frontiers | Frontiers in Computer Science

TYPE Original Research
PUBLISHED 20 January 2026
DOI 10.3389/fcomp.2025.1655377

OPEN ACCESS

EDITED BY
Novarun Deb,
University of Calgary, Canada

REVIEWED BY
Mohamed Wiem Mkaouer,
University of Michigan-Flint, MI, United States
Benoît Montagu,
Inria Rennes - Bretagne Atlantique Research Centre, France

*CORRESPONDENCE
Luca Negrini
✉ luca.negrini@unive.it

RECEIVED 27 June 2025
REVISED 11 December 2025
ACCEPTED 23 December 2025
PUBLISHED 20 January 2026

CITATION
Negrini L (2026) Whole-value analysis by abstract interpretation.
Front. Comput. Sci. 7:1655377.
doi: 10.3389/fcomp.2025.1655377

Whole-value analysis by abstract interpretation

Luca Negrini*

Department of Environmental Sciences, Informatics and Statistics, Ca' Foscari University of Venice, Venice, Italy

Value analysis is the task of understanding what concrete values a program might compute for each variable or memory region. Historically, research focused mostly on numerical analysis (i.e., value analysis of programs manipulating numeric values), while string analyses have received wider attention in the last two decades. String analyses present a key challenge: reasoning about strings entails reasoning about integer values either used as arguments to string operations (e.g., evaluating a substring) or returned by string operations (e.g., calculating the length of a string). Traditionally, string analyses were formalized with respect to a specific numeric analysis, usually considering constant values or their possible ranges, tailoring definitions, semantic proofs, and implementations to that particular combination, hence hindering the adoption of the analyses in different contexts. This study presents a modular framework to define *whole-value analyses* (that is, combinations of numeric analyses, string analyses, and possibly other value types computed by a program) by Abstract Interpretation. The framework defines information exchange between the different analyses in the form of abstract constraints, allowing each analysis to perform given only a generic and analysis-independent description of the abstract values computed by other analyses. Adopting such a framework (i) ensures that soundness proofs are still valid when changing the combination of domains used, and (ii) eases implementation and experimentation of different combinations of value analyses, simplifying comparisons between different scientific contributions and augmenting the set of domains an abstract interpreter can use to analyze a program.

KEYWORDS

abstract interpretation, products, program analysis, static analysis, value analysis

1 Introduction

Static analysis allows one to verify properties of computer programs before they are executed. This is important for proving that programs do not behave incorrectly at execution time, leading to a runtime error or computing the wrong results. Static analysis can also provide evidence of illicit information flows, a topic highly appreciated by companies that write software dealing with sensitive data or that is exposed to external users' interaction. To have guarantees about bug and vulnerability discovery, static analysis must go beyond simple and naïve matching of well-known patterns leading to errors and vulnerabilities (so-called *code smells*). Instead, formal methods such as Abstract Interpretation (Cousot and Cousot, 1977; Cousot, 2021) must be used.

Abstract Interpretation is a mathematical framework to *soundly* reason on program semantics. Proving non-trivial properties of such semantics is, in general, undecidable (Rice, 1953). Abstract Interpretation overcomes this by reasoning on a sound over-approximation of the uncomputable real semantics, referred to as concrete,

transforming it into so-called abstract semantics that is instead computable. While approximation recovers computability, it does come with the cost of imprecision, as more executions are considered with respect to the actual ones exhibited by the program. However, thanks to the over-approximation, properties proven to hold for the abstract semantics are guaranteed to hold also for the concrete one. The main idea behind abstract interpretation is to define the concrete semantics as the fixpoint of a monotone function. Such a function can then be abstracted to a simpler one that has to be proven sound. In practice, the monotone function is defined inductively on the syntax offered by the programming language. Such a definition allows the framework (i.e., the formula) to be parametric with respect to the language semantics in a way that does not require proving the correctness of the whole abstraction every time. Instead, researchers define the meaning of each instruction they want to analyze (e.g, by defining its big-step semantics), and they later abstract that meaning with an *abstract domain* that models some of its properties. For instance, in Cousot and Cousot (1977), numeric values were abstracted as intervals (thus preserving their ranges) and the semantics of the language with interval arithmetics.

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 (Cousot and Cousot, 1977; Logozzo and Fähndrich, 2010; Miné, 2006; Cousot and Halbwachs, 1978), string values (Costantini et al., 2015; Negrini et al., 2021; Christensen et al., 2003b), types (Cousot, 1997), dependencies (Ernst et al., 2015; Cohen, 1977), and many more. Numerical abstractions have been the first to be studied, as proving numerical properties is pivotal in safety-critical contexts. More recently, string analysis gained notable traction due to the many uses strings find in programming languages. While some of these domains can perform naturally in a standalone setting (e.g., type inference can produce expression types by only relying on the types of sub-expressions), cooperation between domains is sometimes necessary. Consider, for instance, the following snippet of Go code:

```
start := 2
end := 7
str := "Go is a programming language"
sub := str[start : end]
```

reasoning on the value of `sub` at line 4, that is built through a substring operation, entails reasoning on both strings and integers. In fact, while most numeric domains have been formalized in isolation due to the nature of the safety-critical programs they were aimed at, string domains are defined in combination with a numeric domain of choice. Typically, such a domain tracks constant values (e.g., as in Costantini et al., 2015) or intervals modeling their ranges (e.g., as in Negrini et al., 2021). While these explicit combinations are enough to understand the article's contribution, they are often limiting when one wants to reuse the same domain in a different combination. Consider, for instance, the PREFIX domain (Costantini et al., 2015), tracking definite prefixes of string values. The authors define the semantics of `substring` with respect to the integer values of the start and end indices. If one were to reuse the domain in a more precise setting, e.g., in conjunction with the INTERVAL domain (Cousot and Cousot, 1977), the definition has to be lifted by applying the semantics to all valid pairs of indices. While in this case the lift is straightforward, it might not be if the string domain is more elaborate (e.g., if it uses automata as in Arceri et al., 2020). Moreover, lifting the semantics naïvely poses the threat of compromising the soundness of the analysis.

1.1 Contribution

This study presents a framework for *whole-value analyses*, that is, analyses aiming to model all values computed by a program regardless of their type. Specifically, the framework is an instance of the *open product* (Cortesi et al., 2013), and allows client abstract domains for individual data types (e.g., integers, floats, strings, ...) to exchange information modularly, without tailoring the communication to specific domain combinations. The exchange happens by means of abstract constraints that one domain can obtain from the others in the form of (in-)equalities that hold in the current state. Since the format is generic, one can add new domains to the framework or swap one of them with a more refined one without the others having to redefine or lift their semantics. We then show how abstract domains' semantics can be expressed in terms of such constraints whenever a *multi-type expression* (i.e., an expression whose sub-expressions and result are of heterogeneous types — e.g., Java's `substring`) needs to be evaluated. The strength of the framework is both theoretical and practical: domains adopting this framework (i) have their abstract semantics proven sound *independently* from the domain they will be combined with, and (ii) can be plugged in with different abstract domains seamlessly, with no code modifications. We then implement the framework in LiSA (Negrini et al., 2023a), and compare the results of whole-value analyses with and without the presented constraint-based framework to assess its precision. In summary:

- we define a novel constraint-based framework for whole-value analyses by abstract interpretation, proving its soundness;
- we recast the definition of some widespread abstract domains' semantics to fit our framework;
- we provide an open-source implementation of the framework in LiSA (Negrini et al., 2023a);
- we compare the analysis results of whole-value analyses with and without the proposed framework.

1.2 Paper structure

Section 2 introduces the necessary notations and notions that will be used throughout the study. Section 3 defines the IMP language, a minimalistic yet expressive imperative language that we will use for the formalization of the framework, together with its semantics. Section 4 defines the *split state*, a rewriting of the concrete state and semantics of IMP that simplifies the definition of abstract interpretations without introducing any loss of precision. Section 5 formalizes the framework. Section 6 reports our instantiation of the framework with some notable client abstract domains. Section 7 reports our implementation in LiSA

and our experiments to assess the precision of the framework. Section 8 discusses related work. Section 9 concludes and discusses future works. Appendix A reports the proofs of all lemmas and theorems.

2 Preliminaries

2.1 Sets

A set X is a possibly infinite collection of elements, written $X = \{x_0, x_1, \dots\}$, where $\emptyset$ is the set containing no elements. A set can also be defined in terms of a predicate ϕ, causing all elements satisfying ϕ to be part of the set (i.e., $X = \{\, x \mid \phi(x) \,\}$). We write $x \in X$ to denote that x is part of the set X. $|X|$ is the cardinality of X, that is, the number of elements it contains, and $\wp(X)$ is the powerset of X, that is, the set containing all subsets of X. Given two sets X and Y, $X \subseteq Y$ is the inclusion relation between X and Y, $X \cup Y$ is the set union between X and Y, $X \cap Y$ is the set intersection between X and Y, $X \setminus Y$ is the set difference between X and Y, and $X \times Y$ is the Cartesian product between X and Y, that is, the set $\{\, (x, y) \mid x \in X \wedge y \in Y \,\}$.

2.2 Functions

A function $f : X \to Y$ is a subset of the Cartesian product $X \times Y$ such that $\nexists (x, y), (z, w) \in f \;:\; x = z \wedge y \neq w$. The set X is called *domain* (denoted as $dom(f)$), the set Y is called *co-domain* (denoted as $codom(f)$), and $f(x)$ is the *image* of x in f, that is, $(x, f(x)) \in f$. Similarly to sets, a function can be defined either as a set of pairs $\{(x_0, y_0), (x_1, y_1), \dots\}$ or using a formula Φ, written $f(x) = \Phi(x)$, indicating that $f = \{(x_0, \Phi(x_0)), (x_1, \Phi(x_1)), \dots\}$. Function id is the identity function, that is, $id(x) = x$. Finally, given a function $f : X \to Y$, we denote with f^+ its additive lift, that is, the function $f^+ : \wp(X) \to \wp(Y)$ defined as $f^+(S) = \{\, (s, f(s)) \mid s \in S \,\}$.

2.3 Strings

A string σ is a sequence of characters $\sigma_0 \dots \sigma_n, \sigma_i \in \Sigma$, with length $|\sigma| = n + 1$. We denote as Σ^* the set of all possibly unbounded strings. Given a string $\sigma \in \Sigma^*$ and $i, j \in \mathbb{N}, 0 \leq i \leq j < |\sigma|$, we denote the subsequence $\sigma_i \dots \sigma_j$ by $\sigma[i : j]$. Instead, given two strings $\sigma = \sigma_0 \dots \sigma_n, \sigma' = \sigma'_0 \dots \sigma'_k \in \Sigma^*$, we write $\sigma' \curvearrowright \sigma$ if σ' is a substring of σ, that is, if $\exists i, j \in \mathbb{N}, 0 \leq i \leq j < |\sigma|, j - i = k \;:\; \sigma[i : j] = \sigma'$. Furthermore, we write $\sigma' \curvearrowright_p \sigma$ if σ' is a prefix of σ, that is, if $\exists\, \mathrm{k} < |\sigma| \wedge \sigma[0 : k] = \sigma$', and $\sigma' \curvearrowright_s \sigma$ if σ' is a suffix of σ, that is, if $\exists\, \mathrm{k} < |\sigma| \wedge \sigma[n - k : n] = \sigma'$.

2.4 Ordered structures

A set X with a partial ordering relation $\sqsubseteq_X \subseteq X \times X$ is a poset, denoted by $\langle \mathrm{X}, \sqsubseteq_X \rangle$. If a poset has a bottom element $\bot_X$ and is closed under finitary applications of the least upper bound (lub, $\sqcup_X$) operator of X, it is called a complete partial order (cpo), denoted as $\langle X, \sqsubseteq_X, \sqcup_X, \bot_X \rangle$. Moreover, a lattice $\langle X, \sqsubseteq_X, \sqcup_X, \sqcap_X \rangle$ is a poset having a minimum element (bottom, $\bot_X \in X$), a maximum element (top, $\top_X \in X$) and closed under finitary applications of the least upper bound (lub, $\sqcup_X$) and the greatest lower bound (glb, $\sqcap_X$) operators. A *complete* lattice is closed under arbitrary lub and glb, so that $\bigsqcup Y \in X$ and $\bigsqcap Y \in X$, for all $Y \subseteq X$, and it is denoted as $\langle X, \sqsubseteq_X, \sqcup_X, \sqcap_X, \top_X, \bot_X \rangle$. Provided there is no ambiguity, we will omit subscripts of each operator for clarity. Complete lattices can be derived from other structures. For instance, given a set X, $\langle \wp(X), \subseteq, \cup, \cap, X, \emptyset \rangle$ is a complete lattice since $\subseteq$ is a partial ordering relation, $\cup$ and $\cap$ are closed with respect to $\wp(X)$, and $\forall Y \in \wp(X) \;:\; \emptyset \subseteq Y \subseteq X$. By duality, $\langle \wp(X), \supseteq, \cap, \cup, \emptyset, X \rangle$ is also complete. Moreover, given $\langle X, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$ and a set Y, the *functional lift* (Cousot and Cousot, 1979) of X with respect to Y is the complete lattice $\langle Y \to X, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \dot{\bot}, \dot{\top} \rangle$ of total functions $Y \to X$, that is, of functions defined on all elements of Y. Lattice operators are defined as point-wise applications of operators over X on all $y \in Y$. Lastly, given a finite set of complete lattices $\langle Y_i, \sqsubseteq_{Y_i}, \sqcup_{Y_i}, \sqcap_{Y_i}, \bot_{Y_i}, \top_{Y_i} \rangle, i \in \Delta \subset \mathbb{N}$, their Cartesian product (Cousot, 2021) is the complete lattice $\langle \times \sum_{i \in \Delta} Y_i, \overset{\times}{\sqsubseteq}, \overset{\times}{\sqcup}, \overset{\times}{\sqcap}, \overset{\times}{\bot}, \overset{\times}{\top} \rangle$, where lattice operators are component-wise applications of the operators over each Y_i. Given a poset $\langle X, \sqsubseteq_X \rangle$, an increasing chain $C \subseteq X$ is a possibly infinite sequence of elements $x_0, x_1, \dots$ of X such that $x_0 \sqsubseteq_X x_1 \sqsubseteq_X \dots$.

2.5 Abstract interpretation

Abstract Interpretation (Cousot and Cousot, 1977; Cousot, 2021) is a theoretical framework for sound reasoning on semantic properties of a program, establishing a correspondence between the semantics of a program, called concrete semantics, and an approximation of it, called abstract semantics. Let C and A be complete lattices, a pair of functions $\alpha : C \to A$ and $\gamma : A \to C$ forms a *Galois Connection* between C and A, written $\langle C, \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \sqsubseteq_A \rangle$, if $\forall\, c \in C, a \in A : \alpha(c) \sqsubseteq_A a \Leftrightarrow c \sqsubseteq_C \gamma(a)$. Equivalently, the Galois Connection exists if $\alpha \circ \gamma$ is reductive (i.e., if $\forall\, a \in A \;:\; \alpha \circ \gamma(a) \sqsubseteq_A a$), and $\gamma \circ \alpha$ is extensive (i.e., if $\forall\, c \in C \;:\; c \sqsubseteq_C \gamma \circ \alpha(c)$). In addition, if $\alpha \circ \gamma = id$, then α and γ form a *Galois Embedding*, written $\langle C, \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftleftarrows}} \langle A, \sqsubseteq_A \rangle$, where no two abstract elements have the same concretization. Furthermore, if also $\gamma \circ \alpha = id$, then α and γ form a *Galois Isomorphism*, written $\langle C, \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftleftarrows\!\!\rightrightarrows}} \langle A, \sqsubseteq_A \rangle$, where A is simply a reshaping of C and no abstraction (i.e., loss of precision) happens. Note that Abstract Interpretation can be employed also when a Galois Connection does not exist: in fact, it is sufficient that C and A are complete partial orders, and that a monotone concretization γ exists.

2.6 Soundness

Given $\langle C, \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \sqsubseteq_A \rangle$, a concrete function $f : C \to C$ is, in general, not computable. Hence, a function $f^\sharp : A \to A$ that must *correctly* approximate the function f is needed. If so, we say that the function $f^\sharp$ is *sound*. Given $\langle C, \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \sqsubseteq_A \rangle$ and a concrete function $f : C \to C$, an abstract function $f^\sharp : A \to A$ is sound with respect to f if $\forall c \in C \;:\; \alpha(f(c)) \sqsubseteq_A f^\sharp(\alpha(c))$, or equivalently $\forall a \in A \;:\; f(\gamma(a)) \sqsubseteq_C \gamma(f^\sharp(a))$. Note that the latter relation can

$$
\begin{aligned}
&\mathsf{a} \in \textsc{Ae} ::= x \in \textsc{Id} \mid n \in \mathbb{Z} \mid \mathsf{a} \oplus \mathsf{a} \mid \texttt{len}(\mathsf{s})\\
&\mathsf{b} \in \textsc{Be} ::= x \in \textsc{Id} \mid \texttt{true} \mid \texttt{false} \mid \mathsf{b}\ \texttt{and}\ \mathsf{b} \mid \mathsf{b}\ \texttt{or}\ \mathsf{b} \mid \texttt{not}\ \mathsf{b} \mid \mathsf{a} \oslash \mathsf{a} \mid \mathsf{s} == \mathsf{s} \mid \texttt{contains}(\mathsf{s}, \mathsf{s})\\
&\mathsf{s} \in \textsc{Se} ::= x \in \textsc{Id} \mid \sigma \in \Sigma^* \mid \texttt{concat}(\mathsf{s}, \mathsf{s}) \mid \texttt{substr}(\mathsf{s}, \mathsf{a}, \mathsf{a})\\
&\mathsf{e} \in \textsc{Expr} ::= \mathsf{a} \mid \mathsf{s} \mid \mathsf{b}\\
&\mathsf{st} \in \textsc{Stmt} ::= \mathsf{st}\ ;\ \mathsf{st} \mid \texttt{skip} \mid x = \mathsf{e} \mid \texttt{if}\ \mathsf{b}\ \{\ \mathsf{st}\ \}\ \texttt{else}\ \{\ \mathsf{st}\ \} \mid \texttt{while}\ \mathsf{b}\ \{\ \mathsf{st}\ \}\\
&\mathsf{P} \in \textsc{Imp} ::= \mathsf{st}\ ;
\end{aligned}
$$

FIGURE 1
Syntax of the IMP language.

be used to prove soundness even when a Galois Connection does not exist.

2.7 Abstract domains

In the Abstract Interpretation framework, abstractions are defined through so-called *abstract domains*. These are composed by a partial order $\langle X, \sqsubseteq_X \rangle$, possibly extended to a complete partial order, a lattice, or a complete lattice, an upper bound operator $\sqcup_X$ on X, a bottom element $\bot_X \in X$ a widening operator ∇_X that over-approximates $\sqcup_X$ and ensures the convergence on increasing chains, an *abstract transformer* $\wr \mathsf{st} \wr : X \to X$ for evaluating statements, and an *abstract transformer* $\wr \mathsf{b} \wr : X \to X$ for traversing conditions. The purpose of the transformers is to evolve an instance of the domain according to the semantics of the statement st to be executed, and to refine an instance of the domain assuming that a condition b holds. When a domain is *non-relational*, that is, it does not maintain explicit relations between program variables, a third transformer $\wr \mathsf{e} \wr : X \to V$ usually exists, evaluating an expression e to an abstract value $v \in V$, with $\langle V, \sqsubseteq_V, \sqcup_V, \bot_V \rangle$ being the complete partial order of abstract values (possibly also a lattice or a complete lattice). For instance, the domain of intervals $\langle \textsc{Id} \to \mathbb{I}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \dot{\bot}, \dot{\top} \rangle$ uses functions as abstract elements, but evaluates expressions to single intervals.

3 The Imp language

We begin by defining the target programming language that we aim to analyze: IMP. IMP, whose syntax is visible in Figure 1, is a simple imperative language that features arithmetic expressions (AE, where $\oplus \in \{+, -, *, /\}$ with / being the integer division), Boolean expression (BE, where $\oslash \in \{==, !=, <, <=, >, >=\}$), and string expressions (SE). It then features variable assignments to integers, strings, and Booleans, branching and looping. Note that, despite its simplicity, it features multi-type expressions **len**(s), a $\oslash$ a, s == s, **contains**(s, s), and **substr**(s, a, a) that require mixing values from different domains to be computed (i.e., **substr**(s, a, a) requires both strings and integers to be computed, while a $\oslash$ a, s == s, **len**(s) and **contains**(s, s) can be fully computed using the input values — integers or strings — but have output of a different type — integers or Booleans). IMP programs need to be well-typed to be valid: each variable defined and used throughout the program must be assigned to values of a single data type.

3.1 Concrete state and semantics

Expressions of the IMP language evaluate to values in the set $\textsc{Val} \triangleq \mathbb{Z} \cup \Sigma^* \cup \mathbb{B} \cup \{\Uparrow\}$, that is, to integers, strings, Booleans ($\{\texttt{true}, \texttt{false}\}$), or to a special value $\Uparrow$ denoting an error in the evaluation. Program memories $\mu \in \mathcal{M} : \textsc{Id} \to \textsc{Val}$ map program variables in the set ID to their values in VAL. Abusing notation, the set $\mathcal{M}$ contains a special memory $\Uparrow$ produced after invalid computations. Function $[\![\mathsf{st}]\!] : \mathcal{M} \to \mathcal{M}$ defines the semantics of each statement in terms of the effect it has on the program memory it is executed on. Expression evaluation is instead defined, abusing notation, through the function $[\![\mathsf{e}]\!] : \mathcal{M} \to \textsc{Val}$ that computes the value of the expression given the values of each variable. The semantics of IMP statements and expressions is standard, and it is thus not fully specified: in the following, we only report the semantics of multi-type expressions in a big-step fashion. Note that both $[\![\mathsf{st}]\!]$ and $[\![\mathsf{e}]\!]$ yield $\Uparrow$ if the semantics of any sub-expression appearing in their argument evaluates to $\Uparrow$.

$$\frac{[\![\mathsf{s}]\!]\mu = \sigma}{[\![\texttt{len}(\mathsf{s})]\!]\mu = |\sigma|} \tag{3.1}$$

$$\frac{[\![\mathsf{a}_1]\!]\mu = n_1 \qquad [\![\mathsf{a}_2]\!]\mu = n_2}{[\![\mathsf{a}_1 \oslash \mathsf{a}_2]\!]\mu = n_1 \oslash n_2} \tag{3.2}$$

$$\frac{[\![\mathsf{s}_1]\!]\mu = \sigma_1 \qquad [\![\mathsf{s}_2]\!]\mu = \sigma_2}{[\![\mathsf{s}_1 == \mathsf{s}_2]\!]\mu = \sigma_1 == \sigma_2} \tag{3.3}$$

$$\frac{[\![\mathsf{s}_1]\!]\mu = \sigma_1 \qquad [\![\mathsf{s}_2]\!]\mu = \sigma_2 \qquad \sigma_2 \curvearrowright \sigma_1}{[\![\texttt{contains}(\mathsf{s}_1, \mathsf{s}_2)]\!]\mu = \texttt{true}} \tag{3.4}$$

$$\frac{[\![\mathsf{s}_1]\!]\mu = \sigma_1 \qquad [\![\mathsf{s}_2]\!]\mu = \sigma_2 \qquad \sigma_2 \not\curvearrowright \sigma_1}{[\![\texttt{contains}(\mathsf{s}_1, \mathsf{s}_2)]\!]\mu = \texttt{false}} \tag{3.5}$$

$$\frac{[\![\mathsf{s}]\!]\mu = \sigma \qquad [\![\mathsf{a}_1]\!]\mu = i \qquad [\![\mathsf{a}_2]\!]\mu = j \qquad 0 \le i \le j < |\sigma|}{[\![\texttt{substr}(\mathsf{s}, \mathsf{a}_1, \mathsf{a}_2)]\!]\mu = \sigma[i:j]} \tag{3.6}$$

$$\frac{[\![\mathsf{s}]\!]\mu = \sigma \qquad [\![\mathsf{a}_1]\!]\mu = i \qquad [\![\mathsf{a}_2]\!]\mu = j \qquad i < 0 \vee j \ge |\sigma| \vee i > j}{[\![\texttt{substr}(\mathsf{s}, \mathsf{a}_1, \mathsf{a}_2)]\!]\mu = \Uparrow} \tag{3.7}$$

Intuitively, Equation 3.1 shows that **len** returns the number of characters in its input. Comparisons on both integers (Equation 3.2) and strings (Equation 3.3) simply apply the comparisons over the literals obtained through recursive evaluation of the arguments.The semantics of **contains** is partitioned according to its output: Equation 3.4 yields **true** if the second argument is contained in the first one, while Equation 3.5 yields **false** if it is not. Finally, **substr** returns either (i) the substring of its first argument delimited by its second and third ones as shown in Equation 3.1, or (ii) an evaluation error if the bounds are invalid, as visible in Equation 3.1.

As usual in Abstract Interpretation, the concrete semantics is lifted to the *collecting semantics* by using sets of possible program memories. Specifically, instead of considering a single memory, accounting for an individual program execution, the collecting semantics considers possibly infinite sets of memories, describing possibly infinite sets of executions. In this setting, the semantics of statements and expressions are defined as the additive lift of the concrete ones. The statement collecting semantics $[\![\mathsf{st}]\!]^+ : \wp(\mathcal{M}) \to \wp(\mathcal{M})$ is thus defined as $[\![\mathsf{st}]\!]^+ M \triangleq \{\, [\![\mathsf{st}]\!]\mu \mid \mu \in M \,\}$, while the expression collecting semantics $[\![\mathsf{e}]\!]^+ : \wp(\mathcal{M}) \to \wp(\mathrm{VAL})$ is defined as $[\![\mathsf{e}]\!]^+ M \triangleq \{\, [\![\mathsf{e}]\!]\mu \mid \mu \in M \,\}$. Lastly, note that sets of program memories are elements of the complete powerset lattice $\langle \wp(\mathcal{M}), \subseteq, \cup, \cap, \emptyset, \mathcal{M} \rangle$.

4 The split state

The collecting semantics of IMP programs defined in Section 3 provides information on all concrete executions of a given program, but it is not computable. Following the Abstract Interpretation framework, we aim at *abstracting* the collecting semantics, regaining decidability by introducing imprecision. While it is possible to design an abstraction for the collecting semantics we defined, it would be inconvenient for the purpose of this work. We aim at building a framework where semantic computations are delegated to several abstract domains operating on disjoint data types: abstract states will thus be the conjunction of the states of the individual domains. A direct abstraction would thus complicate both definitions and proofs, since the conversion from a monolithic state to a partitioned one would be necessary at each step.

Instead, we adapt the idea from Ferrara (2016) of introducing an intermediate abstraction. In this section, we define a *split state*, that is, a rewriting of the concrete state (i.e., program memories and concrete semantics) into one where values of different data types are stored in separate maps. This rewriting models the concrete state in a way that is convenient for the remainder of this work, simplifying definitions and proofs. Having separate memories also facilitates modular abstractions: existing abstract domains (both non-relational and relational) can be employed to abstract individual sub-memories. Concretizations and transformers from such domains can be used as-is, only redefining the evaluation of expressions spanning multiple data types. Moreover, such rewriting *does not introduce imprecision*: proving soundness and/or completeness with respect to the split state is thus equivalent to proving it with respect to the concrete state.

4.1 Split memories

We begin by partitioning program memories, grouping program variables by the type of values they hold. Since IMP can store values of three different types (integers, strings and Booleans), we define three type-specific memories: $\overline{\mu}_a \in \mathcal{A} : \mathrm{ID} \to \mathbb{Z}$ stores the values of integer variables, $\overline{\mu}_s \in \mathcal{S} : \mathrm{ID} \to \Sigma^*$ stores the values of string variables, and $\overline{\mu}_b \in \mathcal{B} : \mathrm{ID} \to \mathbb{B}$ stores the values of Boolean variables. Split memories are $\overline{\mu} \in \overline{\mathcal{M}} \triangleq \mathcal{A} \times \mathcal{S} \times \mathcal{B}$, that is, tuples composed by a function for each data type supported by the language. Similarly to $\mathcal{M}$, $\overline{\mathcal{M}}$ also contains a special memory $\Uparrow$. In the following, we will refer to a split memory as either $\overline{\mu}$ or $(\overline{\mu}_a, \overline{\mu}_s, \overline{\mu}_b)$. Being a rewriting of the concrete state, split memories still refer to a single execution and should hold a single value for any given variable. We thus consider only *valid* split memories, that is, memories $\overline{\mu}$ such that $dom(\overline{\mu}_a) \cap dom(\overline{\mu}_s) = dom(\overline{\mu}_a) \cap dom(\overline{\mu}_b) = dom(\overline{\mu}_s) \cap dom(\overline{\mu}_b) = \emptyset$. Split memories arbitrarily constructed to hold information on the same variable in more than one type-specific memory are ignored as they cannot arise when abstracting a valid concrete memory.

Example 4.1. Consider the program memory $\mu = \{(x, 42), (y, \text{“hello”}), (z, \texttt{true})\}$. The corresponding split memory $\overline{\mu} = (\overline{\mu}_a, \overline{\mu}_s, \overline{\mu}_b)$ is such that $\overline{\mu}_a = \{(x, 42)\}$, $\overline{\mu}_s = \{(y, \text{“hello”})\}$, and $\overline{\mu}_b = \{(z, \texttt{true})\}$. The means for commuting between the two representations will be defined in the following section.

4.2 Abstraction and concretization

Before defining $\overline{\alpha}$ and $\overline{\gamma}$, used to convert concrete states into split states and vice-versa, we introduce two operators that will be used in their formalization: the restriction operator $\downharpoonright$ and the union operator $\uplus$.

Definition 1 (Function restriction)**.** Given a function $f : K \to V$ and two sets $X \subseteq K$ and $Y \subseteq V$, the function restriction operator $\downharpoonright_Y^X : (K \to V) \to (X \to Y)$ yields the function $f \downharpoonright_Y^X = \{\, (x, f(x)) \mid x \in X \wedge f(x) \in Y \,\}$, that is, it *restricts* the input function f on the elements in the desired domain and co-domain.

Definition 2 (Function union)**.** Given two functions $f : X \to W$ and $g : Y \to Z$ such that $X \cap Y = \emptyset$, the function union operator $\uplus : (X \to W) \times (Y \to Z) \to (X \cup Y \to W \cup Z)$ yields the function $f \uplus g = \{\, (k, v) \mid (k \in dom(f) \wedge v = f(k)) \vee (k \in dom(g) \wedge v = g(k)) \,\}$, that is, it *combines* the input functions f and g. Since the domains of f and g are disjoint, $f \uplus g$ is also a function.

Intuitively, the above operators are used to split a function into sub-functions and to join them back to the original function, respectively, as shown in the following example.

Example 4.2. Consider the function $f = \{(1, 2), (2, 3), (3, 4)\}$ and the sets $X = \{1, 2\}$ and $Y = \{2, 3\}$. Let $\overline{X} = dom(f) \setminus X$ and $\overline{Y} =$

codom$(f) \setminus Y$. The restriction of f to X and Y is $f\downarrow_Y^X = \{(1,2),(2,3)\}$, and its complement is $f\downarrow_{\overline{Y}}^{\overline{X}} = \{(3,4)\}$. The union of the restriction and its complement is $f\downarrow_Y^X \uplus f\downarrow_{\mathbb{Z}\setminus Y}^{\mathbb{Z}\setminus X} = \{(1,2),(2,3),(3,4)\} = f$.

To avoid cluttering the notation, the superscript of $\downarrow$ will be omitted whenever it coincides with the domain of the whole function. Note that both definitions can be trivially generalized to a partitioning that generates an arbitrary number of sub-functions. We are now in a position to define $\overline{\alpha}$ and $\overline{\gamma}$.

Definition 3 (Abstraction of concrete states)**.** The abstraction function $\overline{\alpha} : \wp(\mathcal{M}) \to \wp(\overline{\mathcal{M}})$ converts sets of program memories to sets of split memories by converting each individual memory through the function $\dot{\alpha} : \mathcal{M} \to \overline{\mathcal{M}}$. Formally:

$$\overline{\alpha}(\{\mu_1,\ldots,\mu_k\}) = \{\,\dot{\alpha}(\mu) \mid \mu \in \{\mu_1,\ldots,\mu_k\}\,\} \qquad \text{where}$$
$$\dot{\alpha}(\mu) = (\mu\downarrow_{\mathbb{Z}}, \mu\downarrow_{\Sigma^*}, \mu\downarrow_{\mathbb{B}}).$$

Lemma 4 (Monotonicity of $\overline{\alpha}$)**.** Function $\overline{\alpha}$ is monotone, that is, $\forall M_1, M_2 \subseteq \mathcal{M} : M_1 \subseteq M_2 \implies \overline{\alpha}(M_1) \subseteq \overline{\alpha}(M_2)$. Proof in Appendix A.1.

Definition 5 (Concretization of split states)**.** The concretization function $\overline{\gamma} : \wp(\overline{\mathcal{M}}) \to \wp(\mathcal{M})$ converts sets of split memories to sets of program memories by converting each individual memory through the function $\dot{\gamma} : \overline{\mathcal{M}} \to \mathcal{M}$. Formally:

$$\overline{\gamma}(\{\overline{\mu}_1,\ldots,\overline{\mu}_k\}) = \{\,\dot{\gamma}(\overline{\mu}) \mid \overline{\mu} \in \{\overline{\mu}_1,\ldots,\overline{\mu}_k\}\,\} \qquad \text{where}$$
$$\dot{\gamma}(\overline{\mu}) = \overline{\mu}_a \uplus \overline{\mu}_s \uplus \overline{\mu}_b.$$

Lemma 6 (Monotonicity of $\overline{\gamma}$)**.** Function $\overline{\gamma}$ is monotone, that is, $\forall M_1, M_2 \subseteq \overline{\mathcal{M}} : M_1 \subseteq M_2 \implies \overline{\gamma}(M_1) \subseteq \overline{\gamma}(M_2)$. Proof in Appendix A.1.

Example 4.3. Consider again the program memory $\mu = \{(x,42),(y,\text{“hello”}),(z,\texttt{true})\}$ from Example 4.1. The abstraction $\overline{\mu} = (\overline{\mu}_a, \overline{\mu}_s, \overline{\mu}_b)$ of such memory is computed by $\dot{\alpha}$ by partitioning the co-domain. Specifically, $\overline{\mu}_a = \mu\downarrow_{\mathbb{Z}} = \{(k,\mu(k)) \mid k \in dom(\mu) \wedge \mu(k) \in \mathbb{Z}\} = \{(x,42)\}$. Similarly, $\overline{\mu}_s = \mu\downarrow_{\Sigma^*} = \{(y,\text{“hello”})\}$ and $\overline{\mu}_b = \mu\downarrow_{\mathbb{B}} = \{(z,\texttt{true})\}$. Since the domains of each type-specific memory are disjoint, the concretization of $\overline{\mu}$ through $\dot{\gamma}$ simply joins the memories together, obtaining μ.

Lemma 7 (Invertibility of $\overline{\alpha}$ and $\overline{\gamma}$)**.** The composition of $\overline{\alpha}$ and $\overline{\gamma}$ corresponds to the identity function, that is, $\overline{\alpha} \circ \overline{\gamma} = id$. Moreover, the composition of $\overline{\gamma}$ and $\overline{\alpha}$ corresponds to the identity function, that is, $\overline{\gamma} \circ \overline{\alpha} = id$. Thus, $\overline{\alpha}$ is the inverse of $\overline{\gamma}$, and vice-versa. Proof in Appendix A.1.

Both the abstraction and concretization functions thus proceed by converting each input memory individually, where the individual conversions partition the memory by data types in the abstraction, and combines them back in the case of the concretization. Since both functions are monotone and they compose to the identity function (as shown in Appendix A.1), they induce the Galois Isomorphism $\langle \wp(\mathcal{M}), \subseteq \rangle \underset{\overline{\alpha}}{\overset{\overline{\gamma}}{\leftrightarrows}} \langle \wp(\overline{\mathcal{M}}), \subseteq \rangle$.

4.3 Split semantics of statements

Instead of specifying a custom semantics for the split state, we take full advantage of the isomorphism. In fact, a well-known property of Galois Connections (and thus of Isomorphisms) is that the abstraction function yields the *best* possible abstraction of any concrete element. This means that the split collecting semantics of a statement can be computed in the concrete after applying $\overline{\gamma}$, and then abstracted back to the split setting by $\overline{\alpha}$. Formally, the collecting split statement semantics $(\!|\mathsf{st}|\!)^+ : \wp(\overline{\mathcal{M}}) \to \wp(\overline{\mathcal{M}})$ is defined as $(\!|\mathsf{st}|\!)^+ = \overline{\alpha} \circ [\![\mathsf{st}]\!]^+ \circ \overline{\gamma}$. Such a definition is always possible within Abstract Interpretation, albeit not desirable: computability is not recovered, as the undecidable concrete semantics is involved in the computation. However, since the purpose of the split state is to provide a more convenient starting point for successive abstractions, uncomputability is not a concern. Similarly, we define the split collecting semantics of expressions $(\!|\mathsf{e}|\!)^+ : \wp(\overline{\mathcal{M}}) \to \wp(\textsc{Val})$ as $(\!|\mathsf{e}|\!)^+ = [\![\mathsf{e}]\!]^+ \circ \overline{\gamma}$ (here, $\overline{\alpha}$ is not involved in the computation since $(\!|\mathsf{e}|\!)^+$ produces concrete values that do not need to be abstracted). Note that sets of split memories are elements of the complete powerset lattice $\langle \wp(\overline{\mathcal{M}}), \subseteq, \cup, \cap, \emptyset, \overline{\mathcal{M}} \rangle$.

While the presence of a Galois Isomorphism ensures that no precision is lost when commuting between concrete and split states, imprecision might still arise during computations in the split semantics. We thus also have to prove the equivalence (i.e., soundness and completeness) of the split semantics with respect to the concrete one. When two semantics f and $f^\sharp$ are expressed in fixpoint form, it is enough to show that $\alpha \circ f = f^\sharp \circ \alpha$ to ensure that $\alpha(\mathsf{lfp}\, f) = \mathsf{lfp}\, f^\sharp$ (Cousot and Cousot, 1979), where $\mathsf{lfp}\, f$ is the least fixpoint of the iterates of f. Instead, since both the concrete and split semantics are defined in their big-step forms, we have to prove that $\forall\, \mathsf{st} \in \textsc{Stmt} : [\![\mathsf{st}]\!]^+ \circ \overline{\gamma} = \overline{\gamma} \circ (\!|\mathsf{st}|\!)^+$.

Theorem 8 (Equivalence of concrete and split semantics)**.** For all sets of split memories $M \subseteq \overline{\mathcal{M}}$, statements $\mathsf{st} \in \textsc{Stmt}$, and expressions $\mathsf{e} \in \textsc{Expr}$, both $[\![\mathsf{st}]\!]^+\overline{\gamma}(M) = \overline{\gamma}((\!|\mathsf{st}|\!)^+ M)$ and $[\![\mathsf{e}]\!]^+\overline{\gamma}(M) = (\!|\mathsf{e}|\!)^+ M$ hold. Proof in Appendix A.2.

We thus conclude that the split state is a rephrasing of the concrete state, with no loss of precision introduced either in the conversion between the two or during the semantics computations over the split state.

5 Constraint-based whole-value analysis

By considering split states instead of concrete ones, we start from a setting where there is a clear distinction between variables holding integers ($\overline{\mu}_a$), ones holding strings ($\overline{\mu}_s$), and ones holding Booleans ($\overline{\mu}_b$). We can thus exploit a combination of existing domains in our framework, one abstracting each set of variables. We then simply need to specify how these domains can communicate to exchange information on values crossing type boundaries. We thus assume that the integer part of the memory $\overline{\mu}_a$ is abstracted by an abstract domain $\mathcal{A}^\sharp$, the string part of the memory $\overline{\mu}_s$ is abstracted by an abstract domain $\mathcal{S}^\sharp$, and that the Boolean part of the memory $\overline{\mu}_b$ is abstracted by an abstract domain $\mathcal{B}^\sharp$. In the following, let $\mathcal{X}$ be an abstract domain in $\{\mathcal{A}^\sharp, \mathcal{S}^\sharp, \mathcal{B}^\sharp\}$, abstracting the respective type-specific memory $\overline{\mathcal{M}}_{\mathcal{X}}$ in $\{\mathcal{A}, \mathcal{S}, \mathcal{B}\}$. We only require $\mathcal{X}$ to provide the minimum ingredients for Abstract Interpretation:

- it must form a partial order, denoted $\langle \mathcal{X}, \sqsubseteq_{\mathcal{X}} \rangle$, and should provide an upper bound operator $\sqcup_{\mathcal{X}}$ and a bottom element $\bot_{\mathcal{X}}$;
- it provides a widening operator $\nabla_{\mathcal{X}}$ (if not needed by the domain itself, it can simply delegate to $\sqcup_{\mathcal{X}}$);
- it defines a monotone concretization function $\gamma_{\mathcal{X}} : \mathcal{X} \to \wp(\overline{\mathcal{M}}_{\mathcal{X}})$;
- it defines abstract transformers for the statement semantics $\wr \mathsf{st} \wr^{\mathcal{X}}$ and expression evaluation $\wr \mathsf{e} \wr^{\mathcal{X}}$ that are sound (i.e., they over-approximate what the collecting semantics would compute on statements and expressions of the data type they abstract).

Note that, while the existence of $\wr \mathsf{e} \wr$ is typical of non-relational domains, relational ones that are to be employed in whole-value analyses still need to produce an abstraction of an expression's value to provide to other domains. Thus, the framework is not limited to non-relational domains only.

5.1 Abstract states

We define the states of our framework as a special instance of the Cartesian product of the base domains.

Definition 9 (Abstract program memories)**.** In our framework, abstract program states are defined as $\mu^\sharp \in \mathcal{M}^\sharp \triangleq \mathcal{A}^\sharp \otimes \mathcal{S}^\sharp \otimes \mathcal{B}^\sharp$, that is, as tuples of instances of the three domains. Here, $\otimes$ denotes the *smash product* (Arceri and Maffeis, 2017), that is, a form of *reduced product* (Cortesi et al., 2013) where a bottom element from one component is propagated to all others. Being a Cartesian product (Section 2), operators over $\mathcal{M}^\sharp$ are element-wise applications of the ones over the sub-domains:

- $(a_1^\sharp, s_1^\sharp, b_1^\sharp) \sqsubseteq_{\mathcal{M}^\sharp} (a_2^\sharp, s_2^\sharp, b_2^\sharp) \iff a_1^\sharp \sqsubseteq_{\mathcal{A}^\sharp} a_2^\sharp \wedge s_1^\sharp \sqsubseteq_{\mathcal{S}^\sharp} s_2^\sharp \wedge b_1^\sharp \sqsubseteq_{\mathcal{B}^\sharp} b_2^\sharp$;
- $(a_1^\sharp, s_1^\sharp, b_1^\sharp) \sqcup_{\mathcal{M}^\sharp} (a_2^\sharp, s_2^\sharp, b_2^\sharp) = (a_1^\sharp \sqcup_{\mathcal{A}^\sharp} a_2^\sharp, s_1^\sharp \sqcup_{\mathcal{S}^\sharp} s_2^\sharp, b_1^\sharp \sqcup_{\mathcal{B}^\sharp} b_2^\sharp)$;
- $(a_1^\sharp, s_1^\sharp, b_1^\sharp) \nabla_{\mathcal{M}^\sharp} (a_2^\sharp, s_2^\sharp, b_2^\sharp) = (a_1^\sharp \nabla_{\mathcal{A}^\sharp} a_2^\sharp, s_1^\sharp \nabla_{\mathcal{S}^\sharp} s_2^\sharp, b_1^\sharp \nabla_{\mathcal{B}^\sharp} b_2^\sharp)$;

Note that, being built with a smash product (that is a special instance of Cartesian product), $\mathcal{M}^\sharp$ and its operators have the same properties of the three sub-domains (i.e., $\sqsubseteq_{\mathcal{M}^\sharp}$ is a partial order, $\sqcup_{\mathcal{M}^\sharp}$ is an upper bound operator, $\bot_{\mathcal{M}^\sharp}$ is the bottom element, and $\nabla_{\mathcal{M}^\sharp}$ is a widening operator). In the following, we will refer to an abstract memory as either $\mu^\sharp$ or $(a^\sharp, s^\sharp, b^\sharp)$.

5.2 Concretization and abstract semantics

Before discussing how to define the semantics of our framework, we first have to define how abstract states concretize to split states, which is essential to prove soundness. As previously mentioned, we assume $\gamma_{\mathcal{A}^\sharp} : \mathcal{A}^\sharp \to \wp(\textsc{Id} \to \mathbb{Z})$, $\gamma_{\mathcal{S}^\sharp} : \mathcal{S}^\sharp \to \wp(\textsc{Id} \to \Sigma^*)$, and $\gamma_{\mathcal{B}^\sharp} : \mathcal{B}^\sharp \to \wp(\textsc{Id} \to \mathbb{B})$ to be monotone functions, concretizing instances of $\mathcal{A}^\sharp$, $\mathcal{S}^\sharp$, and $\mathcal{B}^\sharp$ to sets of memories containing integers, strings, and Booleans, respectively. Since our objective is to connect $\mathcal{M}^\sharp$ to $\overline{\mathcal{M}}$, we equip our framework with a function $\gamma_{\mathcal{M}^\sharp} : \mathcal{M}^\sharp \to \wp(\overline{\mathcal{M}})$ by exploiting the three provided concretizations.

Definition 10 (Concretization of abstract states)**.** The concretization function $\gamma_{\mathcal{M}^\sharp} : \mathcal{M}^\sharp \to \wp(\overline{\mathcal{M}})$ converts an abstract state to a set of split memories. Formally:

$$\gamma_{\mathcal{M}^\sharp}(\mu^\sharp) = \{\, (\overline{\mu}_a, \overline{\mu}_s, \overline{\mu}_b) \mid \overline{\mu}_a \in \gamma_{\mathcal{A}^\sharp}(a^\sharp) \wedge \overline{\mu}_s \in \gamma_{\mathcal{S}^\sharp}(s^\sharp) \wedge \overline{\mu}_b \in \gamma_{\mathcal{S}^\sharp}(b^\sharp) \,\}.$$

Note that this definition is only possible since Imp programs are well-typed: each variable will appear in the state of exactly one domain, and each split memory is thus guaranteed to be valid (i.e., the type-specific memories have disjoint domains). The extension of this framework to programs that are not well-typed is straightforward, as it just entails a new definition of $\gamma_{\mathcal{M}^\sharp}$. We exclude this from our work since it would add unnecessary complexity to all proofs.

Lemma 11 (Monotonicity of $\gamma_{\mathcal{M}^\sharp}$)**.** Function $\gamma_{\mathcal{M}^\sharp}$ is monotone, that is, $\forall \mu_1^\sharp, \mu_2^\sharp \subseteq \mathcal{M}^\sharp : \mu_1^\sharp \subseteq \mu_2^\sharp \implies \gamma_{\mathcal{M}^\sharp}(\mu_1^\sharp) \subseteq \gamma_{\mathcal{M}^\sharp}(\mu_2^\sharp)$. Proof in Appendix A.3.

Example 5.1. Suppose that the framework is instantiated using the numerical constant propagation abstraction, the string prefix abstraction (Costantini et al., 2015), and the Boolean powerset abstraction. Abstract states are thus $\langle \textsc{Id} \to \mathbb{Z}_\bot^\top \otimes \textsc{Id} \to \Sigma_\bot^* \otimes \textsc{Id} \to \wp(\mathbb{B}) \rangle$. Consider the state $\mu^\sharp = (a^\sharp, s^\sharp, b^\sharp)$ where $a^\sharp = \{(x, 42)\}$, $s^\sharp = \{(y, \text{“foo”})\}$, and $b^\sharp = \{(z, \texttt{true})\}$. The individual domain concretizations produce sets of memories containing integers, strings, and Booleans, respectively. When applied to the given state, they thus produce $\{\{(x, 42)\}\}$, $\{\, \{(y, \sigma)\} \mid \text{“foo”} \curvearrowright_p \sigma \,\}$, and $\{\{(z, \texttt{true})\}\}$, respectively. The concretization function $\gamma_{\mathcal{M}^\sharp}$ returns all possible combinations of such memories, producing the set $\{\, (\{(x, 42)\}, \{(y, \sigma)\}, \{(z, \texttt{true})\}) \mid \text{“foo”} \curvearrowright_p \sigma \,\}$.

5.3 Abstract semantics

As typical in Cartesian products, the abstract statement semantics $\wr \mathsf{st} \wr : \mathcal{M}^\sharp \to \mathcal{M}^\sharp$ of our framework is defined as $\wr \mathsf{st} \wr (a^\sharp, s^\sharp, b^\sharp) = (\wr \mathsf{st} \wr^{\mathcal{A}^\sharp} a^\sharp, \wr \mathsf{st} \wr^{\mathcal{S}^\sharp} s^\sharp, \wr \mathsf{st} \wr^{\mathcal{B}^\sharp} b^\sharp)$, where $\wr \mathsf{st} \wr^{\mathcal{A}^\sharp}$ is the statement semantics of $\mathcal{A}^\sharp$, $\wr \mathsf{st} \wr^{\mathcal{S}^\sharp}$ is the statement semantics of $\mathcal{S}^\sharp$, and $\wr \mathsf{st} \wr^{\mathcal{B}^\sharp}$ is the statement semantics of $\mathcal{B}^\sharp$. The state of each component of the framework thus evolves in isolation, storing information on disjoint sets of variables. Note that such a semantics is inherently sound, as it is the element-wise application of sound abstract transformers. Our framework goes beyond the Cartesian product: in fact, we adopt the *open product* framework (Cortesi et al., 2013) to allow for modular communication between the different domains. We thus redefine how multi-type expressions are evaluated within the framework. The key idea is to let one domain evaluate the expression, while providing it with an additional tool: abstract constraints. These are formalized as a subset of the Boolean expressions Be parametric on the expression being constrained, denoted $\overline{\textsc{Be}}(\mathsf{e})$.

Definition 12 (Abstract constraints set $\overline{\textsc{Be}}(\mathsf{e})$)**.** The set of abstract constraints $\overline{\textsc{Be}}(\mathsf{e})$, parametric over an expression e, is the set of

Boolean expressions $\langle v \circledast \mathsf{e} \rangle$ with a value $v \in \textsc{Val}$ on the left-hand side, and an expression e on the right-hand side. Given an expression $\mathsf{e} \in \textsc{Expr}$, $\overline{\textsc{Be}}(\mathsf{e})$ contains the elements:

$$\overline{\textsc{Be}}(\mathsf{e}) \triangleq \begin{cases} \{ \langle v \circledcirc \mathsf{e} \rangle \mid v \in \mathbb{Z} \} & \text{if } \mathsf{e} \in \textsc{Ae}; \\ \{ \langle v == \mathsf{e} \rangle \mid v \in \mathbb{B} \} & \text{if } \mathsf{e} \in \textsc{Be}; \\ \{ \langle v \otimes \mathsf{e} \rangle \mid v \in \Sigma^* \wedge \otimes \in \\ \{==, !=, \curvearrowright, \curvearrowright_p, \curvearrowright_s\} \} \\ \quad \cup \{ \langle v \circledcirc \mathbf{len}(\mathsf{e}) \rangle \mid v \in \mathbb{Z} \} & \text{if } \mathsf{e} \in \textsc{Se}. \end{cases}$$

Depending on the type of expression e, $\overline{\textsc{Be}}(\mathsf{e})$ thus contains (i) numeric (in-)equalities, (ii) Boolean equalities, or (iii) constraints about the contents and length of a string. A set of constraints $B \subseteq \overline{\textsc{Be}}(\mathsf{e})$ represents *definite* information: if $B = \emptyset$ then all possible concrete values compatible with the expression's type are allowed, while each generated constraint reduces the set of possible values. Furthermore, a special constraint set $\lightning$ is used to denote an invalid set (e.g., with contradicting constraints), that models an expression whose evaluation leads to an error. The powerset of constraints form the complete lattice $\langle \wp(\overline{\textsc{Be}}(\mathsf{e})), \supseteq, \cap, \cup, \emptyset, \overline{\textsc{Be}}(\mathsf{e}) \rangle$, whose elements can be converted to sets of concrete values using the concretization function $\gamma_{\overline{\textsc{Be}}(\mathsf{e})}$.

Definition 13 (Concretization of abstract constraints)**.** The concretization function $\gamma_{\overline{\textsc{Be}}(\mathsf{e})} : \wp(\overline{\textsc{Be}}(\mathsf{e})) \to \wp(\textsc{Val})$ converts a set of abstract constraints to the set of concrete values satisfying them. Formally:

$$\gamma_{\overline{\textsc{Be}}(\mathsf{e})}(B) = \{ v \mid \forall \langle v' \circledast e \rangle \in B : v' \circledast v \} \qquad \text{with} \qquad \gamma_{\overline{\textsc{Be}}(\mathsf{e})}(\lightning) = \{\Uparrow\}.$$

Note that $\gamma_{\overline{\textsc{Be}}(\mathsf{e})}$ is trivially monotone, since fewer constraints (recall that the ordering relation over $\wp(\overline{\textsc{Be}}(\mathsf{e}))$ is $\supseteq$) lead to a larger set of values satisfying them.

Abstract domains need to both (i) generate constraints on expressions they can handle (i.e., of the type they model), and (ii) generate an abstract element from a set of constraints, in order to interpret the information provided to them by the other domains. We thus define two functions, $\mathbb{C}$ and $\mathbb{G}$, that model these operations. In the following, let $\mathcal{D}$ be an abstract domain providing all the ingredients enumerated at the beginning of Section 5, and let $\mathcal{V}$ be the complete partial order of the values produced by $\wr \mathsf{e} \wr^{\mathcal{D}}$.

Definition 14 (Abstract constraint function $\mathbb{C}$)**.** Given an abstract domain $\mathcal{D}$, the abstract constraint function $\mathbb{C}_{\mathcal{D}} : \mathcal{D} \times \textsc{Expr} \to \wp(\overline{\textsc{Be}}(\mathsf{e}))$, where the second argument corresponds to the parameter of the generated constraint set, yields Boolean constraints about an expression $\mathsf{e} \in \textsc{Expr}$, generating bounds on its concrete values based on the information contained in an instance of the domain $d^\sharp \in \mathcal{D}$. The result of the function is a set of Boolean constraints parametric in the input expression.

$\mathbb{C}$ can thus be used to restrict the set of values an expression can evaluate to. Note that, by passing a relational expression to $\mathbb{C}$, one can gain relational information despite the output being given as non-relational constraints. We assume that sets generated by $\mathbb{C}$ and received by $\mathbb{G}$ are *minimal*, that is, they do not include constraints that are implied by other ones already in the set. Also, recall that no contradicting constraints can be generated, as they would lead to the invalid set of constraints $\lightning$.

Definition 15 (Constraint interpretation function $\mathbb{G}$)**.** Given an abstract domain $\mathcal{D}$ that models expression values as elements of a complete partial order $\mathcal{V}$, a set of constraints parametric over an expression $\mathsf{e} \in \textsc{Expr}$ can be materialized to an abstract value $v \in \mathcal{V}$ that satisfies all constraints using function $\mathbb{G}_{\mathcal{D}} : \wp(\overline{\textsc{Be}}(\mathsf{e})) \to \mathcal{V}$.

$\mathbb{G}$ is thus able to generate an abstract element that soundly abstracts (i.e., over-approximates) the concrete values identified by the conjunction of the provided constraints. All domains taking part in the framework thus have to define $\mathbb{C}$ and $\mathbb{G}$ to be employed in constraint-based whole-value analyses. Formalizations and examples of both $\mathbb{C}$ and $\mathbb{G}$ are provided in Section 6. We are now in a position to define how multi-type expressions are evaluated within the framework. We express such evaluations in big-step notation.

$$\frac{-}{\wr \mathbf{len}(\mathsf{s}) \wr \mu^\sharp = \mathbb{G}_{\mathcal{A}^\sharp}(\{ \langle n \otimes \mathbf{len}(\mathsf{s}) \rangle \mid \langle n \otimes \mathbf{len}(\mathsf{s}) \rangle \in \mathbb{C}_{s^\sharp}(s^\sharp, \mathsf{s}) \})} \tag{5.1}$$

$$\frac{-}{\wr \mathsf{a}_1 \circledcirc \mathsf{a}_2 \wr \mu^\sharp = \mathbb{G}_{\mathcal{B}^\sharp}(\mathbb{C}_{\mathcal{A}^\sharp}(a^\sharp, \mathsf{a}_1 \circledcirc \mathsf{a}_2))} \tag{5.2}$$

$$\frac{-}{\wr \mathsf{s}_1 == \mathsf{s}_2 \wr \mu^\sharp = \mathbb{G}_{\mathcal{B}^\sharp}(\mathbb{C}_{\mathcal{S}^\sharp}(s^\sharp, \mathsf{s}_1 == \mathsf{s}_2))} \tag{5.3}$$

$$\frac{-}{\wr \mathbf{contains}(\mathsf{s}_1, \mathsf{s}_2) \wr \mu^\sharp = \mathbb{G}_{\mathcal{B}^\sharp}(\mathbb{C}_{\mathcal{S}^\sharp}(s^\sharp, \mathbf{contains}(\mathsf{s}_1, \mathsf{s}_2)))} \tag{5.4}$$

$$\frac{A_1 = \mathbb{C}_{\mathcal{A}^\sharp}(a^\sharp, \mathsf{a}_1) \qquad A_2 = \mathbb{C}_{\mathcal{A}^\sharp}(a^\sharp, \mathsf{a}_2)}{\wr \mathbf{substr}(\mathsf{s}, \mathsf{a}_1, \mathsf{a}_2) \wr \mu^\sharp = \wr \mathbf{substr}(\mathsf{s}, A_1, A_2) \wr^{\mathcal{S}^\sharp} s^\sharp} \tag{5.5}$$

$$\frac{e = \mathsf{a}_1 < 0 \ \mathbf{or}\ \mathsf{a}_2 >= \mathbf{len}(\mathsf{s}) \ \mathbf{or}\ \mathsf{a}_1 > \mathsf{a}_2 \qquad \langle \mathbf{true} == e \rangle \in \mathbb{C}_{\mathcal{A}^\sharp}(a^\sharp, e)}{\wr \mathbf{substr}(\mathsf{s}, \mathsf{a}_1, \mathsf{a}_2) \wr \mu^\sharp = \bot_{\mathcal{S}^\sharp}} \tag{5.6}$$

When all arguments are of a coherent type, the evaluation is straightforward: Equations 5.2–5.4 proceed by (i) generating constraints over the result using $\mathbb{C}$ on the appropriate domain instance ($a^\sharp$ or $s^\sharp$), and (ii) converting the constraints to a Boolean abstraction using $\mathbb{G}_{\mathcal{B}^\sharp}$. The evaluation of $\mathbf{len}(\mathsf{s})$ in Equation 5.1 is slightly different: the abstract domain $s^\sharp$ generates constraints on the expression s, and only the ones regarding its length are kept. Such constraints are then passed to $\mathbb{G}_{\mathcal{A}^\sharp}$ to build an abstract integer. The evaluation of $\mathbf{substr}(\mathsf{s}, \mathsf{a}_1, \mathsf{a}_2)$ is more complex: if one of the necessary conditions is violated, as described in Equation 5.6, the bottom element is generated. Instead, the domain-dependent semantics $\wr \mathbf{substr}(\mathsf{s}, A_1, A_2) \wr^{\mathcal{S}^\sharp}$ is invoked in Equation 5.5, taking in a description of the integer arguments in the form of sets of constraints. Note that, whenever the computation happens in a domain but produces a value of another type (as, e.g., in Equation 5.1), the cross-domain communication happens entirely through functions $\mathbb{C}$ and $\mathbb{G}$. Instead, when an operation has parameters of different types, the semantics of that operation must be redefined in terms of constraints, as in Equation 5.5.

Example 5.2. Let us consider the same setting of Example 5.1, where the framework is instantiated as $\langle \textsc{Id} \to \mathbb{Z}^\top_\bot \otimes \textsc{Id} \to$

$\Sigma^*_\perp \otimes \textsc{Id} \to \wp(\mathbb{B})\rangle$, and consider the same state $\mu^\sharp = (a^\sharp, s^\sharp, b^\sharp)$ where $a^\sharp = \{(x, 42)\}$, $s^\sharp = \{(y, \text{“foo”})\}$, and $b^\sharp = \{(z, \texttt{true})\}$. If we were to evaluate the assignment $k = \texttt{len}(y) + 1$, the framework would employ the multi-type expression semantics first: $\mathbb{C}_{\textsc{Pr}}(s^\sharp, y)$ evaluates, as will be discussed in Section 6.5, to the set $\{3 <= \texttt{len}(y)\}$ (intuitively, being “foo” a prefix, every string stored in y is guaranteed to have length at least 3, with no information on its maximum length). The set of constraints is then passed to the constant propagation domain, and $\mathbb{G}_{\textsc{Cp}}(\{3 <= \texttt{len}(y)\})$ yields, as will be defined in Section 6.1, $\top$ since the expression is not constant. Thus, the final state after the assignment is $\mu^\sharp_1 = (a^\sharp_1, s^\sharp, b^\sharp)$ where $a^\sharp_1 = \{(x, 42), (k, \top)\}$.

Example 5.3. Let us change the setting of Example 5.2 by using the interval domain as numerical abstraction. The framework is now instantiated as $\langle \textsc{Id} \to \mathbb{I} \otimes \textsc{Id} \to \Sigma^*_\perp \otimes \textsc{Id} \to \wp(\mathbb{B})\rangle$, where $\mathbb{I}$ is the set of intervals as defined in Section 6.2. Let the starting state be $\mu^\sharp = (a^\sharp, s^\sharp, b^\sharp)$ where $a^\sharp = \{(x, [42, 42])\}$, $s^\sharp = \{(y, \text{“foo”})\}$, and $b^\sharp = \{(z, \texttt{true})\}$. If we evaluate once more the assignment $k = \texttt{len}(y) + 1$, the evaluation only changes in the generation of the interval element: function $\mathbb{G}_{\textsc{Intv}}(\{3 <= \texttt{len}(y)\})$ yields, as formalized in Section 6.2, $[3, +\infty]$. Thus, the final state after the assignment is $\mu^\sharp_2 = (a^\sharp_2, s^\sharp, b^\sharp)$ where $a^\sharp_2 = \{(x, 42), [4, +\infty]\}$.

5.4 Soundness of the multi-type semantics

Having modified how multi-type expressions are abstracted, we have to ensure that those definitions are sound for the soundness of the abstract semantics to hold. To prove this, we assume that the definitions of $\mathbb{C}$ and $\mathbb{G}$ provided by the domains are sound, relying on the monotone concretization function $\gamma_{\overline{\textsc{Be}}(\mathsf{e})}$.

Definition 16 (Soundness of $\mathbb{C}$)**.** Given an abstract domain $\mathcal{D}$, function $\mathbb{C}_\mathcal{D}$ is sound if the conjunction of the produced constraints soundly approximate the concrete values of the expression, that is, if $\forall \mathsf{e} \in \textsc{Expr}, d^\sharp \in \mathcal{D} : \gamma_\mathcal{D}(\wr \mathsf{e} \wr d^\sharp) \subseteq \gamma_{\overline{\textsc{Be}}(\mathsf{e})}(\mathbb{C}_\mathcal{D}(d^\sharp, \mathsf{e}))$.

Definition 17 (Soundness of $\mathbb{G}$)**.** Given an abstract domain $\mathcal{D}$, function $\mathbb{G}_\mathcal{D}$ is sound if the generated element over-approximates the concrete elements identified by the constraints, that is, if $\forall \mathsf{e} \in \textsc{Expr}, B \subseteq \overline{\textsc{Be}}(\mathsf{e}) : \gamma_{\overline{\textsc{Be}}(\mathsf{e})}(B) \subseteq \gamma_\mathcal{D}(\mathbb{G}_\mathcal{D}(B))$.

Soundness of $\mathbb{C}$ is thus ensured if all possible concrete values of the expression satisfy all the generated constraints. Instead, $\mathbb{G}$ is sound if all the concrete values identified by the constraints are abstracted by the generated element. We now define the criteria for the soundness of the substring semantics $\wr \texttt{substr}(\mathsf{s}, A_1, A_2) \wr^{\mathcal{S}^\sharp}$.

Definition 18 (Soundness of $\wr \texttt{substr}(\mathsf{s}, A_1, A_2) \wr^{\mathcal{S}^\sharp}$)**.** Given a string abstract domain $\mathcal{S}^\sharp$ whose expression semantics produces elements of the complete partial order $\mathcal{V}$, the function $\wr \texttt{substr}(\mathsf{s}, A_1, A_2) \wr^{\mathcal{S}^\sharp}$ is sound if the generated abstract element is a sound approximation of all the possible substrings evaluated in the concrete. Formally, the function is sound if $\forall \mathsf{e} \in \textsc{Expr}, A_1, A_2 \in \overline{\textsc{Be}}(\mathsf{e}), s^\sharp \in \mathcal{S}^\sharp : \{\sigma[i:j] \mid \sigma \in \gamma_{\mathcal{S}^\sharp}(\wr \mathsf{s} \wr^{\mathcal{S}^\sharp} s^\sharp) \wedge i \in \gamma_{\overline{\textsc{Be}}(\mathsf{e})}(A_1) \wedge j \in \gamma_{\overline{\textsc{Be}}(\mathsf{e})}(A_2)\} \subseteq \gamma_{\mathcal{S}^\sharp}(\wr \texttt{substr}(\mathsf{s}, A_1, A_2) \wr^{\mathcal{S}^\sharp} s^\sharp)$.

We can now reason on the soundness of the abstract transformers. Recall that, to be employed in our framework, an abstract domain must provide sound definitions of $\wr \mathsf{st} \wr$ and $\wr \mathsf{e} \wr$. These are then applied element-wise on the abstract memory of each domain (recall that $\wr \mathsf{st} \wr(a^\sharp, s^\sharp, b^\sharp) = (\wr \mathsf{st} \wr^{\mathcal{A}^\sharp} a^\sharp, \wr \mathsf{st} \wr^{\mathcal{S}^\sharp} s^\sharp, \wr \mathsf{st} \wr^{\mathcal{B}^\sharp} b^\sharp)$, and $\wr \mathsf{e} \wr$ delegates to the appropriate domain for single-type expressions). Thus, the evaluation of statements and single-type expressions is trivially sound, provided that the multi-type expression evaluation is sound as well. Assuming all domains provide sound $\mathbb{C}$ and $\mathbb{G}$, and that the string domain provides a sound $\wr \texttt{substr}(\mathsf{s}, A_1, A_2) \wr^{\mathcal{S}^\sharp}$, we now define the soundness of the multi-type semantics, which we generically denote with $\wr \mathsf{e} \wr$.

Theorem 19 (Soundness of $\wr \mathsf{e} \wr$)**.** The evaluation of multi-type expressions $\wr \mathsf{e} \wr$ is a sound approximation of the split collecting semantics $(\!|\mathsf{e}|\!)^+$. Formally, $\forall \mathsf{e} \in \textsc{Expr} : (\!|\mathsf{e}|\!)^+ \gamma_{\mathcal{M}^\sharp}(\mu^\sharp) \subseteq \gamma_{\mathcal{M}^\sharp}(\wr \mathsf{e} \wr \mu^\sharp)$. Proof in Appendix A.4.

6 Instantiation

In this section, we provide definitions of $\mathbb{C}$, $\mathbb{G}$, and $\wr \texttt{substr}(\mathsf{s}, A_1, A_2) \wr^{\mathcal{S}^\sharp}$ for domains commonly used in whole-value analyses. Thanks to our definitions, such domains can be implemented in a static analyzer modularly, and several of their combinations can be tested without modifying their code. The choice of the abstractions is guided by the literature on string analyses since, as will be discussed in section 8, they are the most common source of whole-value analyses definitions. We thus select:

- the constant propagation and interval (Cousot and Cousot, 1977) domains as numeric abstractions;
- the powerset abstraction for Boolean values;
- the bounded string set (Madsen and Andreasen, 2014), the prefix (Costantini et al., 2015), and the Tarsis (Negrini et al., 2021) domains for string abstractions.

The rationale behind this choice is that constant propagation and intervals are the most common numeric abstractions used in string analyses, while powerset abstraction is the only employed Boolean abstraction. Bounded string set, prefix, and Tarsis are instead domains tracking increasingly complex string information, showcasing how our framework can be instantiated on different levels of complexity. All domains that will be discussed are non-relational, meaning that they are defined as maps from program variables to abstract elements. To unify the notation, in each definition we will use $\top$ to define maps having all keys mapped to the top abstract element, and $\perp$ to indicate an error state.

6.1 Constant propagation

The constant propagation domain $\textsc{Cp}$ is a simple non-relational domain whose lattice elements are maps from program variables to integer values. The abstract domain is defined as follows:

$$\textsc{Cp} = \langle \textsc{Id} \to \mathbb{Z}^\top_\perp, \dot{\leq}, \dot{\sqcup}, \dot{\sqcap}, \top, \perp \rangle,$$

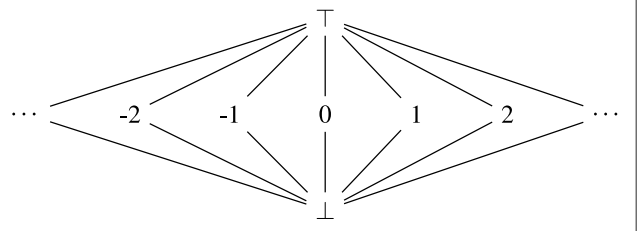

FIGURE 2
Hasse diagram of the numeric constant propagation values $\mathbb{Z}_{\perp}^{\top}$.

where the co-domain of the maps is $\mathbb{Z}$ extended with $\top$ (for non-constant values) and $\perp$ (for erroneous values), and $\dot{\leq}$, $\dot{\sqcup}$, and $\dot{\sqcap}$ are point-wise applications of the operators that can be inferred by the Hasse diagram of $\mathbb{Z}_{\perp}^{\top}$, visible in Figure 2. We now define functions $\mathbb{C}_{\text{CP}}$ and $\mathbb{G}_{\text{CP}}$.

Definition 20 (Constraint generation for CP). Let $\mu^{\sharp} \in \text{CP}$ be a mapping from program variables to integer constants. Function $\mathbb{C}_{\text{CP}} : \text{CP} \times \text{EXPR} \to \wp(\overline{\text{BE}}(\text{e}))$ is defined as:

$$\mathbb{C}_{\text{CP}}(\mu^{\sharp}, \text{e}) = \begin{cases} \emptyset & \text{if } \wr \text{e} \wr^{\text{CP}} \mu^{\sharp} = \top; \\ \{\langle v == \text{e} \rangle\} & \text{if } \wr \text{e} \wr^{\text{CP}} \mu^{\sharp} = v \in \mathbb{Z}; \\ \lightning & \text{if } \wr \text{e} \wr^{\text{CP}} \mu^{\sharp} = \perp. \end{cases}$$

$\mathbb{C}_{\text{CP}}$ thus generates (i) no constraints if the expression does not have a constant value, (ii) a constraint binding the exact value of the expression if it is constant, or (iii) $\lightning$ if the evaluation of e leads to an error.

Definition 21 (Constraint interpretation for CP). Let $B \subseteq \overline{\text{BE}}(\text{e})$ be a set of constraints. Function $\mathbb{G}_{\text{CP}} : \wp(\overline{\text{BE}}(\text{e})) \to \mathbb{Z}_{\perp}^{\top}$ is defined as:

$$\mathbb{G}_{\text{CP}}(B) = \begin{cases} \perp & \text{if } B = \lightning; \\ v & \text{if } \langle v == \text{e} \rangle \in B \vee \{\langle v >= \text{e} \rangle, \langle v <= \text{e} \rangle\} \subseteq B; \\ \top & \text{otherwise.} \end{cases}$$

$\mathbb{G}_{\text{CP}}$ thus returns $\perp$ if the set of constraints is $\lightning$, the constant value v if the constraints bind the value of the expression to v, and $\top$ otherwise. Soundness of $\mathbb{C}_{\text{CP}}$ and $\mathbb{G}_{\text{CP}}$ is proven in Appendix A.5.1.

6.2 Intervals

The intervals domain INTV (Cousot and Cousot, 1977) is a non-relational domain whose lattice elements are maps from program variables to the ranges $[l, u]$ their values might take, with $l \in \mathbb{Z} \cup \{-\infty\}$ and $u \in \mathbb{Z} \cup \{+\infty\}$. The abstract domain is defined as follows:

$$\text{INTV} = \langle \text{ID} \to \mathbb{I}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \top, \perp \rangle,$$

where the co-domain of the maps is $\mathbb{I} \triangleq (\mathbb{Z} \cup \{-\infty\} \times \mathbb{Z} \cup \{+\infty\}) \cup \perp$ corresponding to all possible intervals plus $\perp$ to model erroneous values. Lattice operators are point-wise applications of the operators that can be inferred by the Hasse diagram of $\mathbb{I}$, visible in Figure 3. We now define $\mathbb{C}_{\text{INTV}}$ and $\mathbb{G}_{\text{INTV}}$.

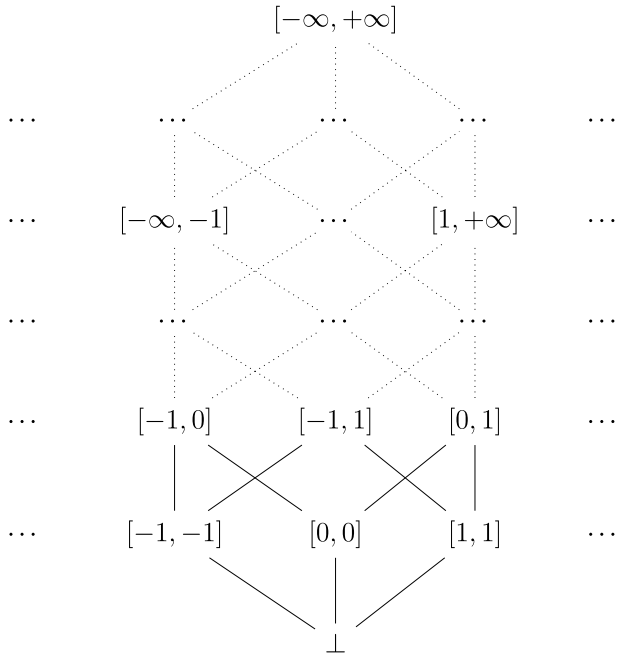

FIGURE 3
Hasse diagram of the intervals $\mathbb{I}$.

Definition 22 (Constraint generation for INTV). Let $\mu^{\sharp} \in \text{INTV}$ be a mapping from program variables to intervals. Function $\mathbb{C}_{\text{INTV}} : \text{INTV} \times \text{EXPR} \to \wp(\overline{\text{BE}}(\text{e}))$ is defined as:

$$\mathbb{C}_{\text{INTV}}(\mu^{\sharp}, \text{e}) = \begin{cases} \emptyset & \text{if } \wr \text{e} \wr^{\text{INTV}} \mu^{\sharp} = [-\infty, +\infty]; \\ \{\langle l <= \text{e} \rangle\} & \text{if } \wr \text{e} \wr^{\text{INTV}} \mu^{\sharp} = [l, +\infty], l \in \mathbb{Z}; \\ \{\langle u >= \text{e} \rangle\} & \text{if } \wr \text{e} \wr^{\text{INTV}} \mu^{\sharp} = [-\infty, u], u \in \mathbb{Z}; \\ \{\langle l <= \text{e} \rangle, & \\ \langle u >= \text{e} \rangle\} & \text{if } \wr \text{e} \wr^{\text{INTV}} \mu^{\sharp} = [l, u], l, u \in \mathbb{Z}; \\ \lightning & \text{if } \wr \text{e} \wr^{\text{INTV}} \mu^{\sharp} = \perp. \end{cases}$$

$\mathbb{C}_{\text{INTV}}$ thus generates (i) no constraints if the expression can evaluate to any value, (ii) a constraint for each finite bound of the interval produced by evaluating e, or (iii) $\lightning$ if the evaluation of e leads to an error.

Definition 23 (Constraint interpretation for INTV). Let $B \subseteq \overline{\text{BE}}(\text{e})$ be a set of constraints. Function $\mathbb{G}_{\text{INTV}} : \wp(\overline{\text{BE}}(\text{e})) \to \mathbb{I}$ is defined as:

$$\mathbb{G}_{\text{INTV}}(B) = \begin{cases} \perp & \text{if } B = \lightning; \\ [v, v] & \text{if } \langle v == \text{e} \rangle \in B; \\ [l, u] & \text{if } \{\langle l <= \text{e} \rangle, \langle u >= \text{e} \rangle\} \subseteq B; \\ [l, +\infty] & \text{if } \langle l <= \text{e} \rangle \in B \wedge \nexists u \in \mathbb{Z} : \langle u >= \text{e} \rangle \in B; \\ [-\infty, u] & \text{if } \langle u >= \text{e} \rangle \in B \wedge \nexists l \in \mathbb{Z} : \langle l <= \text{e} \rangle \in B; \\ [-\infty, +\infty] & \text{otherwise.} \end{cases}$$

$\mathbb{G}_{\text{INTV}}$ thus returns $\perp$ if the set of constraints is $\lightning$, and the interval corresponding to the range of possible values of

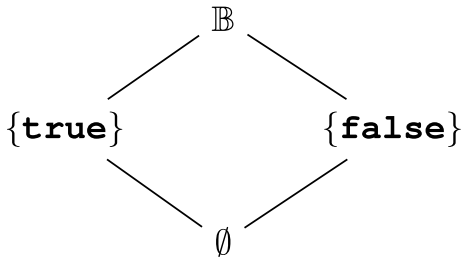

FIGURE 4
Hasse diagram of the Boolean powerset $\wp(\mathbb{B})$.

e otherwise. Soundness of $\mathbb{C}_{\text{INTV}}$ and $\mathbb{G}_{\text{INTV}}$ is proven in Appendix A.5.2.

6.3 Boolean powerset

The boolean powerset domain BP is a non-relational domain whose lattice elements are maps from program variables to subsets of the Boolean values $\mathbb{B}$. The abstract domain is defined as follows:

$$\text{BP} = \langle \text{ID} \to \wp(\mathbb{B}), \dot{\subseteq}, \dot{\cup}, \dot{\cap}, \top, \bot \rangle,$$

where the co-domain of the maps is $\wp(\mathbb{B})$. Lattice operators are point-wise applications of the operators that can be inferred by the Hasse diagram of $\wp(\mathbb{B})$, visible in Figure 4. $\mathbb{C}_{\text{BP}}$ and $\mathbb{G}_{\text{BP}}$ are defined as follows.

Definition 24 (Constraint generation for BP). Let $\mu^\sharp \in \text{BP}$ be a mapping from program variables to sets of Booleans. Function $\mathbb{C}_{\text{BP}} : \text{BP} \times \text{EXPR} \to \wp(\overline{\text{BE}}(\text{e}))$ is defined as:

$$\mathbb{C}_{\text{BP}}(\mu^\sharp, \text{e}) = \begin{cases} \emptyset & \text{if } (\!|\text{e}|\!)^{\text{BP}}\mu^\sharp = \mathbb{B}; \\ \{\langle \textbf{true} == \text{e} \rangle\} & \text{if } (\!|\text{e}|\!)^{\text{BP}}\mu^\sharp = \{\textbf{true}\}; \\ \{\langle \textbf{false} == \text{e} \rangle\} & \text{if } (\!|\text{e}|\!)^{\text{BP}}\mu^\sharp = \{\textbf{false}\}; \\ \lightning & \text{if } (\!|\text{e}|\!)^{\text{BP}}\mu^\sharp = \bot. \end{cases}$$

$\mathbb{C}_{\text{BP}}$ thus generates (i) no constraints if the expression can evaluate to any value, (ii) a constraint binding the expression to its Boolean value as determined by the domain if it can only assume one value, or (iii) $\lightning$ if evaluating the expression leads to an error.

Definition 25 (Constraint interpretation for BP). Let $B \subseteq \overline{\text{BE}}(\text{e})$ be a set of constraints. Function $\mathbb{G}_{\text{BP}} : \wp(\overline{\text{BE}}(\text{e})) \to \wp(\mathbb{B})$ is defined as:

$$\mathbb{G}_{\text{BP}}(B) = \begin{cases} \emptyset & \text{if } B = \lightning; \\ \{b\} & \text{if } \langle b == \text{e} \rangle \in B; \\ \mathbb{B} & \text{otherwise.} \end{cases}$$

$\mathbb{G}_{\text{BP}}$ thus returns $\emptyset$ if the set of constraints is $\lightning$, the set $\{b\}$ if the expression is constrained to the Boolean value b, or the top element $\mathbb{B}$ otherwise. Soundness of $\mathbb{C}_{\text{BP}}$ and $\mathbb{G}_{\text{BP}}$ is proven in Appendix A.5.3.

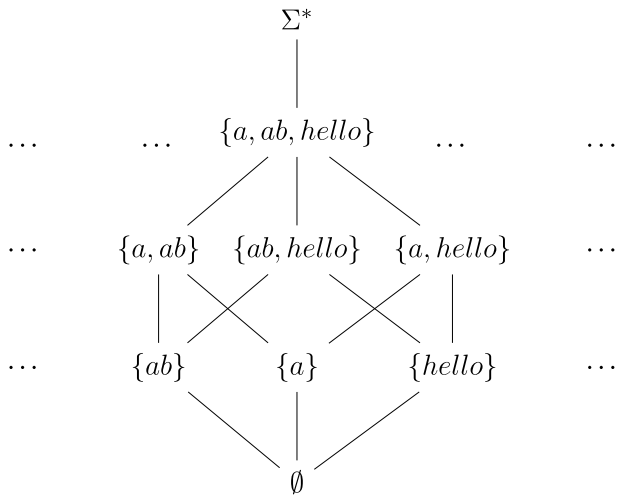

FIGURE 5
Hasse diagram of the bounded set of strings $\wp_3(\Sigma^*)$.

6.4 Bounded string set

The bounded string set domain SS (Madsen and Andreasen, 2014) is a non-relational domain whose lattice elements are maps from program variables to bounded sets of up to k strings. The abstract domain is defined as:

$$\text{SS} = \langle \text{ID} \to \wp_k(\Sigma^*), \dot{\subseteq}, \dot{\cup}_k, \dot{\cap}, \top, \bot \rangle,$$

where the co-domain of the maps is $\wp_k(\Sigma^*) \subset \wp(\Sigma^*)$, that is, the powerset $\wp(\Sigma^*)$ where all sets with cardinality greater than k have been removed. An example of such a powerset, with $k = 3$, is visible in Figure 5. Σ^* represents an unknown string (i.e., a set with more than k elements), while $\emptyset$ represents erroneous values. Lattice operators $\dot{\subseteq}$, $\dot{\cup}_k$, and $\dot{\cap}$ are point-wise applications of the respective set-theoretic operators, with $\cup_k$ defined as:

$$A \cup_k B \triangleq \begin{cases} A \cup B & \text{if } |A \cup B| \leq k; \\ \Sigma^* & \text{otherwise.} \end{cases}$$

We now define functions $\mathbb{C}_{\text{SS}}$ and $\mathbb{G}_{\text{SS}}$.

Definition 26 (Constraint generation for SS). Let $\mu^\sharp \in \text{SS}$ be a mapping from program variables to bounded sets of strings. Function $\mathbb{C}_{\text{SS}} : \text{SS} \times \text{EXPR} \to \wp(\overline{\text{BE}}(\text{e}))$ is defined as:

$$\mathbb{C}_{\text{SS}}(\mu^\sharp, \text{e}) = \begin{cases} \{\langle 0 <= \textbf{len}(\text{e}) \rangle\} & \text{if } (\!|\text{e}|\!)^{\text{SS}}\mu^\sharp = \\ & \wp_k(\Sigma^*); \\ \left\{ \begin{array}{l} \langle \min_{i \in [1..n]} |\sigma_i| <= \textbf{len}(\text{e}) \rangle, \\ \langle \max_{i \in [1..n]} |\sigma_i| >= \textbf{len}(\text{e}) \rangle, \\ \langle \text{gcp}\{\sigma_1, \ldots, \sigma_n\} \curvearrowright_p \text{e} \rangle, \\ \langle \text{gcs}\{\sigma_1, \ldots, \sigma_n\} \curvearrowright_s \text{e} \rangle \end{array} \right\} & \text{if } (\!|\text{e}|\!)^{\text{SS}}\mu^\sharp = \\ & \{\sigma_1, \ldots, \sigma_n\}, n \leq k; \\ \lightning & \text{if } (\!|\text{e}|\!)^{\text{SS}}\mu^\sharp = \emptyset. \end{cases}$$

$\mathbb{C}_{\text{SS}}$ thus generates (i) a constraint indicating that the length of the expression is non-negative if the expression has more than k possible values, (ii) constraints binding (a) the prefix of the string to the greatest common prefix (gcp) of the set, (b) the suffix of the string to the greatest common suffix (gcs) of the set, and (c) bounds on the length of the string corresponding to the length of the shortest and longest string in the set, if it can have at most k possible values, or (iii) $\lightning$ if evaluating the expression leads to an error. Note that we always produce length constraints when possible, to provide information to numerical domains that might use it.

Definition 27 (Constraint interpretation for SS). Let $B \subseteq \overline{\text{BE}}(\mathsf{e})$ be a set of constraints. Function $\mathbb{G}_{\text{SS}} : \wp(\overline{\text{BE}}(\mathsf{e})) \to \wp_k(\Sigma^*)$ is defined as:

$$\mathbb{G}_{\text{SS}}(B) = \begin{cases} \emptyset & \text{if } B = \lightning; \\ \{\sigma\} & \text{if } \langle \sigma == \mathsf{e} \rangle \in B; \\ \Sigma^* & \text{otherwise.} \end{cases}$$

$\mathbb{G}_{\text{SS}}$ thus returns $\emptyset$ if the set of constraints is $\lightning$, the set $\{\sigma\}$ if the constraints bind the value of the expression to σ, and Σ^* otherwise. Note that, since the constraint set holds definite information, there cannot be more than one constraint using $==$, and constraints on prefix and suffix yield infinitely many strings: we thus cannot generate sets with more than one concrete string. We can now provide the definition of $\wr \mathbf{substr}(\mathsf{s}, A_1, A_2) \wr^{\mathcal{S}^\sharp}$. Since SS can only model bounded sets, we have to ensure that the resulting set holds at most k substrings; otherwise, the semantics returns Σ^* if no evaluation error happens.

Definition 28 (Substring semantics for SS). Let $\mu^\sharp \in \text{SS}$ be a mapping from program variables to bounded sets of strings, and let $A_1 \subseteq \overline{\text{BE}}(\mathsf{e}_1)$ and $A_2 \subseteq \overline{\text{BE}}(\mathsf{e}_2)$ be sets of constraints describing integer expressions e_1 and e_2, respectively. Function $\wr \mathbf{substr}(\mathsf{s}, A_1, A_2) \wr^{\text{SS}}$ is defined as:

$$\wr \mathbf{substr}(\mathsf{s}, A_1, A_2) \wr^{\text{SS}} \mu^\sharp =$$

$$\begin{cases} \emptyset & \text{if } \wr \mathsf{s} \wr^{\text{SS}} \mu^\sharp = \emptyset \vee A_1 = \lightning \vee A_2 = \lightning; \\ \left\{ \sigma_w[x:y] \;\middle|\; \begin{array}{l} 1 \le w \le n, \\ i_l \le x \le i_h, \\ j_l \le y \le j_h, \\ 0 \le x \le y \le |\sigma_w| \end{array} \right\} & \text{if } \wr \mathsf{s} \wr^{\text{SS}} \mu^\sharp = \{\sigma_1, \ldots, \sigma_n\} \\ & \wedge \{\langle i_l <= \mathsf{e}_1 \rangle, \langle i_h >= \mathsf{e}_1 \rangle\} \\ & \subseteq A_1 \\ & \wedge \{\langle j_l <= \mathsf{e}_2 \rangle, \langle j_h >= \mathsf{e}_2 \rangle\} \\ & \subseteq A_2 \\ & \wedge \mathsf{count}(\mu^\sharp, \mathsf{s}, A_1, A_2) \le k; \\ \Sigma^* & \text{otherwise,} \end{cases}$$

where function count returns the number of valid substrings that can be computed through its parameters (the definition is left implicit, but intuitively it returns the cardinality of the set returned in the second case) and, for the sake of readability, we omit the cases when at least one of i or j are constants (i.e., when $\langle i == \mathsf{e}_1 \rangle \in A_1$ and $\langle j == \mathsf{e}_2 \rangle \in A_2$, respectively), as they can be assimilated to the central case when both inequalities have the same bound. We also omit the case where either i or j is unbounded (i.e., where no upper bound on their value is present in the respective constraint set), as they will generate infinitely many substrings and will collapse the result to Σ^*.

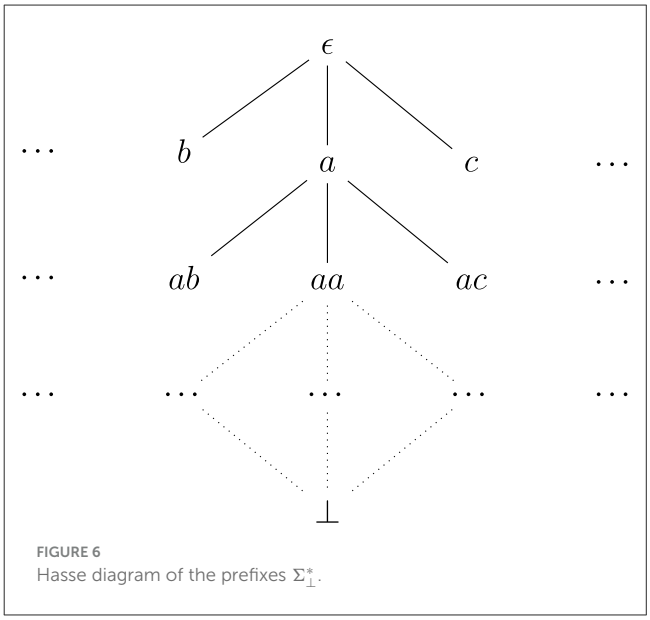

FIGURE 6
Hasse diagram of the prefixes $\Sigma^*_\bot$.

Soundness of $\mathbb{C}_{\text{SS}}$, $\mathbb{G}_{\text{SS}}$, and $\wr \mathbf{substr}(\mathsf{s}, A_1, A_2) \wr^{\text{SS}}$ is proven in Appendix A.5.4.

6.5 Prefix

The prefix domain PR (Costantini et al., 2015) is a non-relational domain whose lattice elements are maps from program variables to strings acting as prefixes. The abstract domain is defined as:

$$\text{PR} = \langle \text{ID} \to \Sigma^*_\bot, \dot{\sqsubseteq}, \dot{\text{gcp}}, \dot{\sqcap}, \top, \bot \rangle,$$

where the co-domain of the maps is $\Sigma^*_\bot$, whose Hasse diagram is visible in Figure 6. Elements are ordered according to the reverse prefix relation: $\sigma_1 \sqsubseteq \sigma_2 \iff \sigma_2 \curvearrowright_p \sigma_1$. The empty string ϵ thus represents an unknown string (since $\forall \sigma \in \Sigma^* : \epsilon \curvearrowright_p \sigma$), while $\bot$ represents erroneous values. Lattice operators $\dot{\sqsubseteq}$, $\dot{\text{gcp}}$, and $\dot{\sqcap}$ are point-wise applications of the respective operators over $\Sigma^*_\bot$, with gcp and $\sqcap$ defined as:

$$\sigma_1 \text{ gcp } \sigma_2 \triangleq \begin{cases} \sigma_1 & \text{if } \sigma_1 \curvearrowright_p \sigma_2; \\ \sigma_2 & \text{if } \sigma_2 \curvearrowright_p \sigma_1; \\ \sigma_p & \text{otherwise, with } \sigma_1 = \sigma_p \sigma' \\ & \wedge\, \sigma_2 = \sigma_p \sigma'' \wedge \sigma'_0 \neq \sigma''_0, \end{cases} \quad \text{and}$$

$$\sigma_1 \sqcap \sigma_2 \triangleq \begin{cases} \sigma_1 & \text{if } \sigma_1 \curvearrowright_p \sigma_2; \\ \sigma_2 & \text{if } \sigma_2 \curvearrowright_p \sigma_1; \\ \bot & \text{otherwise.} \end{cases}$$

We now define functions $\mathbb{C}_{\text{PR}}$ and $\mathbb{G}_{\text{PR}}$.

Definition 29 (Constraint generation for Pr). Let $\mu^\sharp \in \textsc{Pr}$ be a mapping from program variables to string prefixes. Function $\mathbb{C}_{\textsc{Pr}} : \textsc{Pr} \times \textsc{Expr} \to \wp(\overline{\textsc{Be}}(\mathsf{e}))$ is defined as:

$$\mathbb{C}_{\textsc{Pr}}(\mu^\sharp, \mathsf{e}) = \begin{cases} \{\langle 0 <= \textbf{len}(\mathsf{e})\rangle\} & \text{if } \wr \mathsf{e} \wr^{\textsc{Pr}} \mu^\sharp = \epsilon; \\ \{\langle |\sigma| <= \textbf{len}(\mathsf{e})\rangle, \sigma \frown_p \mathsf{e}\} & \text{if } \wr \mathsf{e} \wr^{\textsc{Pr}} \mu^\sharp = \sigma; \\ \lightning & \text{if } \wr \mathsf{e} \wr^{\textsc{Pr}} \mu^\sharp = \bot. \end{cases}$$

$\mathbb{C}_{\textsc{Pr}}$ thus generates (i) a constraint indicating that the length of the expression is non-negative if the expression can assume any string value, (ii) constraints binding (a) the prefix of the string to the result of the evaluation, and (b) the lower bound on the length of the string corresponding to the length of the prefix, if the evaluation produces a valid string as a result, or (iii) $\lightning$ if evaluating the expression leads to an error. Note that we always produce length constraints when possible provide information to numerical domains that might use it.

Definition 30 (Constraint interpretation for Pr). Let $B \subseteq \overline{\textsc{Be}}(\mathsf{e})$ be a set of constraints. Function $\mathbb{G}_{\textsc{Pr}} : \wp(\overline{\textsc{Be}}(\mathsf{e})) \to \Sigma^*_\bot$ is defined as:

$$\mathbb{G}_{\textsc{Pr}}(B) = \begin{cases} \bot & \text{if } B = \lightning; \\ \sigma & \text{if } \langle \sigma == \mathsf{e}\rangle \in B \vee \langle \sigma \frown_p \mathsf{e}\rangle \in B; \\ \epsilon & \text{otherwise.} \end{cases}$$

$\mathbb{G}_{\textsc{Pr}}$ thus returns $\bot$ if the set of constraints is $\lightning$, the prefix σ if the constraints bind either the value of the expression or its prefix to σ, and ϵ otherwise. We can now provide the definition of $\wr \textbf{substr}(\mathsf{s}, A_1, A_2) \wr^{\mathcal{S}^\sharp}$.

Definition 31 (Substring semantics for Pr). Let $\mu^\sharp \in \textsc{Pr}$ be a mapping from program variables to string prefixes, and let $A_1, A_2 \subseteq \overline{\textsc{Be}}(\mathsf{e})$ be sets of constraints describing integer expressions. Furthermore, let us denote as $\bar{i}$ the minimal non-negative value admitted by A_1, and by $\bar{j}$ the minimal non-negative value admitted by A_2 that is greater than or equal to $\bar{i}$. Function $\wr \textbf{substr}(\mathsf{s}, A_1, A_2) \wr^{\textsc{Pr}}$ is defined as:

$$asem\textbf{substr}(\mathsf{s}, A_1, A_2)^{\textsc{Pr}} \mu^\sharp = \begin{cases} \bot & \text{if } \wr \mathsf{e} \wr^{\textsc{Pr}} \mu^\sharp = \bot \vee A_1 = \lightning \vee A_2 = \lightning; \\ \sigma[\bar{i} : \bar{j}] & \text{if } \wr \mathsf{e} \wr^{\textsc{Pr}} \mu^\sharp = \sigma \wedge \bar{i} \leq \bar{j} \leq |\sigma|; \\ \sigma[\bar{i} : |\sigma| - 1] & \text{if } \wr \mathsf{e} \wr^{\textsc{Pr}} \mu^\sharp = \sigma \wedge \bar{i} \leq |\sigma| < \bar{j}; \\ \epsilon & \text{if } \wr \mathsf{e} \wr^{\textsc{Pr}} \mu^\sharp = \sigma \wedge |\sigma| < \bar{i} \leq \bar{j}. \end{cases}$$

$\wr \textbf{substr}(\mathsf{s}, A_1, A_2) \wr^{\textsc{Pr}}$ thus shortens the approximation for the expression if (part of) the substring lies within the prefix, truncating it to ϵ otherwise. Soundness of $\mathbb{C}_{\textsc{Pr}}$, $\mathbb{G}_{\textsc{Pr}}$, and $\wr \textbf{substr}(\mathsf{s}, A_1, A_2) \wr^{\textsc{Pr}}$ is proven in Appendix A.5.5.

6.6 Tarsis

The Tarsis domain Ta (Negrini et al., 2021) is a non-relational domain whose lattice elements are maps from program variables to special finite state automata defined over an alphabet of strings instead of single characters. The abstract domain is defined as:

$$\textsc{Ta} = \langle \textsc{Id} \to \mathcal{T}\textsc{Fa}_{/\equiv}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \top, \bot \rangle,$$

where the co-domain of the maps is the set of equivalence classes of finite state automata with string alphabets $\mathcal{T}\textsc{Fa}_{/\equiv}$, and whose lattice operators are point-wise applications of the respective operators over $\mathcal{T}\textsc{Fa}_{/\equiv}$. Specifically, $\sqsubseteq$ is the partial order induced by language inclusion, $\sqcup$ and $\sqcap$ correspond to automata union and intersection, respectively, and the top and bottom elements are $\mathsf{Min}(\mathbb{A}^*_{\mathsf{p}})$ and $\mathsf{Min}(\emptyset)$ (respectively, the minimum automata recognizing all possible strings and the one recognizing the empty language). Following the notation from Negrini et al. (2021), we denote as $\mathsf{A} \in \mathcal{T}\textsc{Fa}_{/\equiv}$ a Tarsis automaton, and as $\mathscr{L}(\mathsf{A}) \in \wp(\Sigma^*)$ the regular language recognized by A. We now define functions $\mathbb{C}_{\textsc{Ta}}$ and $\mathbb{G}_{\textsc{Ta}}$.

Definition 32 (Constraint generation for Ta). Let $\mu^\sharp \in \textsc{Ta}$ be a mapping from program variables to $\mathcal{T}\textsc{Fa}_{/\equiv}$ automata. Function $\mathbb{C}_{\textsc{Ta}} : \textsc{Ta} \times \textsc{Expr} \to \wp(\overline{\textsc{Be}}(\mathsf{e}))$ is defined as:

$$\mathbb{C}_{\textsc{Ta}}(\mu^\sharp, \mathsf{e}) = \begin{cases} \{\langle 0 <= \textbf{len}(\mathsf{e})\rangle\} & \text{if } \wr \mathsf{e} \wr^{\textsc{Ta}} \mu^\sharp = \mathsf{Min}(\mathbb{A}^*_{\mathsf{p}}); \\ \{\langle \sigma == \mathsf{e}\rangle, \langle |\sigma| <= \textbf{len}(\mathsf{e})\rangle, \langle |\sigma| >= \textbf{len}(\mathsf{e})\rangle\} & \text{if } \mathscr{L}(\wr \mathsf{e} \wr^{\textsc{Ta}} \mu^\sharp) = \{\sigma\}; \\ \left\{ \begin{array}{l} \langle \mathsf{lcp}(\mathsf{A}) \frown_p \mathsf{e}\rangle, \\ \langle \mathsf{lcp}(\mathsf{rev}(\mathsf{A})) \frown_s \mathsf{e}\rangle, \\ \langle i <= \textbf{len}(\mathsf{e})\rangle, \\ \langle j <= \textbf{len}(\mathsf{e})\rangle \end{array} \right\} & \text{if } \wr \mathsf{e} \wr^{\textsc{Ta}} \mu^\sharp = \mathsf{A} \wedge \mathsf{len}(\mathsf{A}) = [i, j]; \\ \lightning & \text{if } \wr \mathsf{e} \wr^{\textsc{Ta}} \mu^\sharp = \mathsf{Min}(\emptyset). \end{cases}$$

$\mathbb{C}_{\textsc{Ta}}$ thus generates (i) a constraint indicating that the length of the expression is non-negative if the expression can assume any string value, (ii) constraints binding the exact value and length of the string if it can assume exactly one string value, (iii) constraints binding prefix (where lcp is the largest common prefix, e.g., the one proposed in Béal and Carton, 2000), suffix (where $\mathsf{rev}(\mathsf{A})$ reverses an automaton by swapping initial and accepting states and inverting the direction of edges), minimum length and maximum length of the string (where $\mathsf{len}(\mathsf{A})$ is the abstract semantics of **len** as defined in Negrini et al., 2021) according to the result of the evaluation, or (iv) $\lightning$ if evaluating the expression leads to an error. For the sake of readability, the case when $j = +\infty$ is not shown as it is equal to the displayed case, except that the bound using j is absent. Note that we always produce length constraints when possible provide information to numerical domains that might use it.

Definition 33 (Constraint interpretation for Ta). Let $B \subseteq \overline{\textsc{Be}}(\mathsf{e})$ be a set of constraints, let $\mathscr{L}(\sigma)$ be the minimum automata $\mathsf{A} \in \mathcal{T}\textsc{Fa}_{/\equiv}$ recognizing the regular language $\{\sigma\}$, and let $\frown$ be the automata

concatenation. Function $\mathbb{G}_{\text{TA}} : \wp(\overline{\text{BE}}(\text{e})) \to \Sigma^*_\perp$ is defined as:

$$\mathbb{G}_{\text{TA}}(B) = \begin{cases} \text{Min}(\emptyset) & \text{if } B = \lightning; \\ \mathscr{L}(\sigma) & \text{if } \langle \sigma == \text{e} \rangle \in B; \\ \mathscr{L}(\sigma) \frown \text{Min}(\mathbb{A}^*_{\text{P}}) & \text{if } \langle \sigma \curvearrowright_p \text{e} \rangle \in B \wedge \nexists \sigma' \in \Sigma^* : \langle \sigma' \curvearrowright_s \text{e} \rangle \in B; \\ \text{Min}(\mathbb{A}^*_{\text{P}}) \frown \mathscr{L}(\sigma) & \text{if } \langle \sigma \curvearrowright_s \text{e} \rangle \in B \wedge \nexists \sigma' \in \Sigma^* : \langle \sigma' \curvearrowright_p \text{e} \rangle \in B; \\ \mathscr{L}(\sigma) \frown \text{Min}(\mathbb{A}^*_{\text{P}}) \frown \mathscr{L}(\sigma') & \text{if } \{\langle \sigma \curvearrowright_p \text{e} \rangle, \langle \sigma' \curvearrowright_s \text{e} \rangle\} \subseteq B; \\ \text{Min}(\mathbb{A}^*_{\text{P}}) & \text{otherwise.} \end{cases}$$

$\mathbb{G}_{\text{TA}}$ thus returns $\text{Min}(\emptyset)$ if the set of constraints is $\lightning$, the automaton recognizing σ if the constraints bind the value of the expression to σ, the left — resp. right — concatenation between the definite prefix — resp. suffix — and $\text{Min}(\mathbb{A}^*_{\text{P}})$ if they are known, and $\text{Min}(\mathbb{A}^*_{\text{P}})$ otherwise. We can now provide the definition of $\wr\textbf{substr}(\text{s}, A_1, A_2)\wr^{\mathcal{S}^\sharp}$.

Definition 34 (Substring semantics for TA)**.** Let $\mu^\sharp \in \text{TA}$ be a mapping from program variables to $\mathcal{T}\text{FA}_{/\equiv}$ automata, and let $A_1, A_2 \subseteq \overline{\text{BE}}(\text{e})$ be sets of constraints describing integer expressions. Furthermore, let us denote as i^+ and i^- the maximal and minimal values admitted by A_1, and by j^+ and j^- the maximal and minimal values admitted by A_2, with i^+ and j^+ possibly equal to $+\infty$. Function $\wr\textbf{substr}(\text{s}, A_1, A_2)\wr^{\text{TA}}$ is defined as:

$$\wr\textbf{substr}(\text{s}, A_1, A_2)\wr^{\text{TA}}\mu^\sharp = \begin{cases} \text{Min}(\emptyset) & \text{if } \wr\text{e}\wr^{\text{TA}}\mu^\sharp = \text{Min}(\emptyset) \vee A_1 = \lightning \vee A_2 = \lightning; \\ \text{A}[[i^-, i^+] : [j^-, j^+]] & \text{if } \wr\text{e}\wr^{\text{TA}}\mu^\sharp = \text{A}. \end{cases}$$

$\wr\textbf{substr}(\text{s}, A_1, A_2)\wr^{\text{TA}}$ thus directly delegates to the substring semantics of Tarsis using the intervals $[i^-, i^+]$ and $[j^-, j^+]$ as indices, since the domain defines its semantics in terms of intervals. Soundness of $\mathbb{C}_{\text{TA}}$, $\mathbb{G}_{\text{TA}}$, and $\wr\textbf{substr}(\text{s}, A_1, A_2)\wr^{\text{TA}}$ is proven in Appendix A.5.6.

6.7 Products of abstract domains

The formalizations given in this section all refer to individual domains. However, it is possible to abstract a single data type using a product of abstract domains, e.g., as in Madsen and Andreasen (2014). The extension to such a setting is straightforward: $\mathbb{G}$ and $\wr\textbf{substr}(\text{s}, A_1, A_2)\wr$ are simply applied domain-wise, making each produce abstract elements independently, possibly followed by reductions. Instead, since $\mathbb{C}$ must produce a minimal set of constraints containing no contradictions, the result of the individual constraint generations from each domain must be composed together to eliminate redundant constraints, collapsing the result to $\lightning$ if a contradiction is found.

7 Implementation and evaluation

The framework presented in this study has been implemented in LiSA (Negrini et al., 2023a), an open-source[1] Java library designed to ease the creation of static analyzers and offering a common platform for the development of abstract interpretations (see, e.g., Zanatta et al., 2025; Negrini et al., 2024; Olivieri et al., 2023; Negrini et al., 2023b). Among these, Negrini et al. (2021) provided a comparison between five string abstractions on four expressive string manipulation programs, with the objective of proving assertions. The considered abstractions were the prefix, suffix, character inclusion, and bricks domain from Costantini et al. (2015), the FSA domain from Arceri et al. (2020), and the Tarsis abstract domain presented in that study. The comparison was performed by building, for each domain, a *smashed product* between the string abstraction, the interval domain, and the Boolean powerset abstraction. The semantics of each domain were thus coded explicitly for that combination. Some domains were, however, formalized with respect to integer constants instead of intervals, and thus required their semantics to be explicitly lifted beforehand.

In this section, we (i) discuss the implementation effort to code both the smashed product and the constraint-based analysis in LiSA, and (ii) replicate the experiments from Negrini et al. (2021) to ensure that the constraint-based analysis can achieve the same precision as the smashed product. Specifically, we employed the following domains:

- the constant propagation domain CP and the interval domain INTV as numeric abstractions;
- the Boolean powerset domain BP as Boolean abstraction;
- the prefix domain PR, the suffix domain SU, the character inclusion domain CI, the bounded string set domain SS (with $k = 5$), and the Tarsis domain TA as string abstractions.

Note that our evaluation includes two additional domains SU and CI, and that each domain supports additional multi-type expressions with respect to the ones defined in this study (e.g., the `index` operator discovering the first index where a search string appears in a target string). These have been omitted for conciseness, as they would not add any technical contribution.

7.1 Implementation in LiSA

In LiSA, each abstract domain is implemented as a separate Java class that must implement some key interfaces. All domains chosen for the evaluation are non-relational (i.e., they do not explicitly track relations across different variables' values): thus, they all implement the `NonRelationalValueDomain` interface, which requires the definition of lattice operators for individual abstract values (e.g., operating on single intervals) and of the evaluation logic for expressions given the value of each

1 LiSA's GitHub repository.

```
func Subs(nondet bool) {
  res := "substring test"
  if (nondet) {
    res = res + " passed"
  } else {
    res = res + " failed"
  }
  res = res[5:18]
  assert (strings.Contains(res, "g"));
  assert (strings.Contains(res, "p"));
  assert (strings.Contains(res, "f"));
  assert (strings.Contains(res, "d"));
}
```

a.

```
func ToString(names []string) {
  res := "People : {"
  i := 0;
  for i < len(names) {
    res = res + names[i];
    if i != len(names) -1 {
      res = res + ",";
    }
    i++;
  }
  res = res + "}";
  assert (strings.Contains(res, "People"));
  assert (strings.Contains(res, ","));
  assert (strings.Contains(res, "not"));
}
```

b.

```
func Loop(value string, nondet bool) {
  res := "Repeat: "
  for nondet {
    res = res + value + "!"
  }
  assert (strings.Contains(res, "t"));
  assert (strings.Contains(res, "!"));
  assert (strings.Contains(res, "f"));
}
```

c.

```
func CountMatches(nondet boolean) {
  str := ""
  if nondet {
    str = "this is the thing";
  } else {
    str = "the throat";
  }
  count := strings.Count(str, "th");
  assert (count>0);
  assert (count==0);
  assert (count==3);
}
```

d.

FIGURE 7
Program samples used for domain comparison. **(a)** Program Subs. **(b)** Program ToString. **(c)** Program Loop. **(d)** Program `CountMatches`.

variable. Abstract transformers for statements (e.g., assignments) are handled modularly by the `ValueEnvironment` class, which lifts instances of the non-relational domains to maps from variables to abstract values.

Both the smashed product[2] and the constraint-based analysis[3] have been implemented as instances of `NonRelationalValueDomain` as well, each holding references to the client domains that they use for type-specific computations. The implementations are fairly similar both in complexity and length (364 and 357 lines, respectively). The per-domain effort instead varies: the methods implemented for the smashed product are typically shorter, but they require non-trivial reasoning; instead, methods for the constraint-based analysis are fewer and simpler (mainly consisting of iterations over the constraint sets), but are longer. The sizes, however, remain comparable (e.g., the largest difference is seen in Tarsis, where the constraint-based analysis requires writing ∼50 more lines of code), indicating that the constraint-based analysis does not require additional development efforts. Note, however, that LiSA is built to be as modular as possible: all domains are made to be pluggable and, to some extent, replaceable with no code modifications. This is reflected in the non-relational smashed product implementation as well: relational domains would not fit within the current communication scheme, and would require additional complexity. Instead, the constraint-based analysis features modular communication by design, and its structure is expected to be preserved regardless of the chosen domains.

7.2 Comparison with the smashed product

The experiments of Negrini et al. (2021) were run on the program samples visible in Figure 7, written in Go (where the code of `strings.Count` used in program `CountMatches` is visible in Figure 8). For each combination of the domains reported at the beginning of this section, we ran the analysis with the smashed product-based combination and with the constraint-based combination presented in this work, ensuring that the assertions they can prove are the same.

The results of each analysis are visible in Table 1, where column **Domain** reports the combination of abstract domains used and, for each analyzed program, columns **Smash** and **Constr** report the assertions' results (where ✓ denotes an assertion that never fails, ✗ an assertion that always fails, and ✶ an assertion that might fail, according to the invariants computed by the analysis) with the smashed product and the constraint-based analysis, respectively. Invariants computed by each analysis

2 Implementation of the smashed product is publicly available inside LiSA.

3 Implementation of the constraint-based analysis is publicly available inside LiSA.

are omitted for space reasons, but were nonetheless manually inspected at each program point. The experiments highlight that our constraint-based approach yields the same precision as the explicit smashed product: in fact, LiSA was able to determine the same result (never fails, always fails, or might fail) for all assertions in the target programs. Moreover, during the manual inspection of the invariants, we did not find differences in the states produced by each domain in the smashed product and constraint-based analysis, thus justifying the observed equality in the assertions' results.

8 Related work

The problem of combining heterogeneous abstractions to perform whole-value analyses is not new, as it is implicitly formalized in all string analysis definitions. However, formalizations are typically tailored to a specific combination of abstract domains.

```
1  func Count(s, substr string) int {
2    if len(substr) == 0 {
3      return len(s) + 1
4    }
5    n := 0
6    for true {
7      i := strings.Index(s, substr)
8      if i == -1 {
9        return n
10     }
11     n++
12     s = s[i+len(substr):]
13   }
14 }
```

FIGURE 8
The strings.Count function of the Go API.

Christensen et al. (2003b) introduces JSA, an analyzer for Java that analyzes strings by building flow graphs expressing how string sources (i.e., constants and user inputs) are manipulated along the program execution, with the objective of validating the structure of dynamically-generated content like XML documents or the targets of reflective calls. Christensen et al. (2003a) uses such analysis to further validate web services generating HTML pages. In such flow graphs, non-string arguments used in string expressions are treated as part of the operators' definition, e.g., `setCharAt(0, "x")` and `setCharAt(1, "x")` are seen as two different operators. Thus, no reasoning on how to combine the flow graphs with domain modeling, additional data types is given. Further investigating document validation techniques, Kim and Choe (2011) employs a domain based on pushdown automata, but only defines concatenation as an abstract operation. A notable effort by the community targeted the analysis of dynamic property accesses in JavaScript code. Park et al. (2016) defines a domain based on regular expressions. Madsen and Andreasen (2014) reports several existing string domains, but also defines new ones: Length Hash, (Sliding) Index Predicate, String Hash, Number Strings, and Type Strings, and reasons on their combination. In both works, the authors only formalize the semantics of equality tests and string concatenation, with no reasoning for the usage of the presented domains in whole-value analyses.

Several general-purpose string abstractions have been proposed over the years, which are the most common source of whole-value analyses. Costantini et al. (2015) formalizes the Character Inclusion, Prefix, Suffix, Bricks, and String Graph domains, tracking increasingly complex non-relational information on string values. Each domain defines the semantics of `substring` using integer coefficients, and the semantics of `contains` that returns an element of the Boolean powerset. However, no discussion on what the result of the `substring` is when the indexes are not constant is given. Instead, Arceri et al.

TABLE 1 Proved assertions by each domain combination on the target programs.

Domain	Subs		ToString		Loop		CountMatches	
	Smash	Constr	Smash	Constr	Smash	Constr	Smash	Constr
Bp-Cp-Ss	✓✶✶✗	✓✶✶✗	✶✶✶	✶✶✶	✶✶✶	✶✶✶	✓✗✶	✓✗✶
Bp-Cp-Ci	✶✶✶✶	✶✶✶✶	✶✶✶	✶✶✶	✓✶✶	✓✶✶	✶✶✶	✶✶✶
Bp-Cp-Pr	✓✶✶✶	✓✶✶✶	✶✶✶	✶✶✶	✓✶✶	✓✶✶	✶✶✶	✶✶✶
Bp-Cp-Su	✶✶✶✶	✶✶✶✶	✶✶✶	✶✶✶	✶✶✶	✶✶✶	✶✶✶	✶✶✶
Bp-Cp-Ta	✓✶✶✗	✓✶✶✗	✓✶✶	✓✶✶	✓✶✶	✓✶✶	✓✗✶	✓✗✶
Bp-Intv-Ss	✓✶✶✗	✓✶✶✗	✶✶✶	✶✶✶	✶✶✶	✶✶✶	✓✗✶	✓✗✶
Bp-Intv-Ci	✶✶✶✶	✶✶✶✶	✶✶✶	✶✶✶	✓✶✶	✓✶✶	✶✶✶	✶✶✶
Bp-Intv-Pr	✓✶✶✶	✓✶✶✶	✶✶✶	✶✶✶	✓✶✶	✓✶✶	✶✶✶	✶✶✶
Bp-Intv-Su	✶✶✶✶	✶✶✶✶	✶✶✶	✶✶✶	✶✶✶	✶✶✶	✶✶✶	✶✶✶
Bp-Intv-Ta	✓✶✶✗	✓✶✶✗	✓✶✶	✓✶✶	✓✶✶	✓✶✶	✓✗✶	✓✗✶

(2020) uses finite state automata (FSA) to model string values, and formalizes several string operations including `substring`, `length`, and `startsWith`. Boolean values are represented using the Boolean powerset, and numerical values using intervals. Negrini et al. (2021) presents TARSIS, an evolution of the FSA domain that uses automata built over an alphabet of strings instead of characters, aiming at providing the same precision while requiring fewer resources. In terms of whole-value analyses, it still uses the same abstractions for Booleans and integers. Choi et al. (2006) defines the REGULAR STRINGS domain, a subset of the regular expressions with efficient widening. The domain is defined in conjunction with a simple constant propagation domain for numerical quantities, and the authors exploit it in their definition of `substring`. Li et al. (2015) instead introduces a string-specific intermediate representation (IR) that defines data dependencies between string variables, with different levels of context sensitivity. The IR can then be analyzed with several string domains that just have to provide transformers for the IR constructs. Rather than a novel analysis, this work constitutes a framework for the definition of new analyses. Regardless, it is not clear how non-string values appear within the IR, and how client abstractions can leverage them.

Several other string analysis techniques exist in the realm of symbolic execution (Veanes, 2013; Dalla Preda et al., 2015; Han et al., 2011; Yu et al., 2014; Nguyen et al., 2011; Yu et al., 2008) and constraint solving (Abdulla et al., 2020; Chen et al., 2019; Zheng et al., 2013; Abdulla et al., 2019, 2014; Amadini et al., 2020; D'Antoni and Veanes, 2013; Wang et al., 2018), but they are orthogonal to our work and are thus not discussed.

Many techniques for combining domains have also been presented. Cortesi et al. (2013) reports the most common instances of products (Cartesian, reduced, Granger, and open) that can be used to combine arbitrary sets of abstract domains, possibly exchanging information between them. Our framework is an instance of the open product. Amadini et al. (2018) introduces a framework aiming at replacing Granger products, achieving a precision closer to that of the reduced product with fewer computational requirements. The framework is based on the choice of a *reference domain*, i.e., a domain that is at least as precise as the domains involved in the product. Refinement between domains is replaced by converting each domain into an instance of the reference domain, computing their meet, and then converting the meet back to instances of each domain. However, this framework is aimed at products of domains abstracting the same data type rather than different ones. Gulwani and Tiwari (2006) introduces the logical product, a systematic technique to combine logical lattices. This requires the underlying theories to be convex, stably infinite, and disjoint. While the last requirement is always satisfied in our framework, we do not require either convexity or stable infiniteness on the domains we consider (note that the convexity of the set $\overline{\text{BE}}(e)$ is a current limitation on the information exchange, rather than a requirement of the domains one can employ). The Astrée static analyzer and the Verasco formally-verified static analyzer employ communication channels (Cousot et al., 2007; Jourdan et al., 2015) to exchange information between numerical abstractions, allowing mutual refinement between them. The channels act similarly to our functions $\mathbb{G}$ and $\mathbb{C}$: when a domain needs information on the value of some expression, it can query the input channel to obtain it (the form taken by the information can vary: it may be an interval, a linear constraint, …), and the domain's transformers also populate output channels with information other domain can query. While both analyzers use channels to exchange numerical information, their extension to other data types should be feasible. The Apron library (Jeannet and Miné, 2009) also has means of (i) converting a domain instance to a set of constraints, and (ii) creating a (possibly different) domain instance from a set of constraints. Once more, these roughly correspond to functions $\mathbb{G}$ and $\mathbb{C}$, but are only used as a form of conversion between numerical domains instances instead of information exchange. Finally, the Goblint static analyzer uses queries (Apinis, 2014) based on constraints to allow domains to ask for information about expressions from other domains. This workflow is very similar to the one presented in this work, but, to the best of our knowledge, it has no theoretical formalization and has not been employed outside of numerical analyses. All these works were essential in showcasing how constraints can be used effectively to exchange information between different abstract domains, possibly at the cost of some precision. Our framework, and specifically the set $\overline{\text{BE}}(e)$ and the functions $\mathbb{C}$ and $\mathbb{G}$, draws inspiration from them.

9 Conclusion

This study presents a modular framework for constraint-based whole-value analyses where existing domains can be used to provide abstractions of all data types supported by a programming language. The requirements are minimal: domains need to implement means for (i) converting their instances to a set of constraints, (ii) generating instances based on a set of constraints, and (iii) providing the semantics of expressions with heterogeneous argument types in terms of constraints. Provided these requirements are met, several combinations of abstract domains can be executed with no additional modification to their code. Moreover, since the constraint-based information exchange has been proven sound, each combination is guaranteed to be sound without requiring lifting of the abstract semantics, which would in turn require additional soundness proofs. An implementation of the framework is available inside the LiSA static analysis library. The framework has also been compared with *ad-hoc* domain combinations, showcasing the same degree of precision.

Our work explicitly targets a combination of forward analyses that are not compositional [i.e., do not run modularly (Cousot and Cousot, 2002)]. The extension of the framework to both backward and compositional analyses is left as future work. Moreover, the constraints set $\overline{\text{BE}}(e)$ considered allow only for a convex set of constraints: for instance, we are currently unable to express disjoint ranges for a single integer variable. While supporting non-convex sets could be trivial (e.g., by considering sets of sets of constraints instead, where each inner set represents convex information), special care must be employed

to ensure soundness. We thus leave such an extension as future work.

Data availability statement

The original contributions presented in the study are included in the article/Supplementary material, further inquiries can be directed to the corresponding author.

Author contributions

LN: Writing – original draft, Formal analysis, Methodology, Software, Visualization, Conceptualization, Supervision, Validation, Investigation, Writing – review & editing, Project administration.

Funding

The author(s) declared that financial support was received for this work and/or its publication. Work supported by the SERICS (PE00000014 - CUP H73C2200089001) project funded by PNRR NextGeneration EU.

Conflict of interest

The author(s) declared that this work was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Generative AI statement

The author(s) declared that generative AI was not used in the creation of this manuscript.

Any alternative text (alt text) provided alongside figures in this article has been generated by Frontiers with the support of artificial intelligence and reasonable efforts have been made to ensure accuracy, including review by the authors wherever possible. If you identify any issues, please contact us.

Publisher's note

Supplementary material

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fcomp.2025.1655377/full#supplementary-material

References

Abdulla, P. A., Atig, M. F., Chen, Y.-F., Diep, B. P., Dolby, J., Janků, P., et al. (2020). "Efficient handling of string-number conversion," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020* (New York, NY: Association for Computing Machinery), 943–957. doi: 10.1145/3385412.3386034

Abdulla, P. A., Atig, M. F., Chen, Y.-F., Holík, L., Rezine, A., Rümmer, P., et al. (2014). "String constraints for verification," in *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26* (Cham: Springer), 150–166.

Abdulla, P. A., Atig, M. F., Diep, B. P. Holík, L., and Janků, P. (2019). "Chain-free string constraints," in *Automated Technology for Verification and Analysis: 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings 17* (Cham: Springer), 277–293. doi: 10.1007/978-3-030-31784-3_16

Amadini, R., Gange, G., Gauthier, F., Jordan, A., Schachte, P., Søndergaard, H., et al. (2018). Reference abstract domains and applications to string analysis. *Fundam. Inform.* 158, 297–326. doi: 10.3233/FI-2018-1650

Amadini, R., Gange, G., and Stuckey, P. J. (2020). Dashed strings for string constraint solving. *Artif. Intell.* 289:103368. doi: 10.1016/j.artint.2020.103368

Apinis, K. (2014). *Frameworks for Analyzing Multi-Threaded C* (PhD thesis). Technischen Universität München, München.

Arceri, V., and Maffeis, S. (2017). Abstract domains for type juggling. *Electron. Notes Theor. Comput. Sci.* 331, 41–55. doi: 10.1016/j.entcs.2017.02.003

Arceri, V., Mastroeni, I., and Xu, S. (2020). Static analysis for ecmascript string manipulation programs. *Appl. Sci.* 10:3525. doi: 10.3390/app10103525

Béal, M.-P., and Carton, O. (2000). Computing the prefix of an automaton. *RAIRO - Theor. Inform. Appl.* 34, 503–514. doi: 10.1051/ita:2000127

Chen, T., Hague, M., Lin, A. W., Rümmer, P., and Wu, Z. (2019). Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc ACM Program Lang.* 3(POPL), 1–30. doi: 10.1145/3290362

Choi, T.-H., Lee, O., Kim, H., and Doh, K.-G. (2006). "A practical string analyzer by the widening approach," in *Asian Symposium on Programming Languages and Systems* (Cham: Springer), 374–388. doi: 10.1007/11924661_23

Christensen, A. S., Møller, A., and Schwartzbach, M. I. (2003a). Extending java for high-level web service construction. *ACM Trans. Program. Lang. Syst.* 25, 814–875. doi: 10.1145/945885.945890

Christensen, A. S., Møller, A., and Schwartzbach, M. I. (2003b). "Precise analysis of string expressions," in *Static Analysis*, ed. R. Cousot (Cham: Springer Berlin Heidelberg), 1–18. doi: 10.1007/3-540-44898-5_1

Cohen, E. (1977). Information transmission in computational systems. *SIGOPS Oper. Syst. Rev.* 11, 133–139. doi: 10.1145/1067625.806556

Cortesi, A., Costantini, G., and Ferrara, P. (2013). A survey on product operators in abstract interpretation. *Electron. Proc. Theor. Comp. Sci.* 129, 325–336. doi: 10.4204/EPTCS.129.19

Costantini, G., Ferrara, P., and Cortesi, A. (2015). A suite of abstract domains for static analysis of string values. *Softw. Pract. Exp.* 45, 245–287. doi: 10.1002/spe.2218

Cousot, P. (1997). "Types as abstract interpretations," in *Proceedings of POPL '97, POPL '97* (New York, NY: ACM), 316–331. doi: 10.1145/263699.263744

Cousot, P. (2021). *Principles of Abstract Interpretation, Vol. 1*, 1 Edn. Cambridge, MA: MIT Press.

Cousot, P., and Cousot, R. (1977). "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of*

programming languages, POPL '77 (New York, NY: Association for Computing Machinery), 238–252. doi: 10.1145/512950.512973

Cousot, P., and Cousot, R. (1979). "Systematic design of program analysis frameworks," in *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '79* (New York, NY: Association for Computing Machinery), 269–282. doi: 10.1145/567752.567778

Cousot, P., and Cousot, R. (2002). "Modular static program analysis," in *Compiler Construction, Vol. 2304*, eds. G. Goos, J. Hartmanis, J. Van Leeuwen, and R. N. Horspool (Berlin: Springer Berlin Heidelberg), 159–179. Series Title: Lecture Notes in Computer Science. doi: 10.1007/3-540-45937-5_13

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., et al. (2007). "Combination of abstractions in the ASTRÉE static analyzer," in *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, Vol. 4435*, M. Okada, and I. Satoh (Berlin: Springer Berlin Heidelberg), 272–300. Series Title: Lecture Notes in Computer Science. doi: 10.1007/978-3-540-77505-8_23

Cousot, P., and Halbwachs, N. (1978). "Automatic discovery of linear restraints among variables of a program," in *POPL '78*, eds. A. V. Aho, S. N. Zilles, and T. G. Szymanski (New York, NY: ACM Press), 84–96. doi: 10.1145/512760.512770

Dalla Preda, M., Giacobazzi, R., Lakhotia, A., and Mastroeni, I. (2015). "Abstract symbolic automata: Mixed syntactic/semantic similarity analysis of executables," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY: ACM Press), 329–341. doi: 10.1145/2676726.2676986

D'Antoni, L., and Veanes, M. (2013). "Static analysis of string encoders and decoders," in *International Workshop on Verification, Model Checking, and Abstract Interpretation* (Cham: Springer), 209–228. doi: 10.1007/978-3-642-35873-9_14

Ernst, M. D., Lovato, A., Macedonio, D., Spiridon, C., and Spoto, F. (2015). "Boolean formulas for the static identification of injection attacks in java," in *Logic for Prog., Art. Int., Reason.*, eds. M. Davis, A. Fehnker, A. McIver, and A. Voronkov (Berlin: Springer Berlin Heidelberg), 130–145. doi: 10.1007/978-3-662-48899-7_10

Ferrara, P. (2016). A generic framework for heap and value analyses of object-oriented programming languages. *Theor. Comput. Sci.* 631, 43–72. doi: 10.1016/j.tcs.2016.04.001

Gulwani, S., and Tiwari, A. (2006). Combining abstract interpreters. *ACM SIGPLAN Notices* 41, 376–386. doi: 10.1145/1133255.1134026

Han, W., Ren, M., Tian, S., Ding, L., and He, Y. (2011). "Static analysis of format string vulnerabilities," in *2011 First ACIS International Symposium on Software and Network Engineering* (Seoul: IEEE), 122–127. doi: 10.1109/SSNE.2011.9

Jeannet, B., and Miné, A. (2009). "Apron: a library of numerical abstract domains for static analysis," in *Computer Aided Verification, Vol. 5643*, eds. A. Bouajjani, and O. Maler (Berlin: Springer Berlin Heidelberg), 661–667. Series Title: Lecture Notes in Computer Science. doi: 10.1007/978-3-642-02658-4_52

Jourdan, J.-H., Laporte, V., Blazy, S., Leroy, X., and Pichardie, D. (2015). A formally-verified c static analyzer. *ACM Sigplan Notices* 50, 247–259. doi: 10.1145/2775051.2676966

Kim, S.-W., and Choe, K.-M. (2011). "String analysis as an abstract interpretation," in *International Workshop on Verification, Model Checking, and Abstract Interpretation* (Cham: Springer), 294–308. doi: 10.1007/978-3-642-18275-4_21

Li, D., Lyu, Y., Wan, M., and Halfond, W. G. (2015). "String analysis for java and android applications," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (New York, NY: ACM), 661–672. doi: 10.1145/2786805.2786879

Logozzo, F., and Fähndrich, M. (2010). Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. *Sci. Comput. Program.* 75, 796–807. doi: 10.1016/j.scico.2009.04.004

Madsen, M., and Andreasen, E. (2014). "String analysis for dynamic field access," in *Compiler Construction*, ed. A. Cohen (Berlin: Springer Berlin Heidelberg), 197–217. doi: 10.1007/978-3-642-54807-9_12

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

Negrini, L., Arceri, V., Ferrara, P., and Cortesi, A. (2021). "Twinning automata and regular expressions for string static analysis," in *Proc. of VMCAI '21, Volume 12597 of LNCS* (Cham: Springer), 267–290. doi: 10.1007/978-3-030-67067-2_13

Negrini, L., Ferrara, P., Arceri, V., and Cortesi, A. (2023a). "LiSA: a generic framework for multilanguage static analysis," in *Challenges of Software Verification*, eds. V. Arceri, A. Cortesi, P. Ferrara, and M. Olliaro (Singapore: Springer Nature), 19–42. doi: 10.1007/978-981-19-9601-6_2

Negrini, L., Presotto, S., Ferrara, P., Zaffanella, E., and Cortesi, A. (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* (New York, NY: Association for Computing Machinery), 10–17. doi: 10.1145/3689609.3689995

Negrini, L., Shabadi, G., and Urban, C. (2023b). "Static analysis of data transformations in jupyter notebooks," in *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP 2023* (New York, NY: Association for Computing Machinery), 8–13. doi: 10.1145/3589250.3596145

Nguyen, H. V., Nguyen, H. A., Nguyen, T. T., and Nguyen, T. N. (2011). "Auto-locating and fix-propagating for html validation errors to php server-side code," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)* (Lawrence, KS: IEEE), 13–22. doi: 10.1109/ASE.2011.6100047

Olivieri, L., Negrini, L., Arceri, V., Tagliaferro, F., Ferrara, P., Cortesi, A., et al. (2023). "Information flow analysis for detecting non-determinism in blockchain," in *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, *Volume 263 of Leibniz International Proceedings in Informatics (LIPIcs)*, eds. K. Ali, and G. Salvaneschi (Dagstuhl: Schloss Dagstuhl - Leibniz-Zentrum für Informatik). 23:1–23:25.

Park, C., Im, H., and Ryu, S. (2016). "Precise and scalable static analysis of jquery using a regular expression domain," in *Proceedings of the 12th Symposium on Dynamic Languages* (New York, NY: ACM), 25–36. doi: 10.1145/2989225.2989228

Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Trans. Am. Math. Soc.* 74, 358–366. doi: 10.1090/S0002-9947-1953-0053041-6

Veanes, M. (2013). "Applications of symbolic finite automata," in *Implementation and Application of Automata: 18th International Conference, CIAA 2013, Halifax, NS, Canada, July 16-19, 2013. Proceedings 18* (Cham: Springer), 16–23. doi: 10.1007/978-3-642-39274-0_3

Wang, H.-E., Chen, S.-Y., Yu, F., and Jiang, J.-H. R. (2018). "A symbolic model checking approach to the analysis of string and length constraints," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE '18* (New York, NY: Association for Computing Machinery), 623–633. doi: 10.1145/3238147.3238189

Yu, F., Alkhalaf, M., Bultan, T., and Ibarra, O. H. (2014). Automata-based symbolic string analysis for vulnerability detection. *Form. Methods Syst. Des.* 44, 44–70. doi: 10.1007/s10703-013-0189-1

Yu, F., Bultan, T., Cova, M., and Ibarra, O. H. (2008). "Symbolic string verification: an automata-based approach," in *International SPIN Workshop on Model Checking of Software* (Cham: Springer), 306–324. doi: 10.1007/978-3-540-85114-1_21

Zanatta, G., Caiazza, G., Ferrara, P., and Negrini, L. (2025). Inference of access policies through static analysis. *Int. J. Softw. Tools Technol. Transfer* 26, 797–821. doi: 10.1007/s10009-024-00777-8

Zheng, Y., Zhang, X., and Ganesh, V. (2013). "Z3-str: a z3-based string solver for web application analysis," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (New York, NY: ACM), 114–124. doi: 10.1145/2491411.2491456