st : x = x + 0$.

The *Stability* domain must determine the trend of variables in the abstract, meaning without access to their concrete value, only reasoning on semantics. Examples 3.2 and 3.3 illustrate a fundamental concept: in some cases, stability information can be inferred from the structure of the statement, regardless of the actual concrete values.

This is not always true. Consider the following cases. The trend of x at statement $st : x = 1$ cannot be determined without knowing at least a range of values for x before st. Again, given statement $st : x = x \times 2$, x may be increasing or decreasing, depending on its sign before st. It may even be stable if its value is 0. Determining the trend of x at $st : x = x + y$ is impossible without information at least on the sign of y. For $st : x = x \times y$ we need information on the sign of x and the value of y. In instances like these, the *Stability* domain looses precision quickly.

3.2 Auxiliary abstract domain

The definitions of increasing, decreasing and stable variables seem to indicate the necessity for a comparison between the value of a variable before and after a statement. Even when this is not the case and stability information can be inferred from the semantics of statements, in most cases the information obtained is not precise enough to be useful, as illustrated at the end of the previous section. These conclusions can be refined by integrating knowledge with the support of another abstract domain, capable of adding key information on variables.

We say that the *Stability* domain *queries* an auxiliary abstract domain. This allows for direct comparison of some abstraction of the value of variables before and after a statement, as well as providing information useful for reasoning about semantics.

Example 3.4. Suppose we have statement $st : x = x + y$ and we can access the *Intervals* abstract domain. *Intervals* guarantees that, before st, $\{x \mapsto [1, 2], y \mapsto [4, 6]\}$ and that the state of x after st is [5, 8]. Then by comparing the prestate ([1, 2]) and poststate ([5, 8]) of x in the auxiliary domain, it is possible to conclude that x is increasing at st, since its value after st is necessarily greater than its value before st.

Different auxiliary domains can provide different levels of precision. In the same example, querying the *Sign* domain in the same way is less precise: the prestate of x in *Sign* is '+', its poststate is '+' as well, and comparing them gives no information on the trend of x.

3.2.1 Reasoning on semantics

In general, it is possible to gain more accurate information from reasoning on the semantics of a statement in relation to a variable, rather than comparing its prestate and poststate. Consider the following example.

Example 3.5. Suppose we have statement $st : x = x + y$ and the *Intervals* abstract domain guarantees that, before st, $\{x \mapsto [1, 4], y \mapsto [1, 2]\}$ and that the state of x after st is [2, 6]. We can draw no conclusions by comparing prestate and poststate: considering only the abstract values of x in *Intervals*, x could be increasing (e.g., $x = 3 \rightarrow x = 6$), decreasing (e.g., $x = 3 \rightarrow x = 2$), or stable (e.g., $x = 3 \rightarrow x = 3$). However, by reasoning on semantics and with the numerical information provided by *Intervals* we can conclude that x must be increasing at st, because we are adding a positive value to it.

For each statement in the form $st : x = x\ OP\ y$, where OP is one of the operators $\{+, -, \times, \div\}$, it is possible to construct a logic based on the operation semantics and on queries to an auxiliary domain to determine the trend of x. We query an auxiliary abstract domain A trough the call $query_A(x, c)$, where x is a variable and c a constant and $query_A(x, c)$ returns:

- *isEqual*, if A is able to guarantee that property $x = c$ holds at the current statement st;
- *isGreater*, if A is able to guarantee that $x > c$ holds at st;
- *isGreaterOrEq*, if A is able to guarantee that $x \geq c$ holds at st;
- *isLess*, if A is able to guarantee that $x < c$ holds at st;
- *isLessOrEq*, if A is able to guarantee that $x \leq c$ holds at st;
- *isNotEq*, if A is able to guarantee that $x \neq c$ holds at st;
- *unknown*, if A is not able to guarantee that any of the previous properties holds at st.

Example 3.6. Suppose we have statement $st : x = x + y$ and we can access the *Sign* abstract domain, which can answer questions: *"is variable y at statement st positive/negative/zero/non negative/non positive/different from zero?"*, i.e., queries in the form $query_{SIGN}(y, 0)$. Then, determining the trend of variable x at st can be achieved by querying *Sign*, as shown by the following pseudo-code.

$answ = query_{SIGN}(y, 0)$
switch *answ* **do**
case *isEqual*:
x is **stable**
break;
case *isGreater*:
x is **increasing**
break;
case *isLess*:
x is **decreasing**
break;
case *isGreaterOrEq*:
x is **not decreasing**
break;
case *isLessOrEq*:
x is **not increasing**
break;
case *isNotEq*:
x is **not stable**
break;
default:
x is **unknown**
break;

end switch

Note that the answer provided by the auxiliary domain is not based on the concrete values of the variables, but on their abstraction. Therefore, it is possible that none of the conditions is verified: *Sign* may not be able to ensure that y is positive, negative, nor zero, etc, and the query returns *unknown*. If we had its concrete value this would not be possible (a value is either positive, negative or zero).

Example 3.7. Consider, now, statement $st : x = x \times y$. The *Sign* domain is not sufficient to reach a precise conclusion, as knowing, for example, that both x and y are positive may still result in increasing (e.g., $x = 2$, $y = 2$), decreasing (e.g., $x = 2$, $y = 0.5$), or stable (e.g., $x = 2$, $y = 1$) trend for x. In this case, instead, we may query the *Interval* abstract domain, which can answer questions: *"is variable y at st greater than/less than/equal to/greater or equal than/less or equal than/different from value c?"* Then we can determine the trend of x as follows:

$ans_x = query_{INT}(x, 0)$
$ans_y = query_{INT}(y, 1)$
switch ans_x **do**
case *isGreater*:
switch ans_y **do**
case *isGreater*:
x is **increasing**
break;
case *isLess*:
x is **decreasing**

```
            break;
        case isEqual:
            x is stable
            break;
        case isGreaterOrEq:
            x is not decreasing
            break;
        case isLessOrEq:
            x is not increasing
            break;
        case isNotEq:
            x is not stable
            break;
        default:
            x is unknown
            break;

    end switch
case isEqual:
    ...
case isLess:
    ...
case isGreaterOrEq:
    ...
case isLessOrEq:
    ...
case isNotEq:
    ...
default:
    x is unknown
    break;

end switch
```

All omitted cases hold logical structures analogous to the one shown in the case *isGreater* for ans_x.

Appendix A shows, with the same notation from the examples above, the complete logic necessary for each of the operators.

Note that in the following cases:

- *st* is not an assignment statement;
- *st* is an assignment, but x is not one of the operands; and
- whenever the logic of operations returns unknown trend for x (i.e., there is no information to be obtained from the semantics of the statement);

it is a viable approach to query the auxiliary domain with questions in the form: *"is the value of variable x after st* $> / < / = / \geq / \leq / \neq$ *the value of variable x before st?"* (i.e., comparing prestate and poststate of x) and provide an answer based on the response. If the domain cannot give a useful response, then the trend for x at *st* remains unknown.

3.2.2 Logic of operations

In this section we overview the conclusions that can be drawn from the complete set of conditions that make up the logic of operations for assignment statements in the form $st : x = x\ OP\ y$, with OP either $\{+, -, \times, \div\}$, having access to auxiliary information from a different abstract domain A.

These considerations on the semantics of operations show which type of queries an auxiliary domain must be able to answer for each operator, as well as aid in the implementation of the domain in LiSA, delineating what will be necessary functions and their internal logic.

On queries' results

Note that $query(x, c)$ returns the most precise property it can guarantee for x. *isGreater* implies *isGreaterOrEq*, *isNotEq* and *unknown*, for example.

Another crucial point to understand when analysing the logic in Appendix A, is that *isEqual* being false, for example, does not imply that *isNotEq* is true, as we are reasoning on the abstract domain and these properties represent what the domain can infer from the semantics on the possible value of a variable.

Inverting trends

In many cases, the logic of one operation can be regarded exactly as the *inverse* of another, if we consider:

- $=$ is the inverse of itself,
- $\neq$ is the inverse of itself,
- $(\uparrow, \downarrow)$ are each the inverse of the other,
- $(\uparrow=, \downarrow=)$ are each the inverse of the other.

Addition and subtraction

Consider the logic presented by Appendix A for cases $st : x = x + y$ or $st : x = y + x$. We call this paradigm *increasingIfGreater()*, and we can say that for the addition operator the trend of $x = increasingIfGreater(y, 0)$.

Considering case $st : x = x - y$, we can observe that, at parity of abstract values of y in A, the trend of x at $st : x = x - y$ is the inverse of the trend of x at $st : x = x + y$ or $st : x = y + x$, and vice versa. We can say that for the subtraction operator the trend of $x = inverse(increasingIfGreater(y, 0))$.

Multiplication

Observing the different cases for $st : x = x \times y$ or $st : x = y \times x$ we note, in particular, that:

- if $query_A(x, 0) = isGreater$, the trend of $x = increasingIfGreater(y, 1)$;
- if $query_A(x, 0) = isLess$, the trend of $x = inverse(increasingIfGreater(y, 1))$;
- if $query_A(x, 0) = isGreaterOrEq$, we can identify new paradigm *nonDecreasingIfGreater()* and say that the trend of $x = nonDecreasingIfGreater(y, 1)$;

- if $query_A(x, 0) = isLessOrEq$, the trend of $x = inverse(nonDecreasingIfGreater(y, 1))$.

Division

In the case $st : x = x \div y$ we can observe:

- if $query_A(x, 0) = isGreater$, we have new paradigm *incIfBetweenZeroOne()*, and the trend of $x = incIfBetweenZeroOne(y)$;
- if $query_A(x, 0) = isLess$, the trend of $x = inverse(incIfBetweenZeroOne(y))$;
- if $query_A(x, 0) = isGreaterOrEq$, we can identify new paradigm *nonDecIfBetweenZeroAndOne()* and say that the trend of $x = nonDecIfBetweenZeroAndOne(y)$;
- if $query_A(x, 0) = isLessOrEq$, the trend of $x = inverse(nonDecIfBetweenZeroAndOne(y))$.

3.3 Formal Definition

The *Stability* domain can be formally defined as a complete lattice, whose elements are in the form $Vars_P \to Stb$, meaning the abstract properties map variables of a program P to analyse, to elements of Stb. Stb is a lattice defined in this chapter, whose elements represent trends of variables. It is important to note that this map corresponds to a specific point of the program and at each statement of the program the map is different. So it is more accurate to say, that *Stability* maps variables to their trends at a specific statement.

This section presents the formal definition of the *Stability* domain as a lattice, along with is operators.

3.3.1 The *Stb* lattice

The *Stb* complete lattice represents the trends of variables and the relation among these trends. We define:

$$Stb \triangleq \langle \{\bot, \uparrow, =, \downarrow, \uparrow=, \neq, \downarrow=, \top\}, \sqcup_{Stb}, \sqcap_{Stb}, \sqsubseteq_{Stb}, \bot, \top \rangle$$

The meaning of the lattice elements is explained as follows:

- $\bot$, the bottom element, i.e., the variable has no trend;
- $\uparrow$, the variable is strictly increasing;
- $=$, the variable is stable;
- $\downarrow$, the variable is strictly decreasing;
- $\uparrow=$, the variable is non-strictly increasing (i.e., it is not decreasing);
- $\neq$, the variable is not stable;
- $\downarrow=$, he variable is non-strictly decreasing (i.e., it is not increasing);
- $\top$, the top element, i.e., the variable can have any trend.

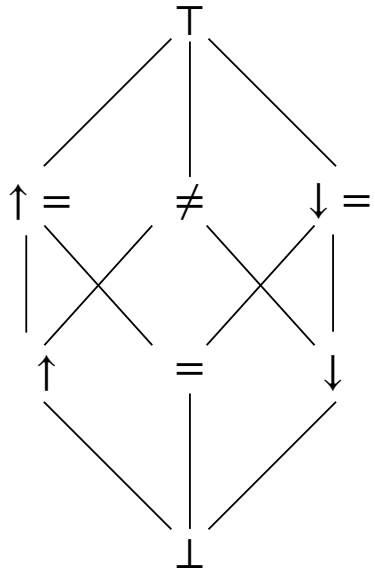

FIGURE 3.1: Hasse diagram of Stb lattice.

Figure 3.1 shows the Hasse diagram representing the lattice. Definitions for the operators can be derived from the diagram, based on the description from Section 2.3.1.

Example 3.8. The least upper bound of elements $\uparrow$ and $\downarrow$ is $\neq$, as both elements are below it and connected to it, which indicates that both $\uparrow \sqsubseteq_{Stb} \neq$ and $\downarrow \sqsubseteq_{Stb} \neq$. $\top$ is also an upper bound of set $\{\uparrow, \downarrow\}$, but since $\neq \sqsubseteq_{Stb} \top$, $\neq$ is the *lub*.

Example 3.9. The greatest lower bound of elements $\uparrow$ and $\downarrow=$ is $\bot$, as both elements are above it, $\bot$ is directly connected to $\uparrow$ and, by the transitive property of the order relation, $\bot$ is connected to $\downarrow=$ by an implied transitive edge, indicating that both $\bot \sqsubseteq_{Stb} \uparrow$ and $\bot \sqsubseteq_{Stb} \downarrow=$.

3.3.2 *STB* abstract domain

The *STB* lattice represents a domain mapping variables ($Vars_P$ indicating the set of variables of program P) to properties representing their trend at a statement. The *STB* lattice is defined as:

$$STB \triangleq \langle Vars_P \to Stb, \sqcup_{STB}, \sqcap_{STB}, \sqsubseteq_{STB}, \bot_{STB}, \top_{STB} \rangle$$

Elements of this lattice are properties mapping variables of a program to elements of *Stb*.

The lattice operators $\sqcup_{STB}, \sqcap_{STB}, \sqsubseteq_{STB}$ are defined as point-wise applications of the corresponding operators of *Stb*, meaning, for example, that the $\sqcup_{STB}$ of two maps results in another map where each variable is mapped to the $\sqcup_{Stb}$ of its values in the two maps. The bottom element $\bot_{STB}$ corresponds to the empty map, while the top element $\top_{STB}$ corresponds to the map where every variable is mapped to $\top \in Stb$.

Example 3.10. The least upper bound $\sqcup_{STB}$ of $\{x \mapsto \text{'}\uparrow\text{'}, y \mapsto \text{'}\uparrow\text{'}\}$ and $\{x \mapsto \text{'}\downarrow\text{'}, y \mapsto \text{'}\downarrow{=}\text{'}\}$ is: $\{x \mapsto \text{'}\neq\text{'}, y \mapsto \text{'}\top\text{'}\}$, because '$\uparrow$' $\sqcup_{Stb}$ '$\downarrow$' is '$\neq$' and '$\uparrow$' $\sqcup_{Stb}$ '$\downarrow{=}$' is '$\top$'.

Elements of *STB* can be concretized into sets of program traces for which the abstraction would reach that state. This means that concretization function γ is defined in terms of abstraction function α. We indicate as $\mathbb{T}$ the set of all traces. Then, for an element s of *STB*:

$$\gamma(s) = \bigcup \{T \in \wp(\mathbb{T}) \mid \alpha(T) \sqsubseteq_{STB} s\}$$

Example 3.11. Given an abstract state $\{x \mapsto \text{'}\uparrow\text{'}, y \mapsto \text{'}\uparrow\text{'}\}$, its concretization is the set of all traces of program P for which *Stability* can indeed state that both x and y are strictly increasing.

For a set of program traces $T \in \wp(\mathbb{T})$, its abstraction $\alpha(T)$ is defined as the *lub* $\sqcup_{STB}$ of the set of all $\dot{\alpha}(t_i)$, $t_i \in T$, with:

$$\dot{\alpha}(t_i) = \begin{cases} \{x \mapsto \text{' = '} \mid x \in Vars_P\} & \text{if } i = 0 \\ \{x \mapsto st(t_i, x) \mid x \in Vars_P\} & \text{otherwise} \end{cases}$$

where t_0 represents the initial state of the program, and $st(t_i, x)$ is a function that determines the trend of variable x at the last statement st_l of trace t_i, according to definitions 3.1 and 3.2:

$$st(t_i, x) = \begin{cases} \uparrow & \text{if } x \text{ is } \textit{increasing} \text{ at } st_l \\ = & \text{if } x \text{ is } \textit{stable} \text{ at } st_l \\ \downarrow & \text{if } x \text{ is } \textit{decreasing} \text{ at } st_l \\ \uparrow= & \text{if } x \text{ is not } \textit{decreasing} \text{ at } st_l \\ \neq & \text{if } x \text{ is not } \textit{stable} \text{ at } st_l \\ \downarrow= & \text{if } x \text{ is not } \textit{increasing} \text{ at } st_l \end{cases}$$

Concretization and abstraction functions form a Galois connection, since the condition: $\forall T \in \wp(\mathbb{T}), \forall s \in STB\ ,\ \alpha(T) \sqsubseteq_{STB} s \iff T \subseteq \gamma(s)$ is inferred from the definition of γ, as, by definition, $\gamma(s)$ is a set of Ts for which $\alpha(T) \sqsubseteq_{STB} s$.

3.4 LiSA Implementation

In this section we present how the concepts from this chapter can be applied through the LiSA framework. Given an IMP program, we are able to extrapolate *Stability* information by constructing an abstract state from the combination of both heap and value information. We implemented a `ValueDomain` to represent the *Stb* lattice and enclosed it into a `ValueEnvironment` to map identifiers to elements of this domain. Then we built the `Stability` class, wrapping *Stability* and auxiliary domain together and implementing their communication through the *query* operation.

3.4.1 `Trend` class

The `Trend` class is a `ValueDomain` which represents the *Stb* lattice defined in Section 3.3.1. We identify four classes of methods, based on their function:

- Lattice methods: all methods necessary to model the lattice operations correctly, according to the logic derived from the Hasse diagram of the *Stb* lattice (e.g., `lubAux`, `glbAux`, `lessOrEqualAux`, ...).
- Logic of operations methods: methods that implement the paradigms identified in Section 3.2.2 (e.g., `generateTrendIncIfGt()`, `generateTrendIncIfBetween()`, ...).
- Invert method: `invert()` returns the inverse of a given `Trend` following the rules from Section 3.2.2.
- Combination method: `combine()` implements the combination of two trends, which will be relevant for determining covariance or contravariance of variables, as discussed in Chapter 4.

The complete implementation of the class is presented in Appendix B. In particular, note that the logic of operation methods take as parameters a set of boolean values, representing the possible return values for $query(x, c)$. For example, consider the code of method `generateTrendIncIfGt()`, corresponding to paradigm *increasingIfGreater()*.

```
1  public static Trend generateTrendIncIfGt(
2      boolean isEqual,
3      boolean isGreater,
4      boolean isGreaterOrEq,
5      boolean isLess,
6      boolean isLessOrEq,
7      boolean isNotEq) {
8
9  if (isEqual) return STABLE;
10 else if (isGreater) return INC;
11 else if (isGreaterOrEq) return NON_DEC;
12 else if (isLess) return DEC;
13 else if (isLessOrEq) return NON_INC;
14 else if (isNotEq) return NON_STABLE;
15
16 else return TOP;
17 }
```

The boolean values of parameters `isEqual`, `isGreater`, `isGreaterOrEq`, etc, are determined by the caller (via queries to the auxiliary domain).

The STB abstract domain is represented by a `ValueEnvironment<Trend>` object, which will map variables to `Trend`s.

3.4.2 `Stability` class

`Stability` is a generic class, parameterized over the type of its auxiliary domain `<V extends BaseNonRelationalValueDomain<V>`. It itself implements `ValueDomain<Stability<V>` and `BaseLattice<Stability<V>`.

`Stability` has two fields: `auxiliaryDomain`, of type `ValueEnvironment<V>`, which represents the abstract domain providing auxiliary information for the analysis, and `trend`, of type `ValueEnvironment<Trend>`, representing the state of the STB lattice itself. `trend` maps variables to their `Trend`s, and `auxiliaryDomain` maps variables to their abstract properties in `V`. `Stability` handles simultaneously the analysis in both domains.

For example, method `lubAux()` performs the *lub* operation between two `Stability`s returning a `Stability` for which:

- `auxiliaryDomain` corresponds to the *lub* of the two `auxiliaryDomain`s (i.e., $\sqcup_{AUX}$ operation, `lub()` method of class `V`), and
- `trend` is the *lub* of the two `trend`s (i.e., $\sqcup_{STB}$ operation, `lub()` method of class `Trend`).

We give a brief overview of the purposes of relevant methods of this class.

- Lattice methods: methods for handling the lattice aspects of the class (i.e., `lubAux()`, `glbAux()`, `wideningAux()`, `lessOrEqualAux()`, `top()`, `bottom()`, `isTop()`, `isBottom()`).
- Domain methods: methods for handling the evolution of abstract information, based on the semantics of statements and expressions (i.e., `pushScope()`, `popScope()`, `assign()`, `smallStepSemantics()`, `assume()`, `knowsIdentifier()`, `forgetIdentifier()`, `forgetIdentifiersIf()`, `satisfies()`). For all these methods, `assign()` excluded, the result is some form of combination of computations delegated to `auxiliaryDomain` and `trend`.
- Query method: `query()` is used to determine the results of the $query_A(x, c)$ function from Section 3.2.1.
- Auxiliary logic methods: methods implementing one specific branch of the logic of operations, as defined in Section 3.2.2 (e.g., `increasingIfGreater()`, `nonDecreasingIfGreater()`, `incIfBetweenZeroAndOne()`, ...).
- Correlation methods: `environmentCombine()` implements function relative to determining covariance or contra-variance of variables, discussed in Chapter 4.

Lattice methods and most Domain methods work similarly, delegating to the two `ValueEnvironment`s the actual operations and implementing the logic at the `Stability` level. For example:

```
1 public Stability<V> forgetIdentifier(Identifier id)
2         throws SemanticException {
3     return new Stability<>(
4         auxiliaryDomain.forgetIdentifier(id),
5         trend.forgetIdentifier(id));
6 }
```

The `assign()` method is the notable exception, handling directly the computation of the *Stability* abstract state, which requires information from both domains. `assign()` method is called by the domain when encountering an assignment statement $st : x = e$. It determines the abstract value for variable x after st in both the auxiliary domain and in the *Stability* domain and yields a copy of the current `Stability` where x is mapped to these new values. The computation for the auxiliary domain is handled by `auxiliaryDomain`, forwarding the call to `assign()` of class `ValueEnvironment<V>`.

```
ValueEnvironment<V> ad =
        auxiliaryDomain.assign(id, expression, pp, oracle);
```

The computation of the *Stability* value is implemented within `assign()`, following exactly the logic presented in Appendix A and discussed in Section 3.2.2 and utilising the set of Auxiliary methods. The Auxiliary methods themselves call the Logic of operations methods of `Trend` class, whose parameters are computed as results of `query()`.

Consider, for example, the call to `Trend.generateTrendIncIfGt()` in `increasingIfGreater()` method:

```
return Trend.generateTrendIncIfGt(
  query(binary(ComparisonEq.INSTANCE, a, b, pp), pp, oracle),
  query(binary(ComparisonGt.INSTANCE, a, b, pp), pp, oracle),
  query(binary(ComparisonGe.INSTANCE, a, b, pp), pp, oracle),
  query(binary(ComparisonLt.INSTANCE, a, b, pp), pp, oracle),
  query(binary(ComparisonLe.INSTANCE, a, b, pp), pp, oracle),
  query(binary(ComparisonNe.INSTANCE, a, b, pp), pp, oracle)
);
```

where `a` and `b` are `SymbolicExpressions`, with `a` representing x and `b` representing the constant c. `binary()` is an auxiliary method that given a `BinaryOperator` comparison operator *CMP* and two `SymbolicExpressions` a and b returns a `BinaryExpression` object representing 'a *CMP* b'. Method `query()` receives a `BinaryExpression q` and returns true if the result of:

```
auxiliaryDomain.satisfies(q, pp, oracle)
```

is '`SATISFIED`'. `generateTrendIncIfGt()` receives this set of boolean values and implements exactly the addition branch defined in the logic of operations, as shown in Section 3.4.1. Any other case is handled in an analogous way.

After the new `auxiliaryDomain ad` and `trend t` have been computed, `assign()` returns:

```
1 if (ad.isBottom() || t.isBottom())
2     return bottom();
3 else
4     return new Stability<>(ad, t);
```

meaning, if either domain went to bottom, the whole analysis goes to bottom. Otherwise, we return the new `Stability` representing the state after *st*.

3.4.3 Running the analysis

In order to run the analysis we create a new `LiSA` object and run it on a `Program`. The `LiSAConfiguration` parameter is defined, in particular, with field:

```
conf.abstractState = new SimpleAbstractState<>(
    // heap domain
    new FieldSensitivePointBasedHeap(),
    // value domain
    new Stability<>(
        new ValueEnvironment<>(new Interval()).top()),
    // type domain
    new TypeEnvironment<>(new InferredTypes()));
```

The results of the analysis can be dumped into a JASON file for each method/CFG by setting:

```
conf.serializeResults = true;
```

Additionally, HTML files can be produced for better visualization by setting:

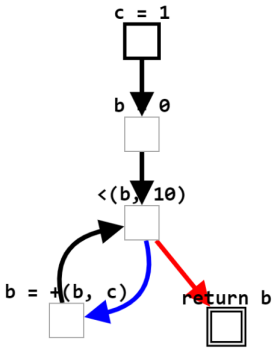

FIGURE 3.2: CFG produced by LiSA analysier for code from Example 3.12

```
conf.analysisGraphs = LiSAConfiguration.GraphType.HTML;
```

The following example shows the results of running the configuration above with this second option enabled on sample code.

Example 3.12. Suppose we run the analysis on the following IMP code.

```
1 class tutorial {
2     constants() {
3         def c = 1;
4         def b = 0;
5         while (b < 10)
6             b = b + c;
7         return b;
8     }
9 }
```

The `IMPFrontend` produces a single CFG representing the `constants()` method and the analysis is run on it with `Stability` value domain, producing for each node a poststate representing the stability of variables at the corresponding statement. These results are transcribed into an HTML file. Figure 3.2 shows the graph presented by this file when opened on a browser. Selecting a node will show the corresponding results for each of the three domains of the `AbstractState`. We summarise the results for the `Stability` environment for each node, referencing the corresponding statement from the code and its label in the CFG.

```
st2 ('c = 1'):  c -> stable
st3 ('b = 0'):  b -> stable, c -> stable
st4 ('<(b, 10)'):  b -> non-decreasing, c -> stable
st5 ('b = +(b, c)'):  b -> increasing, c -> stable
st6 ('return b'):  b -> non-decreasing, c -> stable
```

A variable is stable at the statement defining it. The trends at `st4` are the *lub* of trends at `st3` and `st5`. So the $\sqcup_{STB}$ of $\{b \mapsto =, c \mapsto =\}$ and $\{b \mapsto \uparrow, c \mapsto =\}$. The trend of `b` at `st5` is increasing because of the semantics reasoning, integrated with the knowledge provided by `Interval` that `c` is greater than 0. Finally, trends at `st6` are the trends at `st4`, its predecessor, propagated forward.

The `stabiltiyTest()` method in Appendix E shows the complete configuration of LiSA employed in Example 3.12.

4 Covariance and Contra-variance

This chapter explores the application of the *Stability* domain to statically determining correlation of variables over blocks of code.

We begin with an intuitive explanation of how to determine covariance and contra-variance from *Stability* information. Next, we formally define these concepts, including the process of combining trends to produce a cumulative stability representing the overall change of a single variable over a code block. We then formally define covariant and contra-variant variables.

The last section details the practical implementation of a `SemanticCheck` within the LiSA framework, which runs on the results of the *Stability* analyser to determine covariance and contra-variance. By integrating covariance and contra-variance into LiSA, we enhance its capability to detect and analyze complex variable interactions.

4.1 Intuition

Covariance and contra-variance of variables can be determined from the results provided by the *Stability* abstract domain: if two variables have the same trend at statement st, then they are covariant at st, if they have "*opposite*" trends, then they are contra-variant. This evaluation is straightforward, but not sufficient to verify covariance and contra-variance in a program.

Example 4.1. Consider the following code with corresponding results of the *Stability* analysis, supposing that variables x and y are both *stable* before the branch.

```
1 if (a > 0){
2     x = x + 1;     // x -> increasing, y-> stable
3     y = y + 1;     // x -> increasing, y-> increasing
4 }
```

It is intuitive that variables x and y are covariant for this branch. However, a sequential analysis of the code, comparing trends statement by statement would show that x and y do not have the same trend throughout the block; instead, at statement st_2 x and y are not covariant.

A possible solution seems to be to only consider the last statement of the block, when all operations have been carried out, and compare the stability of variables there.

Example 4.2. Consider the case:

```
1 if (a > 0){
2      x = x - 100;     // x -> decreasing, y-> stable
3      y = y + 1;       // x -> decreasing, y-> increasing
4      x = x + 1;       // x -> increasing, y-> increasing
5 }
```

with x and y both *stable* before the branch. At statement st_4 x and y are covariant but it is not true that they are covariant for this branch.

Our proposed solution is the introduction of a *cumulative stability*, described in Section 4.2.2, capable of capturing some overall information on the trend of a variable through multiple statements. Then, given a block of code to analyse, we compare this cumulative information of variables over the whole block to determine correlation.

Going back to Example 4.1, the cumulative stability of x and y over the block is: $\{x \mapsto\uparrow,\ y \mapsto\uparrow\}$, meaning x and y are covariant. For the code of Example 4.2, the cumulative stability of x and y after statements 1-4 is: $\{x \mapsto \top,\ y \mapsto\uparrow\}$. From this we reach the correct result that x and y are not covariant.

4.2 Formal definition

As anticipated in the previous section, in order to formalise the concepts of covariance and contra-variance we need to provide an operation to combine trends of multiple statements. In the following paragraphs we define the concept of *code block*, of *combination* of two trends, of *cumulative stability* of a variable over multiple statements, and, finally, of *covariant* and *contra-variant* variables.

4.2.1 Code blocks

A code block represents the sequence of statements over which covariance and contra-variance are evaluated. We say that statement st_a *precedes* another statement st_b in a program if, when running the code, st_a is executed first, before st_b. Statement st_a is a *predecessor* of st_b if, for some execution, st_b can be executed immediately after st_a.

Definition 4.1. (Code block) A *code block* is a sequence of statements identified as $b(st_a, st_b)$, where st_a and st_b are the first and the last statement of the block, respectively. A statement st belongs to block $b(st_a, st_b)$ if it is preceded by st_a and it precedes st_b.

4.2.2 Combining trends

We propose the definition of the combination of two trends into a single one, which can be applied sequentially to determine the cumulative stability of a code block. Intuitively, the combination of the two trends of x at statements st_1 and st_2 represents the stability information that can be gained after both

	$\bot$	$\uparrow$	$=$	$\downarrow$	$\uparrow=$	$\neq$	$\downarrow=$	$\top$
$\bot$	$\bot$	$\uparrow$	$=$	$\downarrow$	$\uparrow=$	$\neq$	$\downarrow=$	$\top$
$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\top$	$\uparrow$	$\top$	$\top$	$\top$
$=$	$=$	$\uparrow$	$=$	$\downarrow$	$\uparrow=$	$\neq$	$\downarrow=$	$\top$
$\downarrow$	$\downarrow$	$\top$	$\downarrow$	$\downarrow$	$\top$	$\top$	$\downarrow$	$\top$
$\uparrow=$	$\uparrow=$	$\uparrow$	$\uparrow=$	$\top$	$\uparrow=$	$\top$	$\top$	$\top$
$\neq$	$\neq$	$\top$	$\neq$	$\top$	$\top$	$\top$	$\top$	$\top$
$\downarrow=$	$\downarrow=$	$\top$	$\downarrow=$	$\downarrow$	$\top$	$\top$	$\downarrow=$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

FIGURE 4.1: Results of the *combine* operation between elements of the *Stb* lattice of Figure 3.1

statements, i.e., over block $b(st_1, st_2)$. Figure 4.1 shows a table representation of the *combine* operation, defined as follows.

Definition 4.2. (Combination) We define the *combine* operation on elements of the *Stb* lattice as the set of following rules:

1. *combine*(t, '$\bot$') $= t$ for any t in *Stb*;
2. *combine*(t, '$=$') $= t$ for any t in *Stb*, $t \neq$ '$\bot$';
3. *combine*(t, '$\neq$') $=$ '$\top$' for any t in *Stb*, $t \neq$ '$=$', $t \neq$ '$\bot$';
4. *combine*(t, t) $= t$ for any t in *Stb*, $t \neq$ '$\neq$';
5. *combine*('$\uparrow$', '$\uparrow=$') $=$ '$\uparrow$';
6. *combine*('$\downarrow$', '$\downarrow=$') $=$ '$\downarrow$';
7. *combine*(t_1, t_2) $=$ '$\top$' for any t_1, t_2 in *Stb* not matching any other rule.

Any case in which one rule is in conflict with another is listed as an exception, the reasoning being that any rule overrules the ones following it and is overruled by those preceding it. Rule "*combine*(t, '$\top$') $=$ '$\top$' *for any* t *in Stb*" can be derived from the existing set of rules and it does indeed hold for the *combine* operation.

Definition 4.3. (Cumulative Stability) The *cumulative stability* of a variable x over block $b(st_a, st_b)$ is:

- the stability of x at st_a if $st_a = st_b$;
- the combination of:
  1. the stability of x at st_b; with
  2. the *lub* of all *cumulative stabilities* of x over $b(st_a, st_i)$, where each st_i is a predecessor of st_b;

  if $st_a \neq st_b$.

Example 4.3. Consider the code with corresponding results provided by the *Stability* abstract domain:

```
1  fun(a){
2    def x = 0;       // {x} -> stable
3    def y = 1;       // {x, y} -> stable
4    def z = 1;       // {x, y, z} -> stable
5
6    if (a > 1){
7      x = x + 1;     // {x} -> increasing, {y, z} -> stable
8      y = y - 1;     // {x} -> increasing, {y} -> decreasing,
9                                              // {z} -> stable
10     z = z * a;     // {x, z} -> increasing, {y} -> decreasing
11   }
12
13   return x + y;    // {x, z} -> not decreasing,
14 }                                  // {y} -> not increasing
```

Suppose we want to compute the cumulative stability of x over code block $b(st_4, st_{13})$ (we leave out variable definitions for brevity). This is defined as:

$$combine('\uparrow =', lub(combine('\uparrow', combine('\uparrow', combine('\uparrow', '=')), '='))),$$

meaning the combination of stability at st_{13} ('↑=') with the *lub* of cumulative stability at st_{10} and at st_4, predecessors of st_{13}. The cumulative stability at st_{10} is the combination of stability at st_{10} ('↑') with cumulative stability at st_8, which itself is the combination of stability at st_8 ('↑') with cumulative stability at st_7, meaning the combination of stability at st_7 ('↑') with cumulative stability at st_4, which equals to the stability itself ('=') because st_4 is the base case. From here, the computation consist in applying the combination and *lub* operators:

$$combine('\uparrow =', lub(combine('\uparrow', combine('\uparrow', combine('\uparrow', '=')), '='))) =$$

$$combine('\uparrow =', lub(combine('\uparrow', combine('\uparrow', '\uparrow')), '=')) =$$

$$combine('\uparrow =', lub(combine('\uparrow', '\uparrow')), '=')) =$$

$$combine('\uparrow =', lub('\uparrow', '=')) =$$

$$combine('\uparrow =', '\uparrow =') =$$

$$'\uparrow =$$

4.2.3 Covariant and contra-variant variables

Knowing that we consider the following pairs of trends in the *Stb* lattice as pairs of *opposite trends*:

$$('=', '\neq'), ('\uparrow', '\downarrow'), ('\uparrow =', '\downarrow =') ,$$

and from the previous definitions in this section, we can finally provide a definition of covariance and contra-variance.

Definition 4.4. (Covariant (contra-variant) variables) A pair of variables x and y are *covariant (contra-variant)* across code block $b(st_a, st_b)$ if they have the same (*opposite*) cumulative stability over $b(st_a, st_b)$.

Based on this definition, referencing Example 4.3 where the cumulative stability of variables over $b(st_1, st_{14})$ are $\{x \mapsto$ '↑=', $y \mapsto$ '↓=', $z \mapsto$ '↑='$\}$, we can say that variables x and y are contra-variant, while variables x and z are covariant across the code block. Note that this property implies that for any possible execution of `fun()`, x and y are never not contra-variant and x and z are never not covariant.

Covariance and contra-variance of variables are strong properties for a program. In the context of anomaly detection, if we can prove formally that variables representing determined concepts must be covariant or contra-variant, then a static analysis capable of individuating violations of these properties can identify potential errors, security vulnerabilities, inappropriate program logic, or insecure system designs.

4.3 LiSA implementation

The `LiSAConfiguration` class holds a collection of `SemanticChecks` to be executed after the fixpoint iteration has been completed. These checks are run with access to the structure of the code and to the results of the analysis.

In order to determine the correlation between variables, we define a new `SemanticCheck`, the `CoContraVarianceCheck` class, and implement the `visit()` method. This method will be called on each node of the graph and compute the cumulative stability of variables up to that node.

The combination of `Stability` environments is operated according to the rules for computing cumulative trends over code blocks. The result of `CoContraVarianceCheck` is a `Map` from each node to a cumulative `Stability` object.

4.3.1 `visit()` method

This method receives parameters `tool`, representing the results of the *Stability* analysis, `graph`, representing the current CFG, and `node`, representing the current node within the CFG. For node n, `visit()` identifies the state of the analysis before and after the corresponding statement and combines them into a single cumulative state representing the cumulative stability of all variables from the first statement of the block to the current one. `visit()` retrieves two `Stability` environments: a `preState`, computed by its predecessor nodes as shown later on, and a `postState`, being the stability information of variables at that statement, obtained from the results of the analysis. The analysis results are accessed from within the method by calling:

```
tool.getResultOf(graph)
```

which returns a collection of `AnalyzedCFGs`, each mapping a CFG to the results computed during the analysis. Then, if `result` is a variable containing one of these CFGs, we have:

```
Stability<T> postState = result.getAnalysisStateAfter(node)
                             .getState()
                             .getValueState();
```

From here, `visit()` combines `preState` and `postState` into a `cumulativeState` through the `environmentCombine()` method, as shown below.

```
1 if (preStatesMap.containsKey(node)) {
```

```
2       Stability<T> preState = preStatesMap.get(node);
3       cumulativeState = preState.environmentCombine(postState);
4   }
5   resultsMap.put(node, cumulativeState);
```

`preStatesMap` and `resultsMap` are two `Map<Statement, Stability<T>>` objects maintained by `CoContraVarianceCheck`. `preStatesMap` holds, for each node, the *lub* of the cumulative stabilities of all its predecessors, while `resultsMap` holds the cumulative stability for each node.

If n has no predecessors (i.e., n is the first statement of the program), then the `cumulativeState` is simply its `postState`. This can be seen as the base case for the computation of cumulative stability over code blocks from Definition 4.3. Otherwise, `cumulativeState` is the combination of `preState` and `postState`. The `cumulativeState` is saved in `resultsMap`.

`visit()` also computes the prestate of all nodes successors of n.

```
1   for (Statement next : graph.followersOf(node)) {
2       if (preStatesMap.containsKey(next)) {
3           try {
4               preStatesMap.put(
5                   next,
6                   preStatesMap.get(next).lub(cumulativeState));
7           } catch (SemanticException e) {
8               e.printStackTrace();
9           }
10      } else
11          preStatesMap.put(next, cumulativeState);
12  }
```

If a successor of n is not yet in the `preStatesMap`, then the prestate of that node is the `cumulativeState` of n. Otherwise, its prestate is updated to the *lub* of the existing prestate and `cumulativeState` of its predecessor n.

By construction and by Definition 4.3 of cumulative stability, the `cumulativeState` of n holds for each variable in the environment its cumulative stability from the first statement of the code block to n. Therefore, after the execution of `CoContraVarianceCheck`, `resultsMap` maps `Statements` to `Stability` objects, which in turn map each program variable to its cumulative stability up to that `Statement`.

4.3.2 `environmentCombine()` method

This method of the `Stability` class operates the combination of two `Stability` objects, representing the prestate and the poststate of a statement st_n. `environmentCombine()` returns a new `Stability` where all variables are mapped to a `Trend` corresponding to the combination of their stability in the two environments.

```
1   ValueEnvironment<Trend> retTrendEnv =
2           new ValueEnvironment<>(new Trend((byte) 0));
3
4   for (Identifier id : post.getTrend().getKeys()) {
5       if (pre.getTrend().knowsIdentifier(id)) {
6           Trend tmp = pre.getTrend().getState(id).combine(
7                       post.getTrend().getState(id));
8           retTrendEnv = retTrendEnv.putState(id, tmp);
9       }
```

```
10      else
11          retTrendEnv = retTrendEnv.putState(
12                            id,
13                            post.getTrend().getState(id));
14      }
15  }
```

Intuitively, the stability of each variable known to one environment is combined with the stability of that variable in the other environment. The combine operation is handled by the `combine()` method of the `Trend` class. In the case a variable is known after st_n but not before (i.e., the current statement is a declaration of that variable), its cumulative stability is just its stability after st_n, i.e., its poststate.

4.3.3 `combine()` method

The `combine()` method of the `Trend` class implements exactly the combination of two elements of the *Stb* lattice, represented by `Trend` objects, as defined in Section 4.2.2.

5 Study: the Scale Problem

This chapter demonstrates how covariance and contra-variance analysis can be used to analyse a practical scenario and prove correct behavior. The analysis is applied to IMP code performing the scale transformation of rectangles, in order to prove the correlation of specific sets of program variables.

First, we introduce the reasoning which allows us to determine covariance of a set of program variables as a requirement for correctness in the behavior of the sample code. Then, we run our analysis, showing how the results prove that the property is verified for all possible executions.

5.1 Scale transformation

In geometry, *scaling* refers to an affine transformation that enlarges or reduces a figure by a (separate) constant *scale factor* for each axis direction. A negative scale factor comports a *reflection* transformation along the corresponding axis. If all scale factors are the same, we talk about *uniform* scaling.

Mathematically, this transformation can be described using matrices in a coordinate system. For a figure in a two-dimensional plane, scaling can be represented by the matrix:

$$\begin{bmatrix} scale_x & 0 \\ 0 & scale_y \end{bmatrix}$$

where $scale_x$ and $scale_y$ are the scale factors for the x and y axis, respectively. When this matrix is applied to the coordinates of a point (x, y), the resulting point ($scale_x x$, $scale_y y$) represents the scaled version of the original point. Figure 5.1 shows a rectangle being scaled with factors $scale_x = 2$ and $scale_y = 1$.

5.2 Correlation of variables

By applying the scale transformation to a rectangle R we obtain rectangle R', such that for any point in R with coordinates (x_p, y_p) the coordinates of the corresponding point in R' are ($scale_x x_p$, $scale_y y_p$). From this follows that:

- coordinate $x'_p = x_p \times scale_x$, for any p' in R';

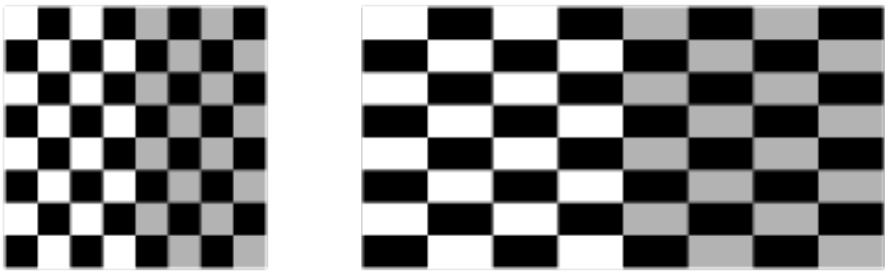

FIGURE 5.1: A rectangle before and after a scale transformation with scale factors $scale_x = 2$ and $scale_y = 1$

- width of R' = width of $R \times scale_x$;
- coordinate $y'_p = y_p \times scale_y$, for any p in R;
- height of R' = height of $R \times scale_y$.

Reasoning, for simplicity, about a rectangle R located in the first quadrant, with all vertices having positive x and y coordinates, we can say that for any code correctly computing the scaling of a rectangle:

- all variables representing the width of the rectangle or the x coordinate of any of its points must be covariant;
- all variables representing the height of the rectangle or the y coordinate of any of its points must be covariant.

Moreover, these two sets of variables may be covariant or contra-variant, depending on the values of $scale_x$ and $scale_y$:

- if both $scale_x$ and $scale_y$ are greater than 1, or they both are less than 1, then all variables from the two sets are covariant;
- if $scale_x > 1$ and $scale_y < 1$, or vice-versa, then the two sets of variables are contra-variant;
- otherwise the two sets are neither covariant nor contra-variant.

Similar reasoning can be extended to all rectangles, generating different sets of covariant variables. For example, in the general case we have that all variables representing the width of the rectangle or the x coordinate of any of its points with positive x must be covariant, and this set must be contra-variant to the set of all variables representing the x coordinate of any of its points with negative x. This is true because multiplying a positive or a negative value by the same factor (positive or negative) will result necessarily in inverse trends.

5.3 Running the analysis

Consider the `Rectangle` IMP class, representing a rectangle aligned with the coordinate axes for which we know width, height, and coordinates (x, y) of its bottom left point.

```
1  class Rectangle{
2      width;
3      height;
4      botLeftX;
5      botLeftY;
6  }
```

We analyse the following IMP code to scale a `Rectangle` by scale factors 2 and -1:

```
1  enlargeAndReflect(){
2      def r = new Rectangle();
3      def scaleX = 2;
4      def scaleY = -1;
5
6      def botRightX = r.botLeftX + r.width;
7      def topLeftY = r.botLeftY + r.height;
8
9      r.botLeftX = r.botLeftX * scaleX;
10     r.botLeftY = r.botLeftY * scaleY;
11
12     botRightX = botRightX * scaleX;
13     topLeftY = topLeftY * scaleY;
14
15     r.width = botRightX - r.botLeftX;
16     r.height = r.botLeftY - topLeftY;
17
18 }
```

Assuming all vertices of `r` have positive coordinates, we can apply the conclusions we drew on the scale transformation:

- {`r.botLeftX`, `botRightX`, `r.width`} must be covariant;
- {`r.botLeftY`, `topLeftY`, `r.height`} must be covariant; and
- the two sets must be contra-variant, because `scaleX`> 1 and `scaleY`< 1.

In order to verify covariance and contra-variance of variables, we run the configuration of LiSA shown in method `correlationTest()` of Appendix E. This code creates a LiSA object running the analysis with an `AbstractState` with value domain `Stability<Interval>`. This corresponds to the *Stability* domain with the *Interval* auxiliary domain.

The results of this analysis are dumped in an HTML file, providing a CFG representation of the program, shown in Figure 5.2. By interacting with the graph it is possible to access the inferred *Stability* of variables at each node, analogously to Example 3.12.

After the analysis is concluded, the analyser runs `CoContraVarianceCheck` on the results, meaning the `visit()` method is invoked on each node of the CFG. We represent, for each statement of the program, the corresponding cumulative stabilities saved in the `resultsMap`.

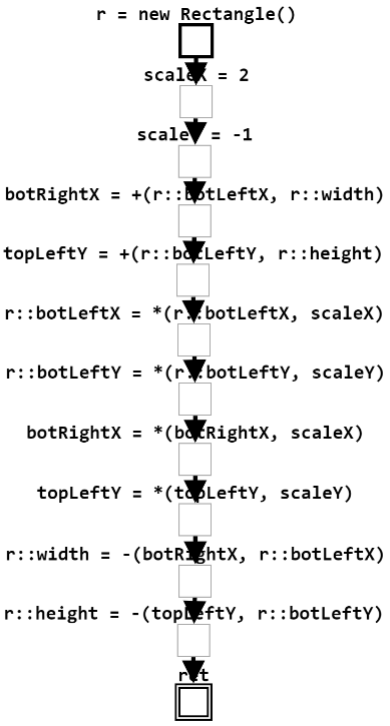

FIGURE 5.2: CFG of `enlargeAndReflect()` method.

```
st2: stable -> {r, r.width, r.height, r.botLeftX,
       r.botLeftY}
st3: stable -> {scaleX, r, r.width, r.height, r.botLeftX,
       r.botLeftY}
st4: stable -> {scaleY, scaleX, r, r.width, r.height,
       r.botLeftX, r.botLeftY}
st6: stable -> {botRightX, scaleY, scaleX, r, r.width,
       r.height, r.botLeftX, r.botLeftY}
st7: stable -> {topLeftY, botRightX, scaleY, scaleX, r,
       r.width, r.height, r.botLeftX, r.botLeftY}

st9: stable -> {topLeftY, botRightX, scaleY, scaleX, r,
       r.width, r.height, r.botLeftY} ,
       increasing -> {r.botLeftX}
st10: stable -> {topLeftY, botRightX, scaleY, scaleX, r,
       r.width, r.height} -> stable ,
       increasing -> {r.botLeftX} ,
       decreasing -> {r.botLeftY}

st12: stable -> {topLeftY, scaleY, scaleX, r, r.width,
       r.height, r.botLeftY} ,
       increasing -> {botRightX, r.botLeftX} ,
       decreasing -> {r.botLeftY}
st13: stable -> {scaleY, scaleX, r, r.width, r.height,
       r.botLeftY} ,
       increasing -> {botRightX, r.botLeftX} ,
       decreasing -> {topLeftY, r.botLeftY}

st15: stable -> {scaleY, scaleX, r, r.height, r.botLeftY} ,
       increasing -> {r.width, botRightX, r.botLeftX} ,
       decreasing -> {topLeftY, r.botLeftY}
```

```
st16: stable -> {scaleY, scaleX, r, r.botLeftY} ,
        increasing -> {r.width, botRightX, r.botLeftX} ,
        decreasing -> {r.height, topLeftY, r.botLeftY}
```

From these sets of covariant variables, we can indeed verify that:

- {`r.botLeftX`, `botRightX`, `r.width`} are covariant over the whole program since they have the same cumulative stability over code block $b(st_1, st_{18})$;
- for the same reason, {`r.botLeftY`, `topLeftY`, `r.height`} are covariant over the whole program; and
- the two sets are contra-variant over the program, because they have inverse cumulative stabilities '↑' and '↓' over block $b(st_1, st_{18})$.

Since the analysis verified the correctness properties above and we are ensured that the results are sound, it is possible to guarantee that there is no execution trace of this program for which the properties do not hold.

6 Related Work

General abstract framework has its main reference in the work of P. Cousot and R. Cousot [3] [8], who introduced the subject in the late 70s. In the recent past a number of different proposals have been studied and corresponding domains have been developed. In particular, for numerical values these include the *Sign* [3] and *Interval* domains [9], the *Pentagon* domain [10], improving the precision of interval analysis, the *Polyhedron* [11] and the *Octagon* [12] domains for reasoning on affine inequalities, the *Donut* domains [13] for working on non-convex invariants, etc.

Several domains have been also proposed for the analysis of strings, such as the *Character inclusion*, the *Prefix and suffix*, the *Bricks* and the *String graphs* domains [14], or the *String hash* domain [15].

As far as we know, no domain has been studied so far regarding the stability property.

Our proposal benefits from the work done on the extended sign domain, because of the lattice homomorphism that can be observed between the two domains.

Different studies considered possible combinations of domain with the aim of enhancing the precision of representation of abstract domains, expanding on the initial proposals of Cousot and Cousot of reduced product, disjunctive completion and reduced cardinal power [8]. A survey by Cortesi et al. [16] identified Granger product [17], open product [18], and reduced relative power [19].

Anomaly detection for software systems, as we observe from the literature, is largely employed for dynamic intrusion detection. This was introduced first as a paradigm for a single system based on the real-time identification of abnormal patterns of system usage [20]. A subsequent study, went on to re-define the original assumptions in the context of software systems, as a base for future research [21].

Different proposals have been developed, addressing several concerns, such as the poisoning of the data for training of anomaly detection sensors [6], poor performance and limited applicability [22].

A unified framework was proposed to define program anomaly detection

methods in terms of their detection capability [5], identifying as the common approaches: n-gram-based dynamic modeling of normal program behavior, automaton-based analysis of normal program behavior [23], probabilistic modeling methods [24] [25], and dynamically built state machines [26].

A relatively recent proposal with promising results is the introduction of machine learning classifiers to predict the health condition of the system [27].

As far as we know there are no domains designed in the abstract interpretation framework for anomaly detection.

7 Conclusions

This thesis allows to appreciate the benefits of applying static analysis to the task of anomaly detection for software programs, and presents the definition of a novel abstract domain for carrying out an analysis of correlation properties of variables based on abstract interpretation.

The *Stability* abstract domain was introduced for soundly extrapolating information on how the trends of variables change throughout a program. We formally defined as the STB lattice, its operands, and concretization and abstraction functions, showing they enjoy a Galois connection. This analysis may have many applications, but in the context of this work it provides a set of results which allow to determine properties of covariance or contra-variance of program variables.

Covariance and contra-variance have been introduced and we described how to determine these properties, starting from stability information, then computing a cumulative stability for variables over multiple statements, and finally comparing this aggregate information. In the appropriate contexts, supported by a formal proof of the necessity for variable correlation, this analysis provides a proof of correctness for the program behavior, guaranteeing that no possible state of execution violating the property can occur. Otherwise, it raises an alarm (possibly a false alarm), allowing to identify misconfigurations, program bugs, unusual usage patterns or other causes for reaching states which violate the properties.

We implemented this analysis within the LiSA framework, defining the `Trend ValueDomain` for representing trends, the `Stability<V> ValueDomain` implementing the logic necessary to infer stability information from statement semantics and auxiliary information from a domain of type `V`, and the `CoContravarianceCheck SemanticCheck` for computing the cumulative trends of variables at each statement. Utilising the infrastructure provided by LiSA, we successfully created and ran a static analyser on multiple code samples to obtain the stability of variables throughout the code and sets of covariant variables.

We showed a practical application, taking the case of the scale transformation as an example and demonstrating how the results of the analysis were able to verify previously determined properties.

By introducing the *Stability* domain and exploring covariance and contra-variance, this work provides new perspectives and tools for analyzing variable trends and, more generally, relations between variables, contributing to the development of more secure and robust software systems through static analysis and formal methods. The integration with the LiSA framework underscores the practical relevance of these proposals, making them accessible for real-world applications.

7.1 Future work

The results of this work leave several possibilities for future research. One potential direction is the further refinement and optimization of the *Stability* domain. In particular, we note that the implementation in LiSA of more abstract domains already present in the literature, such as the *Octagon* domain [12], could lead to results with a higher level of precision.

Additionally, research can be carried out to determine the relevance of the concepts of covariance and contra-variance in different specific classes of applications, in order to identify use cases in which such an analysis would provide the most benefits. By exploring the application of these techniques in different programming paradigms we could broaden their impact and utility. Moreover, once identified a specific class of programs for which correlation is a critical property, it would be possible to focus on optimising the analysis based on specific characteristics of the domain. An example would be comparing the results obtained by running the analysis with different auxiliary abstract domains, in order to determine the optimal one in terms of precision and performance.

One particular area we propose to focus on, is the application of the covariance and contra-variance analysis to robotic system applications, running analyses in order to evaluate its impact on improving security in this specific field and attempting an optimization as shown above. We believe that this is a field which would greatly benefit from a static approach to anomaly detection, in particular given its continuous growth in a variety of sectors with direct impact on everyday life and direct interaction with human beings [28].

Another promising area is the study of integration of these techniques with other formal methods and static analysis tools, with the aim of creating a more comprehensive framework for software verification, combining the strengths of various approaches to achieve greater accuracy and coverage.

Bibliography

[1] K. Y. Xavier Rival, *Introduction to static analysis : an abstract interpretation perspective*. MIT Press, 2020.

[2] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, no. 74.2, pp. 358–366, 1953.

[3] P. Cousot and R. Cousot, "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*, ser. POPL '77, Los Angeles, California: Association for Computing Machinery, 1977, 238–252, ISBN: 9781450373500. DOI: 10.1145/512950.512973. [Online]. Available: https://doi.org/10.1145/512950.512973.

[4] P. Cousot, *Principles of Abstract Interpretation*. MIT Press, 2021, ISBN: 9780262044905. [Online]. Available: https://books.google.it/books?id=Cwk_EAAAQBAJ.

[5] X. Shu, D. Yao, and B. G. Ryder, "A formal framework for program anomaly detection," in *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings 18*, Springer, 2015, pp. 270–292.

[6] G. F. Cretu, A. Stavrou, M. E. Locasto, S. J. Stolfo, and A. D. Keromytis, "Casting out demons: Sanitizing training data for anomaly sensors," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, IEEE, 2008, pp. 81–95.

[7] P. Ferrara, L. Negrini, V. Arceri, and A. Cortesi, "Static analysis for dummies: Experiencing lisa," in *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, ser. SOAP 2021, Virtual, Canada: Association for Computing Machinery, 2021, 1–6, ISBN: 9781450384681. DOI: 10.1145/3460946.3464316. [Online]. Available: https://doi.org/10.1145/3460946.3464316.

[8] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1979, pp. 269–282.

[9] P. Cousot and R. Cousot, "Static determination of dynamic properties of programs," in *Proceedings of the Second International Symposium on Programming*, Dunod, Paris, France, 1976, pp. 106–130.

[10] F. Logozzo and M. Fähndrich, "Pentagons: A weakly relational abstract domain for the efficient validation of array accesses," in *Proceedings of the 2008 ACM symposium on Applied computing*, 2008, pp. 184–188.

[11] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1978, pp. 84–96.

[12] A. Miné, "The octagon abstract domain," *Higher-order and symbolic computation*, vol. 19, pp. 31–100, 2006.

[13] K. Ghorbal, F. Ivančić, G. Balakrishnan, N. Maeda, and A. Gupta, "Donut domains: Efficient non-convex domains for abstract interpretation," in *Verification, Model Checking, and Abstract Interpretation: 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings 13*, Springer, 2012, pp. 235–250.

[14] G. Costantini, P. Ferrara, and A. Cortesi, "A suite of abstract domains for static analysis of string values," *Software: Practice and Experience*, vol. 45, no. 2, pp. 245–287, 2015.

[15] M. Madsen and E. Andreasen, "String analysis for dynamic field access," in *International Conference on Compiler Construction*, Springer, 2014, pp. 197–217.

[16] A. Cortesi, G. Costantini, and P. Ferrara, "A survey on product operators in abstract interpretation," *arXiv preprint arXiv:1309.5146*, 2013.

[17] P. Granger, "Improving the results of static analyses of programs by local decreasing iterations," in *International Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer, 1992, pp. 68–79.

[18] A. Cortesi, B. Le Charlier, and P. Van Hentenryck, "Combinations of abstract domains for logic programming: Open product and generic pattern construction," *Science of Computer Programming*, vol. 38, no. 1-3, pp. 27–71, 2000.

[19] R. Giacobazzi and F. Ranzato, "The reduced relative power operation on abstract domains," *Theoretical Computer Science*, vol. 216, no. 1-2, pp. 159–211, 1999.

[20] D. E. Denning, "An intrusion-detection model," *IEEE Transactions on software engineering*, no. 2, pp. 222–232, 1987.

[21] C. Gates and C. Taylor, "Challenging the anomaly detection paradigm: A provocative discussion," in *Proceedings of the 2006 workshop on New security paradigms*, 2006, pp. 21–29.

[22] A. Bovenzi, F. Brancati, S. Russo, and A. Bondavalli, "An os-level framework for anomaly detection in complex software systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 3, pp. 366–372, 2014.

[23] R Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, IEEE, 2000, pp. 144–155.

[24] W. Lee and S. Stolfo, "Data mining approaches for intrusion detection," 1998.

[25] K. Xu, K. Tian, D. Yao, and B. G. Ryder, "A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, 2016, pp. 467–478.

[26] A. P. Kosoresow and S. Hofmeyer, "Intrusion detection via system call traces," *IEEE software*, vol. 14, no. 5, pp. 35–42, 1997.

[27] F. Huch, M. Golagha, A. Petrovska, and A. Krauss, "Machine learning-based run-time anomaly detection in software systems: An industrial evaluation," in *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, IEEE, 2018, pp. 13–18.

[28] G. W. Clark, M. V. Doran, and T. R. Andel, "Cybersecurity issues in robotics," in *2017 IEEE conference on cognitive and computational aspects of situation management (CogSIMA)*, IEEE, 2017, pp. 1–5.

A Logic of Operations

For the purposes of this appendix, suppose the *STB* abstract domain is able to query auxiliary abstract domain A trough the calls $query_A(x, c)$ and $query_A Btw(x, a, b)$, where x is a variable and a, b and c are constants.

$query_A(x, c)$ returns:

- *isEqual*, if A is able to guarantee that property $x = c$ holds at the current statement st;
- *isGreater*, if A is able to guarantee that $x > c$ holds at st;
- *isGreaterOrEq*, if A is able to guarantee that $x \geq c$ holds at st;
- *isLess*, if A is able to guarantee that $x < c$ holds at st;
- *isLessOrEq*, if A is able to guarantee that $x \leq c$ holds at st;
- *isNotEq*, if A is able to guarantee that $x \neq c$ holds at st;
- *unknown*, if A is not able to guarantee that any of the previous properties holds at st.

$query_A Btw(x, a, b)$ returns:

- *isZero*, if A is able to guarantee that property $x = a$ holds at st;
- *isOne*, if A is able to guarantee that property $x = b$ holds at st;
- *isNotZero*, if A is able to guarantee that property $x \neq a$ holds at st;
- *isNotOne*, if A is able to guarantee that property $x \neq b$ holds at st;
- *isBetween*, if A is able to guarantee that both $x > a$ and $x < b$ hold at st;
- *isBetweenOrEq*, if A is able to guarantee that both $x \geq a$ and $x \leq b$ hold at st;
- *isNotBetween*, if A is able to guarantee that either $x < a$ or $x > b$ hold at st;

- *isNotBetweenOrEq*, if A is able to guarantee that either $x \leq a$ or $x \geq b$ hold at st;
- *unknown*, if A is not able to guarantee that any of the previous properties holds at st.

where q is an expression in the form $x\ CMP\ c$, with x variable, c constant and CMP comparison operator. $query_A(q)$ returns true if A is able to guarantee that q holds at the current statement st.

A.1 Addition

If $st : x = x + y$ or $st : x = y + x$ then:

$ans = query_A(y, 0)$
switch *ans* **do**
  case *isEqual*:
    x is **stable**
    break;
  case *isGreater*:
    x is **increasing**
    break;
  case *isLess*:
    x is **decreasing**
    break;
  case *isGreaterOrEq*:
    x is **not decreasing**
    break;
  case *isLessOrEq*:
    x is **not increasing**
    break;
  case *isNotEq*:
    x is **not stable**
    break;
  default:
    x is **top**
    break;

end switch

A.2 Subtraction

If $st : x = x - y$ then:

$ans = query_A(y, 0)$
switch *ans* **do**
  case *isEqual*:
    x is **stable**
    break;
  case *isGreater*:
    x is **decreasing**
    break;

case *isLess*:
x is **increasing**
break;
case *isGreaterOrEq*:
x is **not increasing**
break;
case *isLessOrEq*:
x is **not decreasing**
break;
case *isNotEq*:
x is **not stable**
break;
default:
x is **top**
break;

end switch

A.3 Multiplication

If $st : x = x \times y$ or $st : x = y \times x$:
switch ans_x **do**
case *isEqual*:
x is **stable**
break;
case *isGreater*:
switch ans_y **do**
case *isGreater*:
x is **increasing**
break;
case *isLess*:
x is **decreasing**
break;
case *isEqual*:
x is **stable**
break;
case *isGreaterOrEq*:
x is **not decreasing**
break;
case *isLessOrEq*:
x is **not increasing**
break;
case *isNotEq*:
x is **not stable**
break;
default:
x is **top**
break;

```
    end switch
case isLess:
    switch ans do
        case isEqual:
            x is stable
            break;
        case isGreater:
            x is decreasing
            break;
        case isLess:
            x is increasing
            break;
        case isGreaterOrEq:
            x is not increasing
            break;
        case isLessOrEq:
            x is not decreasing
            break;
        case isNotEq:
            x is not stable
            break;
        default:
            x is top
            break;

    end switch
case isGreaterOrEq:
    switch ans_y do
        case isGreater ∨ isGreaterOrEq:
            x is not decreasing
            break;
        case isLess ∨ isLessOrEq:
            x is not increasing
            break;
        case isEqual:
            x is stable
            break;
        default:
            x is top
            break;

    end switch
case isLessOrEq:
    switch ans_y do
        case isGreater ∨ isGreaterOrEq:
            x is not increasing
            break;
        case isLess ∨ isLessOrEq:
```

x is **not decreasing**
break;
case *isEqual*:
x is **stable**
break;
default:
x is **top**
break;

end switch
case *isNotEq*:
if $ans_y = isNotEq$ **then**
x is **not stable**
end if
default:
x is **top**
break;

end switch

A.4 Division

If $st : x = x \div y$:
$ans_x = query_A(x, 0)$
$ans_y = queryBtw_A(y, 0, 1)$
switch ans_x **do**
case *isEqual*:
x is **stable**
breack;
case *isGreater*:
switch ans_y **do**
case *isZero*:
x is **bottom**
break;
case *isOne*:
x is **stable**
break;
case *isBetween*:
x is **increasing**
breack;
case *isNotBetween*:
x is **decreasing**
breack;
case *isBetweenOrEq*:
x is **not decreasing**
breack;
case *isNotBetweenOrEq*:
x is **not increasing**
breack;

```
            case isNotOne:
                x is not stable
                break;
            default:
                x is top
                break;

        end switch
    case isLess:
        switch ans_y do
            case isZero:
                x is bottom
                break;
            case isOne:
                x is stable
                break;
            case isBetween:
                x is decreasing
                break;
            case isNotBetween:
                x is increasing
                break;
            case isBetweenOrEq:
                x is not increasing
                break;
            case isNotBetweenOrEq:
                x is not decreasing
                break;
            case isNotOne:
                x is not stable
                break;
            default:
                x is top
                break;

        end switch
    case isGreaterOrEq:
        switch ans_y do
            case isZero:
                x is bottom
                break;
            case isOne:
                x is stable
                break;
            case isBetween ∨ isBetweenOrEq:
                x is not decreasing
                break;
            case isNotBetween ∨ isNotBetweenOrEq:
                x is not increasing
```

```
                break;
            default:
                x is top
                break;

        end switch
    case isLessOrEq:
        switch ans_y do
            case isZero:
                x is bottom
                break;
            case isOne:
                x is stable
                break;
            case isBetween ∨ isBetweenOrEq:
                x is not increasing
                break;
            case isNotBetween ∨ isNotBetweenOrEq:
                x is not decreasing
                break;
            default:
                x is top
                break;

        end switch
    case isNotEq:
        if ans_y = isNotOne then
            x is not stable
        end if
    default:
        x is top
        break;

end switch
```

B Trend Class

```
1  public class Trend
2          implements BaseNonRelationalValueDomain<Trend> {
3
4    /** The abstract top element. */
5    public static final Trend TOP = new Trend((byte) 0);
6
7    /** The abstract bottom element. */
8    public static final Trend BOTTOM = new Trend((byte) 1);
9
10   /** The abstract stable element. */
11   public static final Trend STABLE = new Trend((byte) 2);
12
13   /** The abstract increasing element. */
14   public static final Trend INC = new Trend((byte) 3);
15
16   /** The abstract decreasing element. */
17   public static final Trend DEC = new Trend((byte) 4);
18
19   /** The abstract not decreasing element. */
20   public static final Trend NOT_DEC = new Trend((byte) 5);
21
22   /** The abstract not increasing element. */
23   public static final Trend NOT_INC = new Trend((byte) 6);
24
25   /** The abstract not stable element. */
26   public static final Trend NOT_STABLE = new Trend((byte) 7);
27
28   private final byte trend;
29
30
31   public Trend(byte trend){ this.trend = trend; }
32
33   public Trend(){ this.trend = 0; }
34
35   public byte getTrend() { return trend; }
36
37   public boolean isTop(){ return this.trend == (byte) 0; }
38
39   public boolean isBottom(){ return this.trend == (byte) 1; }
```

```
40
41    public boolean isStable(){ return this.trend == (byte) 2; }
42
43    public boolean isInc(){ return this.trend == (byte) 3; }
44
45    public boolean isDec(){ return this.trend == (byte) 4; }
46
47    public boolean isNotDec(){ return this.trend == (byte) 5; }
48
49    public boolean isNotInc(){ return this.trend == (byte) 6; }
50
51    public boolean isNotStable(){ return this.trend == (byte) 7; }
52
53    @Override
54    public Trend lubAux(Trend other) throws SemanticException {
55
56      if (this.lessOrEqual(other)) return other;
57      else if (other.lessOrEqual(this)) return this;
58
59      else if ((this.isStable() && other.isInc())
60              || (this.isInc() && other.isStable()))
61          return NOT_DEC;
62      else if ((this.isStable() && other.isDec())
63              || (this.isDec() && other.isStable()))
64          return NOT_INC;
65      else if ((this.isInc() && other.isDec())
66              || (this.isDec() && other.isInc()))
67          return NOT_STABLE;
68
69      return TOP;
70    }
71
72    @Override
73    public Trend glbAux(Trend other) throws SemanticException {
74
75      if (this.lessOrEqual(other)) return this;
76      else if (other.lessOrEqual(this)) return other;
77
78      else if ((this.isNotDec() && other.isNotStable())
79              || (this.isNotStable() && other.isNotDec()))
80          return INC;
81      else if ((this.isNotInc() && other.isNotStable())
82              || (this.isNotStable() && other.isNotInc()))
83          return DEC;
84      else if ((this.isNotDec() && other.isNotInc())
85              || (this.isNotInc() && other.isNotDec()))
86          return STABLE;
87
88      return BOTTOM;
89    }
90
91    @Override
92    public boolean lessOrEqualAux(Trend other)
93          throws SemanticException {
94      return (this.isStable() &&
95              (other.isNotInc() || other.isNotDec()))
96              || (this.isInc() &&
```

```
97              (other.isNotDec() || other.isNotStable()))
98              || (this.isDec() &&
99              (other.isNotInc() || other.isNotStable()));
100   }
101
102   @Override
103   public Trend top() { return TOP; }
104
105   @Override
106   public Trend bottom() { return BOTTOM; }
107
108   public Trend invert(){
109     if (this.isTop() || this.isBottom()
110             || this.isStable() || this.isNotStable())
111         return this;
112
113     else if (this.isInc()) return DEC;
114     else if (this.isDec()) return INC;
115     else if (this.isNotInc()) return NOT_DEC;
116     else if (this.isNotDec()) return NOT_INC;
117
118     else return BOTTOM;
119   }
120
121   /** Generates Trend accroding to arguments */
122   public static Trend generateTrendIncIfGt(
123         boolean isEqual,
124         boolean isGreater,
125         boolean isGreaterOrEq,
126         boolean isLess,
127         boolean isLessOrEq,
128         boolean isNotEq) {
129
130     if (isEqual) return STABLE;
131     else if (isGreater) return INC;
132     else if (isGreaterOrEq) return NOT_DEC;
133     else if (isLess) return DEC;
134     else if (isLessOrEq) return NOT_INC;
135     else if (isNotEq) return NOT_STABLE;
136
137     else return TOP;
138
139   }
140
141   /** Generates Trend accroding to arguments */
142   public static Trend generateTrendNotDecIfGt(
143         boolean isEqual,
144         boolean isGreater,
145         boolean isGreaterOrEq,
146         boolean isLess,
147         boolean isLessOrEq,
148         boolean isNotEq) {
149
150     if (isEqual) return STABLE;
151     else if (isGreater || isGreaterOrEq) return NOT_DEC;
152     else if (isLess || isLessOrEq) return NOT_INC;
153
```

```
154       else return TOP;
155     }
156
157     /** Generates Trend accroding to arguments */
158     public static Trend generateTrendIncIfBetween(
159           boolean isEqualA,
160           boolean isGreaterA,
161           boolean isGreaterOrEqA,
162           boolean isLessA,
163           boolean isLessOrEqA,
164           boolean isNotEqA,
165           boolean isEqualB,
166           boolean isGreaterB,
167           boolean isGreaterOrEqB,
168           boolean isLessB,
169           boolean isLessOrEqB,
170           boolean isNotEqB) {
171
172       if (isEqualA || isEqualB) return STABLE;
173       else if (isGreaterA && isLessB) return INC;
174       else if ((isGreaterA || isGreaterOrEqA)
175               && (isLessB || isLessOrEqB)) return NOT_DEC;
176       else if (isLessA || isGreaterB) return DEC;
177       else if (isLessOrEqA || isGreaterOrEqB) return NOT_INC;
178       else if (isNotEqA && isNotEqB) return NOT_STABLE;
179
180       else return TOP;
181     }
182
183     /** Generates Trend accroding to arguments */
184     public static Trend generateTrendNotDecIfBetween(
185           boolean isEqualA,
186           boolean isGreaterA,
187           boolean isGreaterOrEqA,
188           boolean isLessA,
189           boolean isLessOrEqA,
190           boolean isNotEqA,
191           boolean isEqualB,
192           boolean isGreaterB,
193           boolean isGreaterOrEqB,
194           boolean isLessB,
195           boolean isLessOrEqB,
196           boolean isNotEqB) {
197
198       if (isEqualA || isEqualB) return STABLE;
199       else if ((isGreaterA || isGreaterOrEqA)
200               && (isLessB || isLessOrEqB)) return NOT_DEC;
201       else if (isLessA || isLessOrEqA || isGreaterB || isGreaterOrEqB) return NO
202
203       else return TOP;
204     }
205
206     @Override
207     public StructuredRepresentation representation() {
208       if (isBottom())
209           return Lattice.bottomRepresentation();
210       if (isTop())
```

```
211             return Lattice.topRepresentation();
212
213         String repr;
214         if (this.isStable())
215             repr = "stable";
216         else if (this.isInc())
217             repr = "increasing";
218         else if (this.isDec())
219             repr = "decreasing";
220         else if (this.isNotDec())
221             repr = "not␣decreasing";
222         else if (this.isNotInc())
223             repr = "not␣increasing";
224         else
225             repr = "not␣stable";
226
227         return new StringRepresentation(repr);
228     }
229
230     @Override
231     public boolean equals(Object obj) {
232         if (this == obj)
233             return true;
234         if (obj == null)
235             return false;
236         if (getClass() != obj.getClass())
237             return false;
238         Trend other = (Trend) obj;
239         return this.getTrend() == other.getTrend();
240     }
241
242     /** Applies the combine operator to this and post */
243     public Trend combine(Trend post){
244         if (post.isBottom() || post.isStable())
245             return new Trend(this.getTrend());
246         else if (this.isBottom() || this.isStable())
247             return new Trend(post.getTrend());
248         else if (this.isNotStable() || post.isNotStable())
249             return new Trend((byte) 0); //TOP
250         else if (this.equals(post))
251             return new Trend(this.getTrend());
252
253         else if((this.isInc() && post.isNotDec())
254                 || (this.isDec() && post.isNotInc()))
255             return new Trend(this.getTrend());
256
257         if ((post.isInc() && this.isNotDec())
258                 || (post.isDec() && this.isNotInc()))
259             return new Trend(post.getTrend());
260
261         return new Trend((byte) 0); //TOP
262     }
263
264 }
```

C Stability Class

```
1  public class Stability<V extends BaseNonRelationalValueDomain<V>>
2          implements BaseLattice<Stability<V>>,
3          ValueDomain<Stability<V>> {
4
5      private final ValueEnvironment<V> auxiliaryDomain;
6
7      private final ValueEnvironment<Trend> trend;
8
9      public Stability(ValueEnvironment<V> auxiliaryDomain) {
10         this.auxiliaryDomain = auxiliaryDomain;
11         this.trend = new ValueEnvironment<>(new Trend((byte)2));
12     }
13
14     public Stability(ValueEnvironment<V> auxiliaryDomain,
15             ValueEnvironment<Trend> trend) {
16         this.auxiliaryDomain = auxiliaryDomain;
17         this.trend = trend;
18     }
19
20     @Override
21     public Stability<V> lubAux(Stability<V> other)
22             throws SemanticException {
23
24         ValueEnvironment<V> ad =
25             auxiliaryDomain.lub(other.getAuxiliaryDomain());
26         ValueEnvironment<Trend> t = trend.lub(other.getTrend());
27
28         if (ad.isBottom() || t.isBottom()) return bottom();
29         else return new Stability<>(ad, t);
30     }
31
32     @Override
33     public Stability<V> glbAux(Stability<V> other)
34             throws SemanticException {
35
36         ValueEnvironment<V> ad =
37             auxiliaryDomain.glb(other.getAuxiliaryDomain());
38         ValueEnvironment<Trend> t = trend.glb(other.getTrend());
```

```
39
40          if (ad.isBottom() || t.isBottom()) return bottom();
41          else return new Stability<>(ad, t);
42      }
43
44      @Override
45      public Stability<V> wideningAux(Stability<V> other)
46              throws SemanticException {
47
48          ValueEnvironment<V> ad =
49              auxiliaryDomain.widening(
50                  other.getAuxiliaryDomain());
51          ValueEnvironment<Trend> t =
52              trend.widening(other.getTrend());
53
54          if (ad.isBottom() || t.isBottom()) return bottom();
55          else return new Stability<>(ad, t);
56      }
57
58      @Override
59      public boolean lessOrEqualAux(Stability<V> other)
60              throws SemanticException {
61
62          return (
63              getAuxiliaryDomain().
64              lessOrEqual(other.getAuxiliaryDomain())
65              && getTrend().lessOrEqual(other.getTrend()));
66      }
67
68      @Override
69      public boolean isTop() {
70          return (auxiliaryDomain.isTop() && trend.isTop());
71      }
72
73      @Override
74      public boolean isBottom() {
75          return (
76              auxiliaryDomain.isBottom()
77              && trend.isBottom());
78      }
79
80      @Override
81      public Stability<V> top() {
82          return new Stability<>(
83              auxiliaryDomain.top(),
84              trend.top());
85      }
86
87      @Override
88      public Stability<V> bottom() {
89          return new Stability<>(
90              auxiliaryDomain.bottom(),
91              trend.bottom());
92      }
93
94      @Override
95      public Stability<V> pushScope(ScopeToken token)
```

```
96              throws SemanticException {
97          return new Stability<>(
98                  auxiliaryDomain.pushScope(token),
99                  trend.pushScope(token));
100     }
101
102     @Override
103     public Stability<V> popScope(ScopeToken token)
104             throws SemanticException {
105         return new Stability<>(
106             auxiliaryDomain.popScope(token),
107             trend.popScope(token));
108     }
109
110     /** Verifies if query is satisfied in V */
111     private boolean query(
112             BinaryExpression query,
113             ProgramPoint pp,
114             SemanticOracle oracle)
115             throws SemanticException {
116
117         return auxiliaryDomain.satisfies(query, pp, oracle)
118             == Satisfiability.SATISFIED;
119     }
120
121     /** Builds BinaryExpression "l operator r" */
122     private BinaryExpression binary(
123             BinaryOperator operator,
124             SymbolicExpression l,
125             SymbolicExpression r,
126             ProgramPoint pp){
127
128         return new BinaryExpression(
129                 pp.getProgram().getTypes().getBooleanType(),
130                 l,
131                 r,
132                 operator,
133                 SyntheticLocation.INSTANCE);
134     }
135
136     /** Builds Constant with value c */
137     private Constant constantInt(int c, ProgramPoint pp){
138         return new Constant(
139                 pp.getProgram().getTypes().getIntegerType(),
140                 c,
141                 SyntheticLocation.INSTANCE
142         );
143     }
144
145     /** Auxiliary logic method */
146     private Trend increasingIfGreater(
147             SymbolicExpression a,
148             SymbolicExpression b,
149             ProgramPoint pp, SemanticOracle oracle)
150             throws SemanticException {
151
152         return Trend.generateTrendIncIfGt(
```

```
153             query(binary(ComparisonEq.INSTANCE, a, b, pp),
154                 pp, oracle),
155             query(binary(ComparisonGt.INSTANCE, a, b, pp),
156                 pp, oracle),
157             query(binary(ComparisonGe.INSTANCE, a, b, pp),
158                 pp, oracle),
159             query(binary(ComparisonLt.INSTANCE, a, b, pp),
160                 pp, oracle),
161             query(binary(ComparisonLe.INSTANCE, a, b, pp),
162                 pp, oracle),
163             query(binary(ComparisonNe.INSTANCE, a, b, pp),
164                 pp, oracle)
165         );
166     }
167
168     /** Auxiliary logic method */
169     private Trend increasingIfLess(
170             SymbolicExpression a,
171             SymbolicExpression b,
172             ProgramPoint pp, SemanticOracle oracle)
173             throws SemanticException {
174         return increasingIfGreater(a, b, pp, oracle).invert();
175     }
176
177     /** Auxiliary logic method */
178     private Trend nonDecreasingIfGreater(
179             SymbolicExpression a,
180             SymbolicExpression b,
181             ProgramPoint pp, SemanticOracle oracle)
182             throws SemanticException {
183
184         return Trend.generateTrendNotDecIfGt(
185             query(binary(ComparisonEq.INSTANCE, a, b, pp),
186                 pp, oracle),
187             query(binary(ComparisonGt.INSTANCE, a, b, pp),
188                 pp, oracle),
189             query(binary(ComparisonGe.INSTANCE, a, b, pp),
190                 pp, oracle),
191             query(binary(ComparisonLt.INSTANCE, a, b, pp),
192                 pp, oracle),
193             query(binary(ComparisonLe.INSTANCE, a, b, pp),
194                 pp, oracle),
195             query(binary(ComparisonNe.INSTANCE, a, b, pp),
196                 pp, oracle)
197         );
198     }
199
200     /** Auxiliary logic method */
201     private Trend nonDecreasingIfLess(
202             SymbolicExpression a,
203             SymbolicExpression b,
204             ProgramPoint pp, SemanticOracle oracle)
205             throws SemanticException {
206
207         return nonDecreasingIfGreater(a, b, pp, oracle)
208             .invert();
209     }
```

```
210
211     /** Auxiliary logic method */
212     private Trend increasingIfBetweenZeroAndOne(
213             SymbolicExpression a,
214             ProgramPoint pp, SemanticOracle oracle)
215             throws SemanticException {
216
217         Constant zero = constantInt(0, pp);
218         Constant one = constantInt(1, pp);
219
220         return Trend.generateTrendIncIfBetween(
221             false,
222             query(binary(ComparisonGe.INSTANCE, a, zero, pp),
223                 pp, oracle),// Gt -> Ge
224             query(binary(ComparisonGe.INSTANCE, a, zero, pp),
225                 pp, oracle),
226             query(binary(ComparisonLe.INSTANCE, a, zero, pp),
227                 pp, oracle),
228             query(binary(ComparisonLe.INSTANCE, a, zero, pp),
229                 pp, oracle), // Lt -> Le
230             query(binary(ComparisonNe.INSTANCE, a, zero, pp),
231                 pp, oracle),
232             query(binary(ComparisonEq.INSTANCE, a, one, pp),
233                 pp, oracle),
234             query(binary(ComparisonGt.INSTANCE, a, one, pp),
235                 pp, oracle),
236             query(binary(ComparisonGe.INSTANCE, a, one, pp),
237                 pp, oracle),
238             query(binary(ComparisonLt.INSTANCE, a, one, pp),
239                 pp, oracle),
240             query(binary(ComparisonLe.INSTANCE, a, one, pp),
241                 pp, oracle),
242             query(binary(ComparisonNe.INSTANCE, a, one, pp),
243                 pp, oracle)
244         );
245     }
246
247     /** Auxiliary logic method */
248     private Trend increasingIfOutsideZeroAndOne(
249             SymbolicExpression a,
250             ProgramPoint pp, SemanticOracle oracle)
251             throws SemanticException{
252
253         return increasingIfBetweenZeroAndOne(a, pp, oracle)
254             .invert();
255     }
256
257     /** Auxiliary logic method */
258     private Trend nonDecreasingIfBetweenZeroAndOne(
259             SymbolicExpression a,
260
261                 ProgramPoint pp, SemanticOracle oracle)
262             throws SemanticException{
263
264         Constant zero = constantInt(0, pp);
265         Constant one = constantInt(1, pp);
266
```

```
267         return Trend.generateTrendNotDecIfBetween(
268             false,
269             query(binary(ComparisonGe.INSTANCE, a, zero, pp),
270                 pp, oracle),  // Gt -> Ge
271             query(binary(ComparisonGe.INSTANCE, a, zero, pp),
272                 pp, oracle),
273             query(binary(ComparisonLe.INSTANCE, a, zero, pp),
274                 pp, oracle),  // Lt -> Le
275             query(binary(ComparisonLe.INSTANCE, a, zero, pp),
276                 pp, oracle),
277             query(binary(ComparisonNe.INSTANCE, a, zero, pp),
278                 pp, oracle),
279
280             query(binary(ComparisonEq.INSTANCE, a, one, pp),
281                 pp, oracle),
282             query(binary(ComparisonGt.INSTANCE, a, one, pp),
283                 pp, oracle),
284             query(binary(ComparisonGe.INSTANCE, a, one, pp),
285                 pp, oracle),
286             query(binary(ComparisonLt.INSTANCE, a, one, pp),
287                 pp, oracle),
288             query(binary(ComparisonLe.INSTANCE, a, one, pp),
289                 pp, oracle),
290             query(binary(ComparisonNe.INSTANCE, a, one, pp),
291                 pp, oracle)
292         );
293
294     }
295
296     /** Auxiliary logic method */
297     private Trend nonDecreasingIfOutsideZeroAndOne(
298         SymbolicExpression a,
299         ProgramPoint pp, SemanticOracle oracle)
300             throws SemanticException{
301
302         return nonDecreasingIfBetweenZeroAndOne(a, pp, oracle)
303             .invert();
304     }
305
306
307     @Override
308     public Stability<V> assign(
309             Identifier id,
310             ValueExpression expression,
311             ProgramPoint pp, SemanticOracle oracle)
312             throws SemanticException {
313
314         if (!trend.knowsIdentifier(id))
315             return new Stability<>(
316                 auxiliaryDomain.assign(id, expression,
317                         pp, oracle),
318                 trend.putState(id, Trend.STABLE));
319
320         if (this.isBottom()
321                 || auxiliaryDomain.isBottom()
322                 || trend.isBottom())
323             return bottom();
```

```
324
325         Trend returnTrend = Trend.TOP;
326
327         if ((expression instanceof Constant))
328           returnTrend =
329               increasingIfLess(id, expression, pp, oracle);
330
331         if (expression instanceof UnaryExpression
332               && ((UnaryExpression) expression).getOperator()
333               instanceof NumericNegation)
334           returnTrend =
335               increasingIfLess(id, expression, pp, oracle);
336
337         else if (expression instanceof BinaryExpression) {
338           BinaryExpression be = (BinaryExpression) expression;
339           BinaryOperator op = be.getOperator();
340           SymbolicExpression left = be.getLeft();
341           SymbolicExpression right = be.getRight();
342
343           boolean isLeft = id.equals(left);
344           boolean isRight = id.equals(right);
345
346           // x = a / 0
347           if (op instanceof DivisionOperator
348                 && query(binary(ComparisonEq.INSTANCE,
349                         right,
350                         constantInt(0, pp),
351                         pp), pp, oracle))
352             return bottom();
353
354           if (isLeft || isRight) {
355             SymbolicExpression other = isLeft ? right : left;
356
357             // x = x + other || x = other + x
358             if (op instanceof AdditionOperator)
359               returnTrend = increasingIfGreater(
360                       other,
361                       constantInt(0, pp),
362                       pp, oracle);
363
364             // x = x - other
365             else if (op instanceof SubtractionOperator) {
366               if (isLeft)
367                   returnTrend = increasingIfLess(
368                       other,
369                       constantInt(0, pp),
370                       pp, oracle);
371               else
372                   returnTrend = increasingIfLess(
373                       id,
374                       expression,
375                       pp, oracle);
376             }
377
378             // x = x * other || x = other * x
379             else if (op instanceof MultiplicationOperator) {
380
```

```
381         // id == 0 || other == 1
382         if (query(binary(
383                 ComparisonEq.INSTANCE,
384                 id,
385                 constantInt(0, pp),
386                 pp), pp, oracle) || query(binary(
387                 ComparisonEq.INSTANCE,
388                 other,
389                 constantInt(1, pp),
390                 pp), pp, oracle))
391             returnTrend = Trend.STABLE;
392
393         // id > 0
394         else if (query(binary(
395                 ComparisonGt.INSTANCE,
396                 id,
397                 constantInt(0, pp),
398                 pp), pp, oracle))
399             returnTrend = increasingIfGreater(
400                 other, constantInt(1, pp), pp, oracle);
401
402         // id < 0
403         else if (query(binary(
404                 ComparisonLt.INSTANCE,
405                 id,
406                 constantInt(0, pp),
407                 pp), pp, oracle))
408             returnTrend = increasingIfLess(
409                 other, constantInt(1, pp), pp, oracle);
410
411         // id >= 0
412         else if (query(binary(
413                 ComparisonGe.INSTANCE,
414                 id,
415                 constantInt(0, pp),
416                 pp), pp, oracle))
417             returnTrend = nonDecreasingIfGreater(
418                 other, constantInt(1, pp), pp, oracle);
419
420         // id <= 0
421         else if (query(binary(
422                 ComparisonLe.INSTANCE,
423                 id,
424                 constantInt(0, pp),
425                 pp), pp, oracle))
426             returnTrend = nonDecreasingIfLess(
427                 other, constantInt(1, pp), pp, oracle);
428
429         // id != 0 && other != 1
430         else if (query(binary(
431                 ComparisonNe.INSTANCE,
432                 id,
433                 constantInt(0, pp),
434                 pp), pp, oracle) && query(binary(
435                 ComparisonNe.INSTANCE,
436                 other, constantInt(1, pp),
437                 pp), pp, oracle))
```

```
438             returnTrend = Trend.NOT_STABLE;
439
440         //else returnTrend = Trend.TOP;
441       }
442
443       // x = x / other
444       else if (op instanceof DivisionOperator){
445         if (isLeft) {
446
447             // id == 0 || other == 1
448             if (query(binary(
449                     ComparisonEq.INSTANCE,
450                     id,
451                     constantInt(0, pp),
452                     pp), pp, oracle) || query(binary(
453                     ComparisonEq.INSTANCE,
454                     other,
455                     constantInt(1, pp),
456                     pp), pp, oracle))
457                 returnTrend = Trend.STABLE;
458
459             // id > 0
460             else if (query(binary(
461                     ComparisonGt.INSTANCE,
462                     id,
463                     constantInt(0, pp),
464                     pp), pp, oracle))
465                 returnTrend = increasingIfBetweenZeroAndOne(
466                     other, pp, oracle);
467
468             // id < 0
469             else if (query(binary(
470                     ComparisonLt.INSTANCE,
471                     id,
472                     constantInt(0, pp),
473                     pp), pp, oracle))
474                 returnTrend = increasingIfOutsideZeroAndOne(
475                     other, pp, oracle);
476
477             // id >= 0
478             else if (query(binary(
479                     ComparisonGe.INSTANCE,
480                     id,
481                     constantInt(0, pp),
482                     pp), pp, oracle))
483                 returnTrend = nonDecreasingIfBetweenZeroAndOne(
484                     other, pp, oracle);
485
486             // id <= 0
487             else if (query(binary(
488                     ComparisonLe.INSTANCE,
489                     id,
490                     constantInt(0, pp),
491                     pp), pp, oracle))
492                 returnTrend = nonDecreasingIfOutsideZeroAndOne(
493                     other, pp, oracle);
494
```

```
495                 // id != 0 && other != 1
496                 else if (query(binary(
497                       ComparisonNe.INSTANCE,
498                       id,
499                       constantInt(0, pp),
500                       pp), pp, oracle)
501                       && query(binary(
502                       ComparisonNe.INSTANCE,
503                       other,
504                       constantInt(1, pp),
505                       pp), pp, oracle))
506                   returnTrend = Trend.NOT_STABLE;
507
508               } // end isLeft branch
509
510               else returnTrend = increasingIfLess(
511                   id, expression, pp, oracle);
512
513             } // end division branch
514
515         } // end isLeft || isRight branch
516         else returnTrend = increasingIfLess(
517           id, expression, pp, oracle);
518       }
519
520       ValueEnvironment<V> ad = auxiliaryDomain.assign(
521               id, expression, pp, oracle);
522       ValueEnvironment<Trend> t = trend.putState(
523               id, returnTrend);
524
525       if (ad.isBottom() || t.isBottom()) return bottom();
526       else return new Stability<>(ad, t);
527   }
528
529   @Override
530   public Stability<V> smallStepSemantics(
531           ValueExpression expression,
532           ProgramPoint pp,
533           SemanticOracle oracle)
534           throws SemanticException {
535
536       ValueEnvironment<V> ad =
537               auxiliaryDomain.smallStepSemantics(
538                       expression, pp, oracle);
539
540       if (ad.isBottom()) return bottom();
541       else return new Stability<>(ad, trend);
542   }
543
544   @Override
545   public Stability<V> assume(
546           ValueExpression expression,
547           ProgramPoint src,
548           ProgramPoint dest,
549           SemanticOracle oracle)
550           throws SemanticException {
551
```

```
552         ValueEnvironment<V> ad = auxiliaryDomain.assume(
553             expression, src, dest, oracle);
554         ValueEnvironment<Trend> t = trend.assume(
555             expression, src, dest, oracle);
556         if (ad.isBottom() || t.isBottom()) return bottom();
557         else return new Stability<>(ad, t);
558     }
559
560     @Override
561     public boolean knowsIdentifier(Identifier id) {
562         return (auxiliaryDomain.knowsIdentifier(id)
563                 || trend.knowsIdentifier(id));
564     }
565
566     @Override
567     public Stability<V> forgetIdentifier(Identifier id)
568             throws SemanticException {
569         return new Stability<>(
570                 auxiliaryDomain.forgetIdentifier(id),
571                 trend.forgetIdentifier(id));
572     }
573
574     @Override
575     public Stability<V> forgetIdentifiersIf(
576             Predicate<Identifier> test)
577             throws SemanticException {
578         return new Stability<>(
579                 auxiliaryDomain.forgetIdentifiersIf(test),
580                 trend.forgetIdentifiersIf(test));
581     }
582
583     @Override
584     public Satisfiability satisfies(
585             ValueExpression expression,
586             ProgramPoint pp,
587             SemanticOracle oracle)
588             throws SemanticException {
589         return Satisfiability.UNKNOWN;
590     }
591
592     @Override
593     public StructuredRepresentation representation() {
594         return new ListRepresentation(
595                 auxiliaryDomain.representation(),
596                 trend.representation());
597     }
598
599     @Override
600     public boolean equals(Object obj) {
601         if (this == obj)
602             return true;
603         if (obj == null)
604             return false;
605         if (getClass() != obj.getClass())
606             return false;
607         Stability<V> other = (Stability<V>) obj;
608         return this.auxiliaryDomain
```

```
609                 .equals(other.auxiliaryDomain)
610                 && this.trend.equals(other.trend);
611
612     }
613
614     @Override
615     public int hashCode() { return 0; }
616
617     @Override
618     public String toString() {
619         return auxiliaryDomain.representation().toString()
620                 + trend.representation().toString();
621     }
622
623     public ValueEnvironment<Trend> getTrend() { return trend; }
624
625     public ValueEnvironment<V> getAuxiliaryDomain() {
626         return auxiliaryDomain;
627     }
628
629     /** combines two {@code Stability} */
630     public Stability<V> environmentCombine(Stability<V> post){
631         ValueEnvironment<Trend> retTrendEnv =
632                 new ValueEnvironment<>(new Trend((byte) 0));
633
634         for (Identifier id : post.getTrend().getKeys()) {
635             if (this.getTrend().knowsIdentifier(id)) {
636                 Trend tmp = this.getTrend().getState(id)
637                         .combine(post.getTrend().getState(id));
638                 retTrendEnv = retTrendEnv.putState(id, tmp);
639             }
640             else
641                 retTrendEnv = retTrendEnv.putState(
642                     id, post.getTrend().getState(id));
643         }
644
645         return new Stability<>(
646                 post.getAuxiliaryDomain(), retTrendEnv);
647     }
648
649 }
```

D CoContraVarianceCheck Class

```
1  private static class CoContraVarianceCheck<
2          T extends BaseNonRelationalValueDomain<T>>
3          implements SemanticCheck<SimpleAbstractState<
4                  MonolithicHeap,
5                  Stability<T>,
6                  TypeEnvironment<InferredTypes>>> {
7
8      Map<Statement, Stability<T>> preStatesMap =
9              new HashMap<>();
10     Map<Statement, Stability<T>> resultsMap =
11             new LinkedHashMap<>();
12
13     @Override
14     public boolean visit(
15             CheckToolWithAnalysisResults<
16                 SimpleAbstractState<
17                     MonolithicHeap,
18                     Stability<T>,
19                     TypeEnvironment<InferredTypes>>>
20             tool,
21             CFG graph,
22             Statement node) {
23
24         if (graph.containsNode(node)) {
25             for (AnalyzedCFG<
26                     SimpleAbstractState<MonolithicHeap,
27                     Stability<T>,
28                     TypeEnvironment<InferredTypes>>>
29                     result : tool.getResultOf(graph)) {
30
31                 Stability<T> postState =
32                     result.getAnalysisStateAfter(node).
33                     getState().getValueState();
34
35                 Stability<T> cumulativeState = postState;
36
```

```
37                // computing cumulative trend for node
38                if (preStatesMap.containsKey(node)) {
39                  Stability<T> preState =
40                          preStatesMap.get(node);
41                  cumulativeState =
42                          preState.environmentCombine(postState);
43                }
44                resultsMap.put(node, cumulativeState);
45
46                // computing new entry state for next
47                for (Statement next : graph.followersOf(node)) {
48                  if (preStatesMap.containsKey(next)) {
49                      // join converging branches
50                      try {
51                          preStatesMap.put(
52                                  next,
53                                  preStatesMap.get(next)
54                                  .lub(cumulativeState));
55                      } catch (SemanticException e) {
56                          e.printStackTrace();
57                      }
58                  } else
59                      preStatesMap.put(next, cumulativeState);
60                      // the pre of next is the post of this
61                }
62             } // end for each
63          }
64          return true;
65      }
66  }
```

E Running the Analysis

```
1  @Test
2  public void stabiltiyTest()
3          throws ParsingException, AnalysisException {
4
5      /**
6      * parse the program at "filePath" to get the
7      * CFG representation of the code in it
8      */
9      Program program = IMPFrontend.processFile("filePath");
10
11     // build a new configuration for the analysis
12     LiSAConfiguration conf = new DefaultConfiguration();
13
14     // specify where files are generated
15     conf.workdir = "output/stability";
16
17     // specify the visual format of the analysis results
18     conf.analysisGraphs =
19             LiSAConfiguration.GraphType.HTML;
20
21     // specify the analysis to execute
22     conf.abstractState = new SimpleAbstractState<>(
23             // heap domain
24             new MonolithicHeap(),
25             // value domain
26             new Stability<>(
27                 new ValueEnvironment<>(new Interval())
28                 .top()),
29             // type domain
30             new TypeEnvironment<>(new InferredTypes()));
31
32     // instantiate LiSA with our configuration
33     LiSA lisa = new LiSA(conf);
34
35     // tell LiSA to analyze the program
36     lisa.run(program);
37
38 }
```

```
1  @Test
2  public void correlationTest() throws ParsingException {
3
4      Program program = IMPFrontend.processFile("filePath");
5      LiSAConfiguration conf = new DefaultConfiguration();
6      conf.workdir = "output/correlation";
7      conf.abstractState = new SimpleAbstractState<>(
8              new FieldSensitivePointBasedHeap(),
9              new Stability<>(
10                 new ValueEnvironment<>(new Interval())
11                 .top()),
12             new TypeEnvironment<>(new InferredTypes()));
13     conf.interproceduralAnalysis =
14             new ContextBasedAnalysis<>();
15
16     conf.semanticChecks.add(
17             new CoContraVarianceCheck<Interval>());
18
19     LiSA lisa = new LiSA(conf);
20     lisa.run(program);
21 }
```