Stability: An Abstract Domain for the Trend of Variation of Numerical Variables

Luca Negrini
Ca' Foscari University of Venice
Venice, Italy
luca.negrini@unive.it

Sofia Presotto
Ca' Foscari University of Venice
Venice, Italy
870762@stud.unive.it

Pietro Ferrara
Ca' Foscari University of Venice
Venice, Italy
pietro.ferrara@unive.it

Enea Zaffanella
University of Parma
Parma, Italy
enea.zaffanella@unipr.it

Agostino Cortesi
Ca' Foscari University of Venice
Venice, Italy
cortesi@unive.it

Abstract

State-of-the-art abstract domains for static analysis typically focus on over-approximating either the values a variable can hold at a specific program point or the relational dependencies among variables. In this paper, we aim to capture the trend of numerical values during program executions (e.g., increasing, decreasing, or stable) relative to preceding states. By integrating the Stability domain with numerical domains, we can verify co-/contra-variance relationships among potentially unrelated variables. This approach has promising applications for anomaly detection in complex software systems, and for the verification of functional requirements.

CCS Concepts: **• Theory of computation → Program analysis**; **Abstraction**; **• Software and its engineering → Automated static analysis**.

Keywords: Static Analysis, Abstract Interpretation, Numeric Domains

ACM Reference Format:
Luca Negrini, Sofia Presotto, Pietro Ferrara, Enea Zaffanella, and Agostino Cortesi. 2024. Stability: An Abstract Domain for the Trend of Variation of Numerical Variables. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains (NSAD '24), October 22, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3689609.3689995

1 Introduction

Static analysis is a technique for inspecting program properties without *concretely* executing the program. Examples of these properties may be whether the program terminates, which program variables are constants, or whether a program contains safety and security issues (e.g., buffer overflows [17] or injection vulnerabilities [38]).

For static analysis to *guarantee* the presence (or absence) of code properties, bugs, and vulnerabilities, one must adopt an approach based on a formal methods framework. Among these frameworks, a notable example is certainly abstract interpretation [10, 11]. Abstract interpretation is a theoretical framework that provides a systematic way to correctly approximate program behaviors, referred to as *concrete* semantics, and reason about some properties of the program of interest on such approximation, called *abstract* semantics. The abstraction process is carried out by an *abstract domain*, that models the *concrete* values manipulated by the program and their evolution during all possible executions.

Several abstract domains have been proposed over the years, each with a different cost-to-precision trade-off and targeting different kinds of values and properties: numeric values [11, 16, 23, 25], string values [4, 7, 27], types [8], dependencies [5, 19], and many more. Numerical abstractions have been the first to be studied, as proving properties over them is pivotal in safety-critical contexts. Starting from the *non-relational* domain of intervals [11], that computes independent ranges for each variable, the abstractions can be made more powerful by also tracking inter-variable relations (thus taking the name of *relational* abstractions): pentagons [23] (modeling relations such as $x < y$), octagons [25, 36] (considering relations of the form $\pm x \pm y \leq c \in \mathbb{Q}$) or convex polyhedra [3, 16], track increasingly complex relations between variables in addition to their possible values. Generally speaking, relational abstract domains offer higher precision than non-relational ones, but they are also more complex to define and require additional computational efforts to track and maintain the relationships between program variables.

The problem. The abstract domains mentioned above all share a common idea: to track the values of each variable, and possibly the relations over them. Dealing with values in a *sound* way entails being conservative about statically unknown values (e.g., as user inputs). Consider for instance

NSAD '24, October 22, 2024, Pasadena, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1217-3/24/10
https://doi.org/10.1145/3689609.3689995

the following snippet of Solidity code, where function send transfers money between two accounts:

```
 1  contract Coin {
 2    mapping (address => uint) public balances;
 3    // [...]
 4    function send(address dest, uint amount) public {
 5      require(amount > 0);
 6      require(amount <= balances[msg.sender]);
 7      balances[msg.sender] -= amount;
 8      balances[dest] += amount;
 9    }
10  }
```

Lines 3 and 4 check some preconditions over the values of `amount` and `balances[msg.sender]` (where `msg.sender` is a Solidity global variable containing the address of the account that started the transaction). If those are successful, a certain user-given amount of money is transferred from the sender's account to the receiver's. Since `amount`, `balances[msg.sender]` and `balances[dest]` can have any value when function `send` is executed, traditional numeric analyses are not able to infer properties regarding any of the three quantities. However, it should be possible to prove that, after the execution of `send`, `balances[msg.sender]` has decreased and `balances[dest]` has increased (note that `requires` aborts the execution if the given condition does not hold). Such properties are *functional requirements* that express the expected behavior of programs, and often contain conditions over their inputs. Requirements can be arbitrarily complex, and can target a wide range of program behaviors.

Contribution. In this paper, we target requirements that concern the *evolution* of numerical values during a program execution, such as the one introduced in the previous paragraph. We proceed by:

- defining an abstract domain for numerical stability[1], able to the detect the trend of each variable's value (e.g., increasing or stable);
- showing its effectiveness on two simple programs.

Paper structure. Section 2 recalls basic notions of order theory and abstract interpretation. Section 3 defines a minimalistic imperative language Imp that we use during formalization. Section 4 defines the novel domain for numerical stability, together with its abstract semantics. Section 5 applies the proposed domain to two target programs, explaining in detail how it operates. Finally, Section 6 presents related work and Section 7 concludes.

2 Background

Order Theory. A partial order is a reflexive, transitive and antisymmetric binary relation. A set L with a partial ordering relation $\sqsubseteq\, \subseteq L \times L$ is a poset $\langle L, \sqsubseteq \rangle$. A poset $\langle L, \sqsubseteq, \bot, \sqcup \rangle$, where $\sqcup$ is the least upper bound (lub) operator of L and $\bot$ is the least element (bottom) of L, is a complete partial order (CPO) if $\forall x, y \in L$ we have that $x \sqcup y$ belongs to L. Furthermore, a poset $\langle L, \sqsubseteq, \sqcup, \sqcap \rangle$, where $\sqcup$ and $\sqcap$ are respectively the least upper bound (lub) and greatest lower bound (glb) operators of L, is a lattice if $\forall x, y \in L$ we have that $x \sqcup y$ and $x \sqcap y$ belong to L. We say that a lattice is complete when $\forall X \subseteq L$ we have that $\bigsqcup X, \bigsqcap X \in L$. Any finite lattice is a complete lattice. A complete lattice L with greatest element (top) $\top$ and least element (bottom) $\bot$ is denoted by $\langle L, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$.

Abstract Interpretation. Abstract interpretation [11, 13] is a theory to soundly approximate program semantics, focusing on some run-time property of interest. The concrete and the abstract semantics are defined over two CPOs,[2] respectively called the concrete domain C and abstract domain A. Let C and A be CPOs, a pair of monotone functions $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$ forms a *Galois Connection* (GC) between C and A if for every $x \in C$ and for every $y \in A$ we have $\alpha(x) \sqsubseteq_A y \Leftrightarrow x \sqsubseteq_C \gamma(y)$. We denote a Galois Connection by $\langle C, \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \sqsubseteq_A \rangle$. A GC between two complete lattices A and C can be induced if α is a complete join preserving map, i.e., $\alpha(\bigsqcup_C X) = \bigsqcup_A \{\alpha(x) \mid x \in X\}$, with $X \subseteq C$ (Prop. 7 of [14]). Given $\langle C, \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \sqsubseteq_A \rangle$, a concrete function $f : C \rightarrow C$ is, in general, not computable. Hence, an abstract function $f^\sharp : A \rightarrow A$ must correctly approximate the concrete function f. If so, we say that $f^\sharp$ is sound. Formally, given $\langle C, \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \sqsubseteq_A \rangle$ and a concrete function $f : C \rightarrow C$, an abstract function $f^\sharp : A \rightarrow A$ is sound w.r.t. f if $\forall c \in C. \alpha(f(c)) \sqsubseteq_A f^\sharp(\alpha(c))$.

3 The Imp Language

We introduce here a generic imperative language as a reference programming language for the rest of the paper. We consider the core running language Imp, whose syntax is given in Fig. 1. Imp is an imperative language handling arithmetic expressions.[3] Its basic values are integers ranging over $\mathbb{Z}$. Let P be an Imp program. Each Imp statement is annotated with a label $\ell \in Lab_P$ (not belonging to the syntax), where Lab_P denotes the set of the P labels, i.e., its program points. Moreover, $Vars_P$ denotes the variables defined by P.

As usual in static analysis, a program can be analyzed by looking at its control-flow graph (CFG for short), i.e., a directed graph that embeds the control structure of a program, where nodes are the program points, and edges express the flow paths from the entry to the exit block. Following [37], given a program P $\in$ Imp, we define the corresponding CFG $G_P \triangleq \langle Nodes_P, Edges_P, In_P, Out_P \rangle$ as the CFG whose nodes are

[1] In this context, rather than the classical notion of "numeric stability" in floating-point computations, *stability* refers to evolution of variables' values: for instance, we deem a variable as stable if its value does not change, or unstable if its value increases and decreases repeatedly.

[2] While CPOs are the minimum requirement for the abstract interpretation framework, (possibly complete) lattices are typically used.

[3] Division is excluded from Imp since its abstract semantics would not provide any additional contribution, but would add unnecessary complexity.

$$
\begin{array}{l}
\texttt{a} \in \text{AE} ::= x \mid n \mid \texttt{a + a} \mid \texttt{a - a} \mid \texttt{a * a} \\
\texttt{b} \in \text{BE} ::= x \mid \texttt{true} \mid \texttt{false} \mid \texttt{b \&\& b} \mid \texttt{b || b} \\
\qquad \mid \texttt{! b} \mid \texttt{e < e} \mid \texttt{e} \leq \texttt{e} \mid \texttt{e > e} \mid \texttt{e} \geq \texttt{e} \mid \\
\qquad \texttt{e == e} \mid \texttt{e != e} \\
\texttt{e} \in \text{E} ::= \texttt{a} \mid \texttt{b} \\
\texttt{st} \in \text{STMT} ::= {}^{\ell_1}\texttt{st}\ {}^{\ell_2}\texttt{st}^{\ell_3} \mid {}^{\ell_1}\texttt{skip;}^{\ell_2} \mid {}^{\ell_1}x\ \texttt{=}\ \texttt{a;}^{\ell_2} \\
\quad \mid {}^{\ell_1}\texttt{if (b) \{}\ {}^{\ell_2}\texttt{st}^{\ell_3}\ \texttt{\} else \{}\ {}^{\ell_4}\texttt{st}^{\ell_5}\ \texttt{\}}^{\ell_6} \\
\quad \mid {}^{\ell_1}\texttt{while (b) \{}\ {}^{\ell_2}\texttt{st}^{\ell_3}\ \texttt{\}}^{\ell_4} \\
\quad \mid {}^{\ell_1}\texttt{input(}x\texttt{);}^{\ell_2} \\
\texttt{P} \in \text{IMP} ::= {}^{\ell_1}\texttt{st}\ {}^{\ell_2} \\
\qquad \text{where } x \in \mathit{Vars}_{\texttt{P}} \text{ (finite set of variables)}, n \in \mathbb{Z}
\end{array}
$$

Figure 1. Imp syntax.

the program points, i.e., $\mathit{Nodes}_{\texttt{P}} \triangleq \mathit{Lab}_{\texttt{P}}$, $\mathit{In}_{\texttt{P}}$ is the entry program point, and $\mathit{Out}_{\texttt{P}}$ is the last program point. The algorithm computing the CFG of a program P can be found in [1, 37]. An example of a CFG is depicted in Fig. 2.

A CFG embeds the control structure of the program. Thus, to define the behavior of a CFG, it is enough to formalize the semantics of the edge labels, namely $\text{Imp}^{\text{CFG}} ::= \texttt{skip} \mid x = \texttt{a} \mid \texttt{b} \mid \texttt{input}(x)$, expressing the effect that each edge has from its entry node to its exit node. Let $m \in \mathbb{M} \triangleq X \longrightarrow \mathbb{Z}$ be the set of (finite) memories. The semantics of arithmetic expressions is captured by the function $(\!|\ \texttt{e}\ |\!) : \mathbb{M} \longrightarrow \mathbb{Z}$. The semantics of integer and Boolean expressions are standard. Abusing the notation, we define the function $(\!|\ \texttt{st}\ |\!) : \mathbb{M} \longrightarrow \mathbb{M}$ to capture the semantics of the elements of Imp^{CFG} (that also include Boolean expressions):

$$(\!|\ \texttt{skip}\ |\!)m = m \qquad (\!|\ \texttt{input}(x)\ |\!)m = m[x \leftarrow \text{read}()]$$

$$(\!|\ \texttt{b}\ |\!)m = \begin{cases} m & \text{if } (\!|\ \texttt{b}\ |\!)m = \texttt{true} \\ \bot & \text{if } (\!|\ \texttt{b}\ |\!)m = \texttt{false} \end{cases}$$

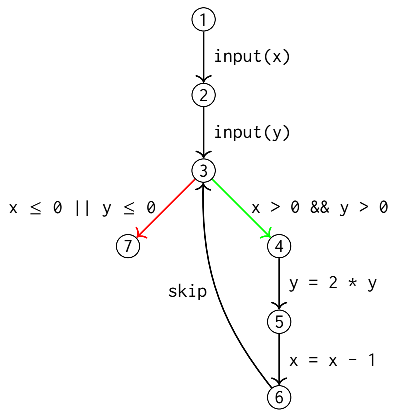
$$(\!|\ x = \texttt{a}\ |\!)m = m[x \leftarrow (\!|\ \texttt{a}\ |\!)m]$$

`skip` propagates the input memory, while `input(`x`)` reads an input value and assigns it to x. Boolean expressions propagate the input memory if the expression holds, or produce an invalid memory ($\bot$) otherwise, preventing the execution of the following statement. Finally, $x = \texttt{a}$ evaluates a in m and updates the latter with the new value for x.

Static analysis computes invariants for each program point. Following [9], we define an operational semantics which relates each program point (i.e., each node of a CFG) to the set of the possible traces reaching that program point. Given a program P and a program label ℓ, our semantics captures all the prefix traces ending in ℓ and it is captured by the least fixpoint of the function $\wr\ \ell\ \int : \wp(\mathbb{T}) \longrightarrow \wp(\mathbb{T})$, where $\mathbb{T}$ corresponds to the set of (finite) prefix traces of the program, i.e., $\mathbb{T} \triangleq \bigcup_{n \in \mathbb{N}} \mathbb{M} \times (\mathit{Lab}_{\texttt{P}} \times \mathbb{M})^n$. We represent a generic trace $\tau \in \mathbb{T}$ as $\sigma_0 \xrightarrow{\ell_1} \sigma_1 \xrightarrow{\ell_2} \ldots \xrightarrow{\ell_{n-1}} \sigma_{n-1} \xrightarrow{\ell_n} \sigma_n$.

```
ℓ1 input(x);
ℓ2 input(y);
ℓ3 while (x > 0
      && y > 0) {
  ℓ4 y = 2 * y;
  ℓ5 x = x - 1;
ℓ6 }ℓ7
```

(a) Imp code.



(b) Corresponding CFG.

Figure 2. Example of CFG.

For instance, let us consider the example reported in Fig. 2a, and let the initial memory for the program execution be $m_{init} \triangleq (x \mapsto 0, y \mapsto 0)$. Supposing that the user inputs 1 and 4 at program points ℓ_1 and ℓ_2, respectively, a possible concrete trace reaching the program point ℓ_5 is:

$$(x \mapsto 0, y \mapsto 0) \xrightarrow{\ell_1} (x \mapsto 1, y \mapsto 0) \xrightarrow{\ell_2} (x \mapsto 1, y \mapsto 4)$$
$$\xrightarrow{\ell_3} (x \mapsto 1, y \mapsto 4) \xrightarrow{\ell_4} (x \mapsto 1, y \mapsto 8)$$

Note that, this definition slightly extends the maximal trace semantics defined in Sect. 5 of [9], enriching the definition of the transition relation with the label corresponding to the computational step.

4 The Stability Abstract Domain

The goal of the stability abstract domain is to infer, for each program label, if a variable is increasing, decreasing or stable w.r.t. the previous label. We define such domain as a functional lift from program variables to elements of the complete lattice of per-variable *trends*:

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

A variable can thus increase strictly ($\uparrow$) or non-strictly ($\uparrow=$), can decrease strictly ($\downarrow$) or non-strictly ($\downarrow=$), can remain stable ($=$), can become unstable ($\neq$, the value is definitely different but nothing more is known), or can have an unknown ($\top$) or erroneous ($\bot$, produced by error-raising computations) trend. Least upper bound $\sqcup_{\text{Trn}}$, greatest lower bound $\sqcap_{\text{Trn}}$, and partial order operator $\sqsubseteq_{\text{Trn}}$ definitions can be derived from the Hasse diagram of Trn reported in Fig. 3.

The stability abstract domain tracks an element of Trn for each variable through functional lifting [13]:

$$\text{Stb} \triangleq \langle \mathit{Vars}_{\texttt{P}} \longrightarrow \text{Trn}, \sqsubseteq_{\text{Stb}}, \sqcup_{\text{Stb}}, \sqcap_{\text{Stb}}, \bot_{\text{Stb}}, \top_{\text{Stb}} \rangle,$$

where the lattice operators are defined as element-wise applications of the operators of Trn. The bottom element $\bot_{\text{Stb}}$

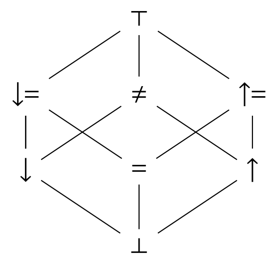


Figure 3. The Hasse diagram of Trn.

is the empty map,[4] while the top element $\top_{\textsc{Stb}}$ is the map where every variable is mapped to $\top \in$ Trn.

Example. If we were to analyze the snippet of Imp code in Figure 2a using Stb, we would obtain a different Stb instance at each location ℓ_i. For instance, at ℓ_2 we would obtain the mapping $\{\mathtt{x} \mapsto =\}$ since x has just been defined. Instead, at ℓ_6 Stb would compute the mapping $\{\mathtt{x} \mapsto \downarrow, \mathtt{y} \mapsto =\}$ since y is not modified by the assignment, while x has been decreased.

Abstraction and concretization. As required by the framework of abstract interpretation, we define an abstraction and a concretization function for Stb. The abstraction function $\alpha_{\textsc{Stb}} : \wp(\mathbb{T}) \longrightarrow$ Stb, mapping sets of prefix traces to instances of Stb, is defined the lub of the abstraction of individual traces:

$$\alpha_{\textsc{Stb}}(T) \triangleq \bigsqcup_{\tau \in T} \left\{ x \mapsto st(\tau, x) \;\middle|\; x \in Vars_{\mathrm{P}} \right\}$$

$\alpha_{\textsc{Stb}}$ uses the auxiliary function $st : \mathbb{T} \times Vars_P \longrightarrow$ Trn, where $\tau = \ldots \sigma_{n-1} \xrightarrow{\ell} \sigma_n$, extracting the trend of a single variable:

$$st(\tau, x) \triangleq \begin{cases} = & \text{if } \tau = m_{init} \vee \sigma_{n-1}(x) = \sigma_n(x) \\ \uparrow & \text{if } \sigma_{n-1}(x) < \sigma_n(x) \\ \downarrow & \text{if } \sigma_{n-1}(x) > \sigma_n(x) \end{cases}$$

where we denote, abusing notation, as $\sigma(x)$ the value of variable x in the state σ. The abstraction function thus joins the abstraction of each individual trace, that is, the trend of each variable at the end of the trace w.r.t. the previous step.

Having defined $\alpha_{\textsc{Stb}}$ as a union (and thus intuitively as a join preserving function), we employ Prop. 7 of [14] to define the concretization function $\gamma_{\textsc{Stb}} : \textsc{Stb} \longrightarrow \wp(\mathbb{T})$ as $\gamma_{\textsc{Stb}}(s) = \bigcup\{T \in \wp(\mathbb{T}) \mid \alpha_{\textsc{Stb}}(T) \sqsubseteq_{\textsc{Stb}} s\}$, inducing the Galois connection $\langle \wp(\mathbb{T}), \subseteq \rangle \underset{\alpha_{\textsc{Stb}}}{\overset{\gamma_{\textsc{Stb}}}{\leftrightarrows}} \langle \textsc{Stb}, \sqsubseteq_{\textsc{Stb}} \rangle$.

4.1 Stb Is Not Enough For Stability

Stability information cannot be inferred from any previous knowledge of stability itself. In general, determining the trend of a variable after a statement requires information provided by some numerical domain. This becomes evident when considering instructions such as `x = x + y`. To deduce the the trend of x after the assignment, one needs to know the sign of y, as its value might be, e.g., increasing even if it is negative. Similarly, with `x = 5` information about the previous value of x is needed to compute the trend of x after the assignment. In both cases, trend information on single variables is irrelevant to reaching a conclusion.

In fact, to track stability through a program we require an auxiliary numerical abstract domain. For this reason, we define the abstract transformers for Stb w.r.t. an *open product* [6] between Stb and a generic numerical domain $\mathcal{A}$. Intuitively, an open product is a form direct product (Chapter 36 of [10]) between abstract domains where they can interact through *abstract queries* in a domain-independent fashion. This allows Stb to query information about values of variables from $\mathcal{A}$ modularly, without tailoring its definition to a fixed auxiliary domain. We refer to such product as $\textsc{Stb} \otimes \mathcal{A}$, and we express queries from Stb to $\mathcal{A}$ using a function $Q : \mathcal{A} \times \textsc{be} \longrightarrow \{\mathtt{true}, \mathtt{false}\}$ (where the instance of $\mathcal{A}$ used for the query will be a subscript) that given a Boolean expression b yields `true` if b always holds.

Note that the choice of such a domain leads to different levels of precision: if one uses the Sign abstract domain in the second example (`x = 5`), knowing that `x > 0` does not provide enough information to deduce the trend of x.

4.2 Abstract Semantics

We present here the abstract semantics of the elements of $\textsc{Imp}^{\textsc{cfg}}$, followed by the semantics of arithmetic expressions.[5] The semantics is defined on the open product $\textsc{Stb} \otimes \mathcal{A}$. Regarding $\mathcal{A}$, we assume that its semantics is defined over abstract states $d^\sharp$, and that it provides (i) abstract transformers for all statements in $\textsc{Imp}^{\textsc{cfg}}$, (ii) an abstract evaluation function for Boolean expressions $\wr\, \mathrm{b} \,\wr_{\mathcal{A}} d^\sharp = \wp(\{\mathtt{true}, \mathtt{false}\})$, and (iii) an (optional) state refinement operator $\lfloor d^\sharp \rfloor_{\mathrm{b}}$ to be applied when traversing a condition b.

Let $m^\sharp = (s^\sharp, d^\sharp) \in \mathbb{M}^\sharp \triangleq \textsc{Stb} \otimes \mathcal{A}$ be the set of abstract memories, modeled through instances of the open product between Stb and $\mathcal{A}$. The abstract semantics of expressions over Stb is the identity function (intuitively, stability information only changes when a variable changes its value), and is thus omitted. Figure 4 defines the function $\wr\, \mathrm{st} \,\wr : \mathbb{M}^\sharp \longrightarrow \mathbb{M}^\sharp$ to capture the semantics of the elements of $\textsc{Imp}^{\textsc{cfg}}$. `skip` propagates the input memory, while the semantics of Boolean expressions depends on the semantics of $\mathcal{A}$: it returns the error memory $\bot$ if it never holds, otherwise it refines $\mathcal{A}$ using its state refining operator. x = a and `input(`x`)` perform similarly, since they are the two Imp instructions changing the value of a variable. We distinguish two cases: the definition of a new variable ($x \notin dom(s^\sharp)$), and the assignment to an existing variable. In the former case, Stb records that the variable is stable since there is no pre-existing value. In the

[4]As usual, given $s \in \textsc{Stb}$, if $\exists x \in dom(s) \,.\, s(x) = \bot$ then $s \equiv \bot_{\textsc{Stb}}$, that is s is normalized to the bottom element.

[5]We do not define the semantics of Boolean expressions since they are a no-op for Stb, and the evaluation just updates $\mathcal{A}$.

$$\llparenthesis\, \mathsf{b} \,\rrparenthesis m^\sharp = \begin{cases} \bot & \text{if } \llparenthesis\, \mathsf{b} \,\rrparenthesis_{\mathcal{A}} m^\sharp = \texttt{false} \\ (s^\sharp, \lfloor d^\sharp \rfloor_{\mathsf{b}}) & \text{otherwise} \end{cases}$$

$$\llparenthesis\, \texttt{input}(x) \,\rrparenthesis m^\sharp = (s_1^\sharp, \llparenthesis\, \texttt{input}(x) \,\rrparenthesis_{\mathcal{A}} d^\sharp) \quad \llparenthesis\, \texttt{skip} \,\rrparenthesis m^\sharp = m^\sharp$$

$$\llparenthesis\, x = \mathsf{a} \,\rrparenthesis m^\sharp = (s_2^\sharp, d_1^\sharp = \llparenthesis\, x = \mathsf{a} \,\rrparenthesis_{\mathcal{A}} d^\sharp)$$

$$s_1^\sharp = \begin{cases} s^\sharp[x \leftarrow =] & \text{if } x \notin dom(s^\sharp) \\ s^\sharp[x \leftarrow \top] & \text{otherwise} \end{cases}$$

$$s_2^\sharp = \begin{cases} s^\sharp[x \leftarrow =] & \text{if } x \notin dom(s^\sharp) \\ s^\sharp[x \leftarrow \amalg(x, \mathsf{a}, d^\sharp, d_1^\sharp)] & \text{otherwise} \end{cases}$$

Figure 4. Abstract semantics of Stb $\otimes \mathcal{A}$.

latter case instead, it updates the value for x according to its evolution. In the case of $\texttt{input}(x)$, since we do not have information on the new value, x is updated to $\top$. On assignments, x is updated using function $\amalg$ using information in $d_1^\sharp$. Function $\amalg : Vars_{\mathrm{P}} \times \mathrm{AE} \times \mathcal{A} \times \mathcal{A} \to \mathsf{Trn}$ produces a new trend for a variable exploiting information in $\mathcal{A}$. We split the definition of $\amalg$ depending on the contents of the right-hand side of the assignment. In the following, we denote as e a generic expression that is independent from the variable x being assigned. Definitions of $\amalg$ should be interpreted as lists of successive queries to $\mathcal{A}$, reading the left-most column first: the result of $\amalg$ is the trend corresponding to the first satisfied query. We omit the case where all queries return `false` and $\amalg$ returns $\top$.

Independent assignment. Let $d_2^\sharp = \llparenthesis\, x = e \,\rrparenthesis_{\mathcal{A}}(\llparenthesis\, \bar{x} = x \,\rrparenthesis_{\mathcal{A}} d^\sharp)$. $d_2^\sharp$ contains information on the previous value of x stored in $\bar{x}$. We define $\amalg$ as:

$$\amalg(x, \mathsf{e}, d^\sharp, d_1^\sharp) = \begin{cases} \uparrow \text{ if } Q_{d_2^\sharp}(\bar{x} < x) & \upharpoonleft\!= \text{ if } Q_{d_2^\sharp}(\bar{x} \le x) \\ \downarrow \text{ if } Q_{d_2^\sharp}(\bar{x} > x) & \downharpoonleft\!= \text{ if } Q_{d_2^\sharp}(\bar{x} \ge x) \\ = \text{ if } Q_{d_2^\sharp}(\bar{x} == x) & \ne \text{ if } Q_{d_2^\sharp}(\bar{x} \,!= x) \end{cases}$$

Addition. When $\mathsf{a} = x + \mathsf{e}$ (or equivalently $\mathsf{a} = \mathsf{e} + x$), we can define $\amalg$ simply by inspecting the sign of e:

$$\amalg(x, x+\mathsf{e}, d^\sharp, d_1^\sharp) = \begin{cases} \uparrow \text{ if } Q_{d_1^\sharp}(\mathsf{e} > 0) & \upharpoonleft\!= \text{ if } Q_{d_1^\sharp}(\mathsf{e} \ge 0) \\ \downarrow \text{ if } Q_{d_1^\sharp}(\mathsf{e} < 0) & \downharpoonleft\!= \text{ if } Q_{d_1^\sharp}(\mathsf{e} \le 0) \\ = \text{ if } Q_{d_1^\sharp}(\mathsf{e} == 0) & \ne \text{ if } Q_{d_1^\sharp}(\mathsf{e} \,!= 0) \end{cases}$$

Subtraction. When $\mathsf{a} = x - \mathsf{e}$ (or equivalently $\mathsf{a} = \mathsf{e} - x$), we can define $\amalg$ by inspecting the sign of e:

$$\amalg(x, x-\mathsf{e}, d^\sharp, d_1^\sharp) = \begin{cases} \uparrow \text{ if } Q_{d_1^\sharp}(\mathsf{e} < 0) & \upharpoonleft\!= \text{ if } Q_{d_1^\sharp}(\mathsf{e} \le 0) \\ \downarrow \text{ if } Q_{d_1^\sharp}(\mathsf{e} > 0) & \downharpoonleft\!= \text{ if } Q_{d_1^\sharp}(\mathsf{e} \ge 0) \\ = \text{ if } Q_{d_1^\sharp}(\mathsf{e} == 0) & \ne \text{ if } Q_{d_1^\sharp}(\mathsf{e} \,!= 0) \end{cases}$$

Table 1. Chaining operator for Trn instances.

t_1 \ t_2	$\bot$	$\upharpoonleft\!=$	$\uparrow$	$=$	$\ne$	$\downarrow$	$\downharpoonleft\!=$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\upharpoonleft\!=$	$\bot$	$\upharpoonleft\!=$	$\uparrow$	$\upharpoonleft\!=$	$\top$	$\top$	$\top$	$\top$
$\uparrow$	$\bot$	$\uparrow$	$\uparrow$	$\uparrow$	$\top$	$\top$	$\top$	$\top$
$=$	$\bot$	$\upharpoonleft\!=$	$\uparrow$	$=$	$\ne$	$\downarrow$	$\downharpoonleft\!=$	$\top$
$\ne$	$\bot$	$\top$	$\top$	$\ne$	$\top$	$\top$	$\top$	$\top$
$\downarrow$	$\bot$	$\top$	$\top$	$\downarrow$	$\top$	$\downarrow$	$\downarrow$	$\top$
$\downharpoonleft\!=$	$\bot$	$\top$	$\top$	$\downharpoonleft\!=$	$\top$	$\downarrow$	$\downharpoonleft\!=$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Multiplication. When $\mathsf{a} = x * \mathsf{e}$ (or equivalently $\mathsf{a} = \mathsf{e} * x$), $\amalg$ becomes more intricate: it not only depends on the sign of x and e, but also on the magnitude of e. We thus define $\amalg$ by inspecting the sign of x and by comparing e to 1:

$$\amalg(x, x*\mathsf{e}, d^\sharp, d_1^\sharp) = \begin{cases} = & \text{if } Q_{d_1^\sharp}(x == 0 \,||\, \mathsf{e} == 1) \\ \downarrow & \text{if } Q_{d_1^\sharp}(x < 0 \,\&\&\, \mathsf{e} > 1) \vee \\ & \quad Q_{d_1^\sharp}(x > 0 \,\&\&\, \mathsf{e} < 1) \\ \uparrow & \text{if } Q_{d_1^\sharp}(x < 0 \,\&\&\, \mathsf{e} < 1) \vee \\ & \quad Q_{d_1^\sharp}(x > 0 \,\&\&\, \mathsf{e} > 1) \\ \upharpoonleft\!= & \text{if } Q_{d_1^\sharp}(x \le 0 \,\&\&\, \mathsf{e} \le 1) \vee \\ & \quad Q_{d_1^\sharp}(x \ge 0 \,\&\&\, \mathsf{e} \ge 1) \\ \downharpoonleft\!= & \text{if } Q_{d_1^\sharp}(x \le 0 \,\&\&\, \mathsf{e} \ge 1) \vee \\ & \quad Q_{d_1^\sharp}(x \ge 0 \,\&\&\, \mathsf{e} \le 1) \\ \ne & \text{if } Q_{d_1^\sharp}(x \,!= 0 \,\&\&\, \mathsf{e}!= 1) \end{cases}$$

4.3 Chaining Stability Information

Stb computes relations between values before and after the execution of *a single statement*. This can be extended to a relation before and after the execution of a *block of statements*, over a whole function, or over the complete program. Trends between two arbitrary points ℓ_i and ℓ_j in a function can be derived by iteratively combining the per-instruction trends. We define the binary operator $\diamond : \mathsf{Trn} \times \mathsf{Trn} \to \mathsf{Trn}$ that, given $t_1, t_2 \in \mathsf{Trn}$, yields the sequential combination of the two. The definition is given by Table 1, where rows and columns are the possible values for t_1 and t_2, respectively.

Intuitively, $\bot$ and $\top$ values are preserved by $\diamond$, and $\top$ is also introduced whenever t_1 and t_2 disagree on the direction of the trend (i.e., when only one of them is $\ne$ or when $t_1 \in \{\uparrow, \upharpoonleft\!=\}$ and $t_2 \in \{\downarrow, \downharpoonleft\!=\}$, or vice versa). Instead, when they agree in direction, the strongest (i.e., lower level on the lattice structure) takes precedence, while $=$ is the neutral element (i.e., the other trend is returned). Note that using $\diamond$ instead of $\sqcup_{\mathsf{Trn}}$ leads to higher precision: for instance, if the same variable has trends $\uparrow$ and $\upharpoonleft\!=$ in two consecutive instructions, $\sqcup_{\mathsf{Trn}}$ would yield $\upharpoonleft\!=$ as an overall trend. Instead, thanks to $\diamond$, we are able to use ↑ as overall trend, since the final value is surely greater than the starting one.

The combination of complete Stb instances is achieved by element-wise application of $\diamond$: we denote such operation as $\dot{\diamond}$. Moreover, given a chain of Stb abstract states $s_1, s_2, \ldots, s_n$ produced at successive program locations $\ell_1, \ell_2, \ldots, \ell_n$, their combination is simply the iterative forward application of $\dot{\diamond}$ (that is, $s_1 \dot{\diamond} s_2 \dot{\diamond} \ldots \dot{\diamond} s_n$). Finally, when merging chained stability information over different paths (e.g., at a loop exit), the chained stability is computed using $\sqcup_{\text{Stb}}$.

4.4 Covariance and Contravariance

When combining Stb instances of a whole function, we obtain the overall trend of a variable from its definition to the function exit. While the trend itself is non-relational in nature, it can be compared with the ones of other variables in the same function to infer *covariance* or *contravariance* relations. In fact, it is possible to show that x and y in Figure 2a are contravariant inside the loop body:

- at ℓ_5, the state computed by Stb is $\{\texttt{x} \mapsto =, \texttt{y} \mapsto \uparrow\}$;
- at ℓ_6, the state computed by Stb is $\{\texttt{x} \mapsto \downarrow, \texttt{y} \mapsto =\}$;
- applying the combination rules, we obtain a new Stb instance covering the per-variable trends of the loop body containing $\{\texttt{x} \mapsto \downarrow, \texttt{y} \mapsto \uparrow\}$, showing that the variables are contravariant.

Note that, on this specific code, Stb can infer the contravariance relation also using the weakest auxiliary domain $\mathcal{A}$ possible, that is, the Sign domain.

5 Experiments

In this section, we demonstrate the capabilities of Stb step-by-step on two sample functions: a simplified Imp form of function send discussed in the introduction, and the Imp code of Figure 2a. The results presented in this section have been produced by a proof-of-concept implementation of Stb in LiSA [20, 28],[6] a Java library for building abstract-interpretation based static analyzers [29–34]. Specifically, our implementation targets programs written in IMP, a toy language that LiSA uses for testing and demonstration, that closely resembles the Imp language used in this paper.

We report here the abstract Stb states produced by the analysis in blue. Each state can be read as both the pre-state before the abstract execution of the next statement, and as the post-state produced by the abstract execution of the preceeding statement. Note that the state reported at the beginning of loop bodies is the least upper bound of the state preceeding the loop with the state after the last statement of the body. The last line contains the combination of the Stb instances from the start to the end of the function, by applying the rules of Section 4.3, in red. In both cases, the code is analyzed using the Sign domain as $\mathcal{A}$.

[6] https://github.com/lisa-analyzer/lisa/tree/master/lisa/lisa-analyses/src/main/java/it/unive/lisa/analysis/stability

```
 1  ≀{}≀
 2  input(dest);
 3  ≀{dest ↦ =}≀
 4  input(amount);
 5  ≀{dest ↦ =, amount ↦ =}≀
 6  input(sbalance);
 7  ≀{dest ↦ =, amount ↦ =, sbalance ↦ =}≀
 8  input(dbalance);
 9  ≀{dest ↦ =, amount ↦ =, sbalance ↦ =, dbalance ↦ =}≀
10  if (amount > 0 && amount <= sbalance) {
11    ≀{dest ↦ =, amount ↦ =, sbalance ↦ =, dbalance ↦ =}≀
12    sbalance = sbalance - amount;
13    ≀{dest ↦ =, amount ↦ =, sbalance ↦ ↓, dbalance ↦ =}≀
14    dbalance = dbalance + amount;
15    ≀{dest ↦ =, amount ↦ =, sbalance ↦ =, dbalance ↦ ↑}≀
16  } else {
17    ≀{dest ↦ =, amount ↦ =, sbalance ↦ =, dbalance ↦ =}≀
18    skip;
19    ≀{dest ↦ =, amount ↦ =, sbalance ↦ =, dbalance ↦ =}≀
20  }
21  ≀{dest ↦ =, amount ↦ =, sbalance ↦ =, dbalance ↦ =}≀
22  ≀{dest ↦ =, amount ↦ =, sbalance ↦ ↓=, dbalance ↦ ↑=}≀
```

Figure 5. Analysis of the Imp version of function send.

The* send *function. Figure 5 reports the Imp version of function send from the introduction. Specifically, we model statically unknown values (the dest account, the amount of money to be transferred, and the sender and receiver balances sbalance and dbalance) as user inputs. Moreover, since Imp does not have require statements, the Boolean predicates are encoded as an if condition guarding the execution of the balances updates. The execution of the first four inputs maps all variables to $\top_{\text{Sign}}$ by Sign, and to = by Stb. Sign infers that amount is always positive inside the true branch of the if, allowing the semantics of addition and subtraction to infer that sbalance strictly decreases after line 12, and dbalance strictly increases after line 14.

When chaining trends, Stb infers that sbalance is non-increasing (↓=) and dbalance is non-decreasing (↑=). This is due to the join happening after the if, where the false branch did not modify any variables. Such an imprecision is due to our instrumentation of the code: if Imp allowed halting, we sould be able to remove this spurious branch to closely resemble the Solidity code, and the balances would be correctly inferred to be decreasing (↓) and increasing (↑), respectively, showing contravariance between the two.

The* Imp *example. The code of the Imp example from Figure 2a, together with the states computed by Stb, is visible in Figure 6. The execution of the first two input maps x and y to $\top_{\text{Sign}}$ by Sign, and to = by Stb. The condition of the while loop enables Sign to determine that x and y are both positive, and in turn to the semantics of addition and subtraction of Stb to infer that y strictly increases after line 8, and x strictly decreases after line 10.

When chaining trends, Stb labels x as non-increasing (↓=) and y as non-decreasing (↑=) after the loop. Note that the chaining operation at the end of the loop body yields

```
⦃{}⦄
input(x);
⦃{x ↦ =}⦄
input(y);
⦃{x ↦ =, y ↦ =}⦄
while (x > 0 && y > 0) {
  ⦃{x ↦ ↓=, y ↦ =}⦄
  y = 2 * y;
  ⦃{x ↦ =, y ↦ ↑}⦄
  x = x - 1;
  ⦃{x ↦ ↓, y ↦ =}⦄
}
⦃{x ↦ =, y ↦ =}⦄
⦃{x ↦ ↓=, y ↦ ↑=}⦄
```

Figure 6. Analysis of the Imp example from Figure 2a.

$\{x \mapsto \downarrow, y \mapsto \uparrow\}$: knowing that x always decreases at each iteration is enough to prove that the loop will eventually terminate. To the best of our knowledge, this result would not be possible with any other numerical domain.[7]

6 Related Work

Static analyses for numerical properties using (weakly) relational abstract domains are sometimes used to track the relations between the program variables and other values that are not directly represented in the considered program: for instance, auxiliary (*ghost*) variables can be introduced to track the size of memory allocations or the computational cost of a specific chunk of code; when considering floating point computations [24], auxiliary variables can be used to model the real value semantics so as to try and bound its distance from the floating point semantics. Anyway, these analysis do not directly track the relations between the values at different program points. They might, however, be employed on an SSA translation [22] of the program, enabling comparison between variables at different program points at the cost of increased computational cost.

When implementing interprocedural analyses using relational domains, the relations between the entry and exit values of program variables can be computed and stored in function summaries [12, 21, 35]. These, however, are not really meant to detect numeric variable trends, covariance and contravariance properties; as a consequence, they may fail to provide this kind of information if the relation holding between entry and exit values does not fit the considered abstract domain (e.g., when using a domain of linear constraints and trying to model a nonlinear increment). Similarly, static analyses syntesizing ranking functions for termination [2, 15, 39] relate the value of program variables at the beginning and at the end of a loop body to find an expression that definitely decreases, which can be interpreted as the search for a rather specific numeric trend.

[7]While dedicated termination analyses can prove the loop termination, contravariance would not be inferred. Moreover, termination analyses typically have higher computational requirements w.r.t. the ones of Stb.

The analysis proposed in [18] is meant to compare the semantics of slightly different versions (an original version and a patched one) of some function of interest. This work shares with the current one the need to compute relations between variables defined at different program points; however, it differs from the current proposal in its main goal: proving that the values computed in the two versions of the program are identical, instead of establishing value trends.

Lastly, the domain presented in Section 4.2 of [26] deduces monotonicity of program variables taking into account thread interferences, and also needs information about the sign of expressions. Our analysis is instead able to determine trends beyond simple monotonicity, but shares the approach of querying information from a numerical abstraction.

7 Conclusion

In this paper, we presented a novel abstract domain for inferring trends of numerical variables, able to recover information in situations where even strong relational domains fail. We provided a formalization of the domain and its abstract semantics, and showed the properties it can infer on two sample functions. As this work is still in its initial phase, we plan on exploring several possible applications: anomaly detection, floating-point errors propagation, and validation of functional requirements. On the formal side, we are currently working on soundness proofs and on providing an abstract semantics for more arithmetic operations.

Acknowledgements

Work partially supported by SERICS (PE00000014 - CUP H73C2200089001) under the NRRP MUR program funded by the EU - NGEU, and by iNEST- Interconnected NordEst Innovation Ecosystem funded by PNRR (Mission 4.2, Investment 49 1.5) NextGeneration EU (ECS_00000043 – CUP H43C22000540006).

References

[1] Vincenzo Arceri and Isabella Mastroeni. 2020. A sound abstract interpreter for dynamic code. In *SAC '20*, Chih-Cheng Hung, Tomás Cerný, Dongwan Shin, and Alessio Bechini (Eds.). ACM, 1979–1988. https://doi.org/10.1145/3341105.3373964

[2] Roberto Bagnara, Fred Mesnard, Andrea Pescetti, and Enea Zaffanella. 2012. A new look at the automatic synthesis of linear ranking functions. *Inf. Comput.* 215 (2012), 47–67. https://doi.org/10.1016/J.IC.2012.03.003

[3] Anna Becchi and Enea Zaffanella. 2020. PPLite: Zero-overhead encoding of NNC polyhedra. *Inf. Comput.* 275 (2020), 104620. https://doi.org/10.1016/J.IC.2020.104620

[4] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. 2003. Precise Analysis of String Expressions. In *Static Analysis*, Radhia Cousot (Ed.). Springer Berlin Heidelberg, 1–18.

[5] Ellis Cohen. 1977. Information transmission in computational systems. *SIGOPS Oper. Syst. Rev.* 11, 5 (nov 1977), 133–139. https://doi.org/10.1145/1067625.806556

[6] Agostino Cortesi, Baudouin Le Charlier, and Pascal Van Hentenryck. 2000. Combinations of abstract domains for logic programming:

open product and generic pattern construction. *Science of Computer Programming* 38, 1 (2000), 27–71. https://doi.org/10.1016/S0167-6423(99)00045-3

[7] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. 2015. A suite of abstract domains for static analysis of string values. *Software: Practice and Experience* 45, 2 (2015), 245–287. https://doi.org/10.1002/spe.2218

[8] Patrick Cousot. 1997. Types as abstract interpretations. In *Proceedings of POPL '97* (Paris, France) *(POPL '97)*. ACM, New York, NY, USA, 316–331. https://doi.org/10.1145/263699.263744

[9] Patrick Cousot. 2002. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.* 277, 1-2 (2002), 47–103. https://doi.org/10.1016/S0304-3975(00)00313-3

[10] Patrick Cousot. 2021. *Principles of Abstract Interpretation.* MIT Press.

[11] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL '77*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

[12] Patrick Cousot and Radhia Cousot. 1977. Static Determination of Dynamic Properties of Recursive Procedures. In *IFIP Working Conference on Formal Description of Programming Concepts*, Erich J. Neuhold (Ed.). North-Holland, 237–278.

[13] Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *POPL 1979*, Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen (Eds.). ACM Press, 269–282. https://doi.org/10.1145/567752.567778

[14] Patrick Cousot and Radhia Cousot. 1992. Abstract Interpretation and Application to Logic Programs. *J. Log. Program.* 13, 2&3 (1992), 103–179. https://doi.org/10.1016/0743-1066(92)90030-7

[15] Patrick Cousot and Radhia Cousot. 2012. An abstract interpretation framework for termination. In *POPL '12*, John Field and Michael Hicks (Eds.). ACM, 245–258. https://doi.org/10.1145/2103656.2103687

[16] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL '78*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 84–96. https://doi.org/10.1145/512760.512770

[17] C. Cowan, F. Wagle, Calton Pu, S. Beattie, and J. Walpole. 2000. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *DISCEX '00*, Vol. 2. 119–129 vol.2. https://doi.org/10.1109/DISCEX.2000.821514

[18] David Delmas and Antoine Miné. 2019. Analysis of Software Patches Using Numerical Abstract Interpretation. In *SAS 2019 (LNCS, Vol. 11822)*, Bor-Yuh Evan Chang (Ed.). Springer, 225–246. https://doi.org/10.1007/978-3-030-32304-2_12

[19] Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Ciprian Spiridon, and Fausto Spoto. 2015. Boolean Formulas for the Static Identification of Injection Attacks in Java. In *Logic for Prog., Art. Int., and Reason.*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, 130–145.

[20] Pietro Ferrara, Luca Negrini, Vincenzo Arceri, and Agostino Cortesi. 2021. Static analysis for dummies: experiencing LiSA. In *SOAP 2021*, Lisa Nguyen Quang Do and Caterina Urban (Eds.). ACM, 1–6. https://doi.org/10.1145/3460946.3464316

[21] Denis Gopan and Thomas W. Reps. 2007. Low-Level Library Analysis and Summarization. In *CAV '07 (LNCS, Vol. 4590)*, Werner Damm and Holger Hermanns (Eds.). Springer, 68–81. https://doi.org/10.1007/978-3-540-73368-3_10

[22] Matthieu Lemerre. 2023. SSA Translation Is an Abstract Interpretation. *Proc. ACM Program. Lang.* 7, POPL, Article 65 (jan 2023), 30 pages. https://doi.org/10.1145/3571258

[23] Francesco Logozzo and Manuel Fähndrich. 2010. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Sci. Comput. Program.* 75, 9 (2010), 796–807. https://doi.org/10.1016/j.scico.2009.04.004

[24] Matthieu Martel. 2002. Propagation of Roundoff Errors in Finite Precision Computations: A Semantics Approach. In *ESOP 2002 (LNCS, Vol. 2305)*. Springer, 194–208. https://doi.org/10.1007/3-540-45927-8_14

[25] Antoine Miné. 2006. The octagon abstract domain. *High. Order Symb. Comp.* 19, 1 (2006), 31–100. https://doi.org/10.1007/s10990-006-8609-1

[26] Antoine Miné. 2014. Relational Thread-Modular Static Value Analysis by Abstract Interpretation. In *VMCAI '14*, Kenneth L. McMillan and Xavier Rival (Eds.). Springer Berlin Heidelberg, 39–58.

[27] Luca Negrini, Vincenzo Arceri, Pietro Ferrara, and Agostino Cortesi. 2021. Twinning Automata and Regular Expressions for String Static Analysis. In *Proc. of VMCAI '21 (LNCS, Vol. 12597)*. Springer, 267–290. https://doi.org/10.1007/978-3-030-67067-2_13

[28] Luca Negrini, Pietro Ferrara, Vincenzo Arceri, and Agostino Cortesi. 2023. *LiSA: A Generic Framework for Multilanguage Static Analysis*. Springer Nature Singapore, 19–42. https://doi.org/10.1007/978-981-19-9601-6_2

[29] Luca Negrini, Guruprerana Shabadi, and Caterina Urban. 2023. Static Analysis of Data Transformations in Jupyter Notebooks. In *SOAP '23*. ACM, New York, NY, USA, 8–13. https://doi.org/10.1145/3589250.3596145

[30] Luca Olivieri, Thomas Jensen, Luca Negrini, and Fausto Spoto. 2023. MichelsonLiSA: A Static Analyzer for Tezos. In *2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. 80–85.

[31] Luca Olivieri, Luca Negrini, Vincenzo Arceri, Badaruddin Chachar, Pietro Ferrara, and Agostino Cortesi. 2024. Detection of Phantom Reads in Hyperledger Fabric. *IEEE Access* 12 (2024), 80687–80697. https://doi.org/10.1109/ACCESS.2024.3410019

[32] Luca Olivieri, Luca Negrini, Vincenzo Arceri, Thomas Jensen, and Fausto Spoto. 2024. Design and Implementation of Static Analyses for Tezos Smart Contracts. *Distrib. Ledger Technol.* (jan 2024). https://doi.org/10.1145/3643567

[33] Luca Olivieri, Luca Negrini, Vincenzo Arceri, Fabio Tagliaferro, Pietro Ferrara, Agostino Cortesi, and Fausto Spoto. 2023. Information Flow Analysis for Detecting Non-Determinism in Blockchain. In *ECOOP 2023 (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 23:1–23:25. https://doi.org/10.4230/LIPIcs.ECOOP.2023.23

[34] Luca Olivieri, Fabio Tagliaferro, Vincenzo Arceri, Marco Ruaro, Luca Negrini, Agostino Cortesi, Pietro Ferrara, Fausto Spoto, and Enrico Talin. 2022. Ensuring determinism in blockchain software with GoLiSA: an industrial experience report. In *SOAP '22* (San Diego, CA, USA) *(SOAP 2022)*. ACM, New York, NY, USA, 23–29. https://doi.org/10.1145/3520313.3534658

[35] M Pnueli and Micha Sharir. 1981. Two approaches to interprocedural data flow analysis. *Program flow analysis: theory and applications* (1981), 189–234.

[36] Michael Schwarz and Helmut Seidl. 2023. Octagons Revisited - Elegant Proofs and Simplified Algorithms. In *SAS '23 (LNCS, Vol. 14284)*, Manuel V. Hermenegildo and José F. Morales (Eds.). Springer, 485–507. https://doi.org/10.1007/978-3-031-44245-2_21

[37] Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack. 2012. *Compiler Design - Analysis and Transformation.* Springer.

[38] Fausto Spoto, Elisa Burato, Michael D. Ernst, Pietro Ferrara, Alberto Lovato, Damiano Macedonio, and Ciprian Spiridon. 2019. Static Identification of Injection Attacks in Java. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 18 (jul 2019), 58 pages. https://doi.org/10.1145/3332371

[39] Caterina Urban and Antoine Miné. 2017. Inference of ranking functions for proving temporal properties by abstract interpretation. *Comput. Lang. Syst. Struct.* 47 (2017), 77–103. https://doi.org/10.1016/J.CL.2015.10.001

Received 2024-07-10; accepted 2024-08-18